Assignment 1

Write a shell interpreter for ThreadOS

* Connect to linux machine uw1-320-lab.bothell.washington.edu
* cp ~css430/ThreadOS/\*.class .
* cp ~css430/ThreadOS/DISK .
* java Boot (to start OS)
* Delete Shell.class
* Implement new Shell.class

10/2/2008

The operating system maintains a linked list of program control block structs containing process-specific info (PCBs)

Each process is a program file that resides on the hard drive that you can load into main memory and execute based on the program counter.

OS data structures are called resources.

Switching between processes is called context switching, which involves overhead.

stdin, stdout, and stderr are files stored in a file descriptor table

Pipes are files stored in the file descriptor table.

forking copies the file descriptor table, including stdfoo

dup2 rewrites file descriptors

Context switch: operating system puts you to sleep, executes another program, and returns to you. It’s expensive.

To get Unix source code samples (dup2() usage, etc)

* cd unix/
* cp –r ~css430/ExampleSourcecode/1.UnixExamples/\* .

Threads get their own stack but share code with the parent process

Assignment 2:

Test cases:

TCB.java

Test2.java

TestThread2.java

10/14/2008

Program 1 review: two processes waiting on STDIN from a pipe will generate unexpected results. Also, because I/O is blocking, you don’t need to (and shouldn’t) wait for a process to exit before reading the pipe. If the child process hangs, you still want to process what’s in the pipe and you don’t want to hang the parent.

Process: has code, stack and heap segments

Thread: only has stack section – shares code and heap with owning process.

Scheduler: submits jobs to one or more TCB queues

10/16/2008

Scheduler: job = thread, controlled via thread control blocks (TCB)

Algorithms:

* FIFO/FCFS - can be inefficient if long-running jobs are not scheduled at the end; turnaround time is fairly bad
* Shortest job first (SJF), preemptive and non-preemptive
* Rount-Robin – schedule a job until it’s interrupted by I/O or until its timeslice expires, then schedule the next one. Doesn’t keep track of whether a job uses its whole timeslice.
* Multilevel feedback queue – maintain queues, watch who uses their entire timeslice. Those that do get demoted, and demoted again if necessary. 3 queues; final queue is FIFO.

Timeslice length: dynamically adjusted based on average CPU burst

Locality: temporal and spatial

Metrics: turn-around time (job completion time); wait time (before execution); CPU utilization, both %running and % devoted to user processes; throughput (# jobs completed per time unit)

Program2: 3 queues, behavior is the same between the three of them. class JobQueue

10/21/2008:

Synchronization.

Context switches cause value swaps at the register value. If multiple threads are allowed to operate on the same variable, unexpected results will happen.

Critical sections – one thread can run

Mutual Exclusive area (mutex): one at a time

Peterson’s solution: implemented with

1. Identify and lock mutex area
2. Wait your turn
3. Critical section must eventually finish

Machine guarantees that load/store are atomic

Peterson’s algorithm (works for only two threads):

int flag[2]

int turn = 1;

Thread 1: while (flag[1] && turn == 1); // spin-lock/spin-wait /busy-wait: blocked thread is doing work.

Thread 2: while (flag[0] && turn == 0);

while (Lock.test(True) yield; - test() /set or swap instructions are atomic actions so they are thread-safe

Dijkstra called the process of AcquireLock() “performing a P() operation” and ReleaseLock “performing a V() operation). Essentially, we disable the context switch.

Semaphore:

P(): disable interrupt, spin-lock on a counter, decrement a counter, enable interrupt

while (count <= 0); count --;

V():increment a counter: count++;

10/23/2008

Response time: elapsed time between thread construction and thead.start() OR the elapsed time between a user action and an indicator that the action has succeeded.

Java synchronized keyword: adds a synchronized block around the entire method.

11/4/2008

Dining Philosophers: tight loop of 5 processes sharing 5 resources. Each process needs 2 resources to run.

state[5];

syncObj[5];

/\* Do work \*/

takeFork(i) // Acquire resources for i - 1 and i - 1

state[i] = hungry;

test(i); // If my neighbors aren’t running, I can run.

if (state[i] != eat) syncObj[i].wait();

/\* Finish doing work \*/

returnfork(i);

state[i] = think;

test(“left”);

test(“right”);

/\* test implementation \*/

test(int i)

{

if (left != eat && right != eat && hungry) {state[i] = eat; syncObj[i].notify()}

}

while (true)

{

// do work

takeFork(4);

doMutexWork();

returnFork(4);

}

Deadlock:

processes that wait on a shared sync object held

there will be a deadlock question on the final.

Resource allocation graph: Boxes are resources, circles are proceses

A

C

B

Deadlock class

Mutex[4]

|  |  |  |  |
| --- | --- | --- | --- |
| . | . | . | . |

11/6/2008

To avoid deadlocks, 4 conditions must be satisified:

1. Mutually-exclusive regions
2. Hold + Wait
3. Pre-emption (break other people’s locks)
4. Circular wait: locking in-order

Filesystem:

Characteristics – data and attributes (metadata) (owner/permissions

File organization

Virtual FS

File commands: open(), close(). An open file has a ‘current’ ptr, which is modified with seek() and append()

Directory: acyclic graph

FAT:

**Design**

The following is an overview of the order of structures in a FAT partition or disk:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot sector | More reserved sectors (optional) | File Allocation Table #1 | File Allocation Table #2 | Root Directory (FAT12/16 only) | Data Region (for files and directories) ... (To end of partition or disk) |

A FAT file system is composed of four different sections.

