4 Implementation

Copyright © 2006 Stephen Fewer

4.1 Development Environment

The development environment for AMOS is comprised of several freely available, open source applications that make up the development tool chain, the layout of which is described in Figure 1 below.

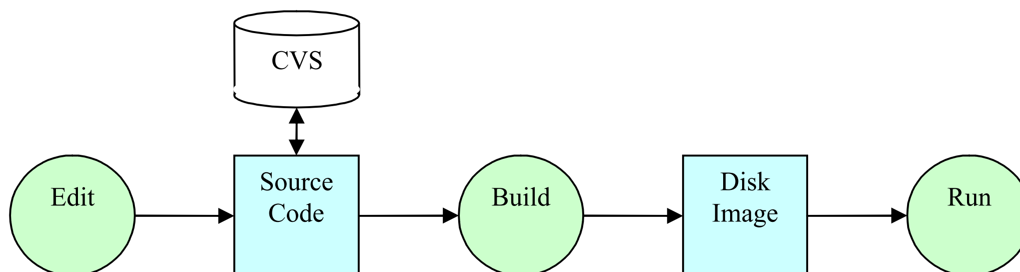

Figure 1

C is the primary language chosen for development of AMOS. C is easily portable across multiple architectures while also being a “mid level” language allowing for low level operation such as pointer manipulation and assembly language integration both of which are essential for developing an operating systems kernel. Assembly language will be used where required for the lowest level operations such as interrupt stubs and enabling paging on the processor. This will be x86 assembler as the currently targeted hardware architecture is x86.

4.1.1 Edit

To edit the source code I chose to use IBM Eclipse[1] with the CDT[2] plugin for C language support. I have configured Eclipse to use a custom build setup for non-standard application development such as OS development allowing for the source code to the kernel, user applications and libraries to be compiled, linked and the systems disk image generated in a single automated process. Eclipse has built in support for the Concurrent Versions System (CVS), a version control system which allows for the tracking of all versions of the projects files throughout its lifecycle.

[1] http://www.eclipse.org/
[2] http://www.eclipse.org/cdt/

AMOS is hosted on SourceForge.Net[3] which provides a CVS server for projects. This allows for a high level of protection against loss of code due to disk failure or software error on my own development machines while allowing me to develop across multiple machines, such as home or college, merging the code together via CVS at a later date.

Running the software metrics tool cccc[4] over the source code tree we find that there are 5842 lines of code and 1485 lines of comments in the current version of AMOS.

4.1.2 Build

The build tool chain is largely composed of the Windows port of the GNU Compiler Collection[5] (GCC) which provides a versatile C compiler and linker allowing us to produce executable files and libraries in multiple formats such as ELF for the kernel and flat binary for the user applications. The NASM[6] assembler is used for the x86 assembly code portions of the project. The mtools[7] package allows for the creation and manipulation of FAT disk images. This allows us to automatically generate a bootable floppy disk image of the system which we can run.

4.1.3 Run

To run the bootable floppy disk image on a real machine we can transfer the image to a floppy disk with a utility such as rawwrite[8]. For development testing a much faster technique is to use a virtual machine. Bochs[9] and QEMU[10] allow us to run AMOS in a virtual machine on our development system and allow for the collection of basic debug information such as the CPU state upon critical problems like a triple fault

[3] http://sourceforge.net/projects/osamos
[4] http://cccc.sourceforge.net/
[5] http://www.delorie.com/djgpp/
[6] http://nasm.sourceforge.net/
[7] http://mtools.linux.lu/
[8] http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm
[9] http://bochs.sourceforge.net/
[10] http://fabrice.bellard.free.fr/qemu/

Copyright © 2006 Stephen Fewer

which is vital for debugging and unavailable on a standalone computer. We can also specify the hardware specifications of a virtual machine such as the amount of RAM, CPU type or peripherals attached to further aid development and testing.

4.2 Kernel - `/src/kernel/kernel.c`

The GRUB[11] boot loader loads the AMOS kernel into physical memory and passes control over to a small assembly stub called `_setup` in the file `loader.asm` which enables basic paging so as we may jump to the virtual address where the kernel has been linked to run at. We enter the kernel in the function `kernel_main()` where we bring up the system as shown in Figure 2 below.

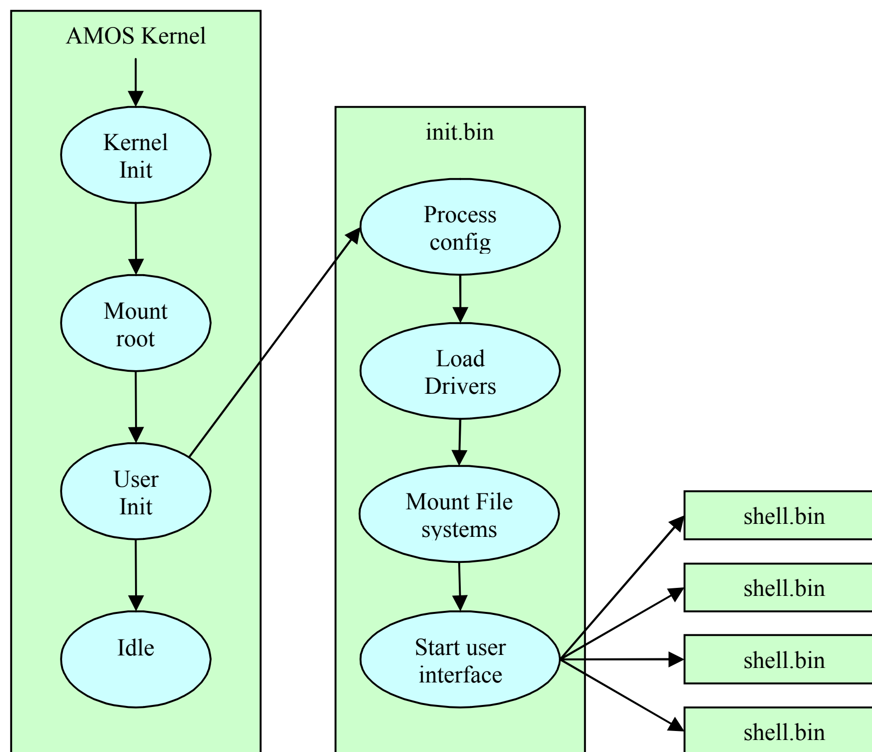

Figure 2

The first steps in bringing up the system involve initializing all the subsystems; we must do this in the following order as they are each dependant on the previously initialized subsystems.

1. `Interrupts`
2. `Memory Manager`
3. `Virtual File System`
4. `Input Output`

[11] http://www.gnu.org/software/grub/

```
5. System Calls
6. Scheduling
```

After the subsystems have been initialized the kernel, now executing as the kernel process, will mount the default root file system, which will be a FAT file system on the first floppy drive. A root file system must be present to allow the kernel to spawn further applications, such as init.bin which is the user mode initialization process. The kernel will then become then systems idle process, discussed later in the process manager section.

The user mode initialization program init.bin relies solely on the system calls to perform the final system initialization. It is left as a separate user process to perform the final initialization so as the kernel need not be modified and reinstalled should the user wish to modify the systems setup. This init process can allow the following tasks to be performed, such as processing configuration files, loading further device and file system drivers, mounting further file systems, spawning background processes and starting a user interface which could be either graphical or textual. The init process in the current version of AMOS starts the console user interface which comprises of spawning a shell application on each virtual console present in the system.

4.2.1 Interrupts - `/src/kernel/interrupt.c`

The interrupt layer gives us full control over how we wish to process interrupts in the system, allowing us to enable and disable specific interrupts at will. When enabling an interrupt we can associate an interrupt handler with it. An interrupt handler is a C function that will perform the required operation when the interrupt it has been bound to has fired. The setup of the interrupt layer is described in Figure 3 below. A series of interrupt service routines (ISR's) are created in assembly for each interrupt in the system. On the x86 architecture there are 32 software (such as a page fault or divide by zero) and 16 hardware (such as timer or floppy drive) interrupts which we must provide for. We also use software interrupt number 48 for system calls, as discussed later. The address of every ISR is loaded into the processors Interrupt Descriptor Table (IDT), so as the processor will transfer control over to the correct ISR upon generating an interrupt. The setup of the IDT occurs in the `interrupt_init()`

function. We also specify a privilege level for each ISR registered; either USER or KERNEL (privilege levels are discussed in detail in the segmentation section in the memory manager). This protects user code from attempting to trigger a KERNEL interrupt. In AMOS the only USER level interrupt is the system call interrupt (48) so as it may be called from user mode. Upon executing an ISR it registers its interrupt number on the stack before calling a common dispatcher function `interrupt_dispatcher()`. The dispatcher will call the correct handler (based on the number left on the stack) should one exist or else perform an appropriate default action should a handler not exist. If an unhandled software interrupt occurs we default to killing the offending process in order to keep the system stable. If an unhandled hardware interrupt occurs we signal its successful completion and return to the process that was executing before the interrupt occurred.