1. The **Reserved sectors**, located at the very beginning. The first reserved sector is the [Boot Sector](http://en.wikipedia.org/wiki/Boot_sector) (aka *Partition Boot Record*). It includes an area called the [*BIOS Parameter Block*](http://en.wikipedia.org/wiki/BIOS_parameter_block) (with some basic file system information, in particular its type, and pointers to the location of the other sections) and usually contains the operating system's [boot loader](http://en.wikipedia.org/wiki/Boot_loader) code. The total count of reserved sectors is indicated by a field inside the Boot Sector. Important information from the Boot Sector is accessible through an operating system structure called the *Drive Parameter Block* in DOS and OS/2. For FAT32 file systems, the reserved sectors include a *Backup Boot Sector* at Sector 6.
2. The **FAT Region**. This typically contains two copies (may vary) of the *File Allocation Table* for the sake of redundancy checking, although the extra copy is rarely used, even by disk repair utilities. These are maps of the Data Region, indicating which clusters are used by files and directories.
3. The **Root Directory Region**. This is a *Directory Table* that stores information about the files and directories located in the root directory. It is only used with FAT12 and FAT16 and means that the root directory has a fixed maximum size which is pre-allocated at creation of this volume. FAT32 stores the root directory in the Data Region along with files and other directories instead, allowing it to grow without such a restraint.
4. The **Data Region**. This is where the actual file and directory data is stored and takes up most of the partition. The size of files and subdirectories can be increased arbitrarily (as long as there are free clusters) by simply adding more links to the file's chain in the FAT. Note however, that files are allocated entirely in a cluster, and so if a 1 KB file resides in a 32 KB cluster, 31 KB are wasted.

The **File Allocation Table** (**FAT**) is a list of entries that map to each cluster on the partition. Each entry records one of five things:

* the cluster number of the next cluster in a chain
* a special *end of clusterchain* (*EOC*) entry that indicates the end of a chain
* a special entry to mark a bad cluster
* a special entry to mark a reserved cluster[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]
* a zero to note that the cluster is unused

Each version of the FAT file system uses a different size for FAT entries. The size is indicated by the name, for example the FAT16 file system uses 16 bits for each entry while the FAT32 file system uses 32 bits. Only 28 of these are actually used, however. This difference means that the File Allocation Table of a FAT32 system can map a greater number of clusters than FAT16, allowing for larger partition sizes with FAT32. This also allows for more efficient use of space than FAT16, because on the same hard drive a FAT32 table can address smaller clusters which means less wasted space.

11/13/2008

1. Filesystem
   1. Data structure + storage on disk
      1. FCB: Metadata owned by filesystem (OS), contains pointer to data
         1. Directory information stores file control blocks
      2. Data owned by you
   2. Cache: In-memory file system
   3. Allocation scheme
      1. Contiguous – historical, not really used anymore
      2. Linked (FAT, NTFS)
         1. A directory entry is a File Control Block (FCB).
         2. Boot sector, two FATs, Root directory (address 0), then blocks (1 –n): FAT 12 = 2^12 allocatable units
         3. Understand the amount of address space \* size of a block for max file size
      3. NTFS
         1. Has a Master File Table for allowing File Control Blocks to be linked together to store unlimited metadata. It can contain 2^48 entries and can be allocated incrementally.
         2. Each MFT entry is a FCB.
         3. Permits user-defined metadata as name-value pairs.
         4. Can consume multiple MFT entries per file.
         5. File data can be stored within the metadata space.
         6. Files are addressed through an abstraction layer – logical/virtual clusters are mapped to physical clusters.
         7. Each file has a 48-bit index into the MFT.
      4. Indexed (UFS)
         1. INode == FCB. Filename is not stored on the inode, but data block# is. It also contains a pointer to a table of pointers that point to other blocks in the file. This allows indexed access into any part of the file.
   4. Free blocks
      1. Stored as a linked list of blocks in UFS
2. Address binding
   1. use a base register to set instruction execution start point
   2. Historically: at compile time. Now, at load time or at runtime (dlls)
3. Memory management
   1. Linked list of frames (memory blocks)
   2. Frames are managed by pages (logical address into the page table, which maps between pages and frames)
   3. Each process has its own page table.
   4. Individual bytes within each page can be loaded with a page index/page offset addressing scheme. Logical addresses are converted into a page table index and a byte offset.   
      For example:  
      8 possible pages (requires 3 bits to address)  
      4 bytes per page (requires 2 bits to address)  
      Logical address 4 = page 1, logical address 00 (100, or 4 in binary)
   5. The page table is located at the page table base register (PTBR), stored inside the TCB/PCB.
   6. Page table length register (PTLR) stores the length of the pagetable.
   7. Translation between logical and physical addresses requires two memory reads. This can be optimized by caching the current page’s logical and physical address in a Translation Look-aside Buffer (TLB). Successful cache lookup is a “hit’, otherwise it’s a “miss”.
   8. Address of a process is continuous. Addressable process space is bounded by the page table address + offset length, and addressed with the program counter (PC).
   9. Stack, heap, and code space all have their own page table.
   10. Multiple levels of pagetable indirection can be used to store blocks from different parts of a program. This gives us space savings, since addresses can span wide ranges without having to have reserved memory for the entire range.
   11. Segmentation involves using a segmentation offset to lookup a base address in a table. The base address is added to the offset to get a page table + offset address. This allows us to flag a segment as code or data for security purposes. See notes about resolving segmented addreses.

12/4/2008

A page fault happens when a page must be loaded from the swap file. It’s very expensive. Pages that aren’t in memory have a value of -1.

Paging/frame-sharing algorithms:

FIFO

Optimal (purely theoretical) – swap the page that won’t be used

Least Recently Used: do not swap the most recently used frames from the past

Working set:

Page fault frequency: cap running jobs when page fault frequency goes above a threshold

Working set: detect scarce resources by summing the number of pages being accessed by processes and comparing that to system resources