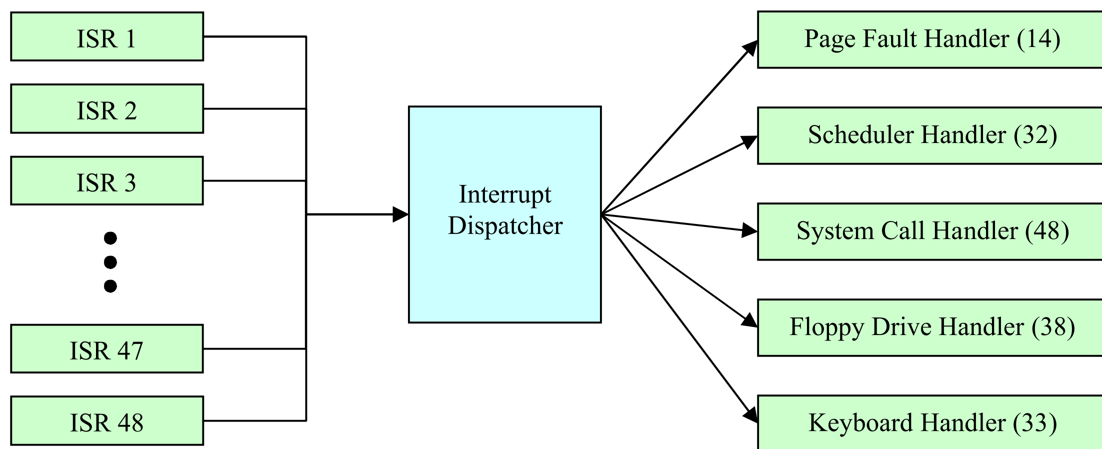

Figure 3

This brings us on to how the processors interrupt mechanisms for pausing and resuming the currently executing process are linked to how we perform context switching in AMOS. AMOS uses software task switching which relies on saving a processes state on its stack and upon returning from an interrupt, its state is automatically restored from its stack. By switching the processors stack pointer to that of another process we can automatically perform a context switch into that process upon returning from an interrupt.

Copyright © 2006 Stephen Fewer

4.2.2 System Calls - `/src/kernel/syscall.c`

System calls export an interface from the kernel to user mode processes via a software interrupt. Each system call such as `open()` or `read()`is specified by a number. This number is stored in one of the processors registers (The accumulator register EAX) so as the system call handler function `syscall_handler()` can dispatch the request to the correct system call wrapper by looking up the entry specified by the number in the system call table. In `syscall_init()` we construct the system call table with each entry pointing to the respective system call wrapper function. We then must register the system call handler with the interrupt layer as a user mode interrupt bound to software interrupt number 48.

The following system calls as shown in Table 1 are available in the current version of AMOS and are discussed in detail in the proceeding sections.

File Operations	**File System Operations**	**Process Operations**
open()	mount()	morecore()
close()	unmout()	spawn()
clone()	create()	kill()
read()	delete()	sleep()
write()	rename()	wake()
seek()	copy()	wait()
control()	list()	exit()

Table 1

4.2.3 User Applications

The operating system itself is comprised of the kernel together with a number of applications and a standard C library.

- **init.bin** - `/src/apps/init/`

  This is the user mode initialization process as described previously.

- **shell.bin** - `/src/apps/shell/`

  This is a console shell application that allows the user to interface with the system by providing a set of commands such as spawn, list or delete which directly correspond to their respective systems calls. Further commands are available to perform operations such as dumping a file's content to screen or writing a buffer of data to a file. The shell also supports auto command completion by pressing the tab key aswell as a command history which may be cycled through using the up and down arrow keys. It is based upon the Tiny Shell[12] library which is an open source library to aid the construction of a shell application. This is an example of how third party code was easily ported over to AMOS.

- **test.bin** - `/src/apps/test/`

  This is a test application for performing a series of controlled illegal operations to test the systems stability, and not strictly part of the operating system. The illegal operations include:

  1. Generating a General Protection Fault by performing an illegal instruction.
  2. Generating a Page Fault by attempting to write to the kernel heap.
  3. Attempting to divide by zero.
  4. Forcing a stack overflow.
  5. Attempting to execute an invalid machine instruction.

- **hanoi.bin** - `/src/apps/hanoi/`

  This is another test application and not part of the operating system itself. It is designed to stress test how the system copes under load and should be run as a background process. It will attempt to solve the Towers of Hanoi[13] problem, with all 64 disks.

- **libc.a** - `/src/lib/libc/`

  This is AMOS's standard C library which currently exports a subset of the ANSI C library specifications. It is required by both the kernel and some of the user applications, and during the build process they will be linked against this library.

[12] http://sourceforge.net/projects/tinysh
[13] http://en.wikipedia.org/wiki/Towers_of_Hanoi

Currently it is an unfinished port of the open source Diet Lib C[14] project, with the exception of malloc() which is based on the kernel malloc() I wrote.

[14] http://www.fefe.de/dietlibc/

4.3 Memory Manager - `/src/kernel/mm/`

The memory manager is the first subsystem to be initialized after the interrupt layer, as other subsystems will require it for memory operations like allocations or manipulating address spaces. In `mm_init()` we bring up the physical memory manager before we initialize the segmentation and paging layers.

4.3.1 Physical Memory Manager - `/src/kernel/mm/physical.c`

The physical memory manager must record what pages of physical memory have been allocated or are reserved and what pages are free and available for allocation. A bitmap is used to record this information, with each bit in the bitmap representing a 4KB page of physical memory in ascending order. We first must create and initialise this bitmap. We default to initializing all entries in the bitmap as being marked as used. We then free the physical memory ranges that are available on the system. We receive this information via the boot loader which queries this systems BIOS to retrieve this information. It is also important that we mark the physical memory that the kernel itself is occupying as used so as we do not allocate this memory at a later time, corrupting the kernel in memory. This initialization occurs in the `physical_init()` function. There are two main function exported by this layer:

```
void physical_pageFree( void * physicalAddress )
```

This function will mark the page associated with physicalAddress as being free in the bitmap so it may be allocated in the future.

```
void * physical_pageAlloc()
```

This function will find an available physical page and mark it as allocated in the bitmap before returning the physical address of the page to the caller. The algorithm for locating a physical page works as follows. We first try to locate a free page in high memory. High memory is any address above the 16MB mark. This is a constraint with the x86 architecture as certain hardware features, most notably ISA DMA may not address a memory location above that limit. If we are unable to locate a free page in high memory we then try low memory, below the 16MB mark. If no more physical

memory may be allocated we issue a kernel panic. A sub function `physical_pageAllocLow()` is also exported for explicitly allocating a low page of physical memory for use with DMA operations.

All access to the bitmap, whether for allocating or freeing is marked as a critical section and is protected by a mutex (covered in the Process Manager section) to guarantee we do not allocate the same physical page more then once.

4.3.2 Segmentation - `/src/kernel/mm/segmentation.c`

AMOS does not use segmentation to achieve virtual memory, however on the x86 architecture we are unable to disable it as segmentation is an integral part of the protection mechanisms of x86. To get around this problem while still taking advantage of the protection features, flat segmentation is setup. This means we define each segment as having a base address of zero, the bottom of memory, while having a limit of 4GB, allowing us to address the entire 4GB address space without any segmentation address translation taking place. This means a virtual address in AMOS is the same as a linear address before paging turns the linear address into a physical address. A segment is defined by a segment descriptor held in a structure known as the Global Descriptor Table (GDT) which the processor accesses. AMOS defines two privilege levels for code to operate in; either USER or KERNEL. The x86 architecture defines 4 privilege levels or rings as they are known, called ring 0 through to ring 3. Kernel code will operate at ring 0 while user code will operate at ring 3. The protection features of x86 require us to have two descriptors present for each priority level present, one for code and one for data. Therefore we must setup four descriptors defining our kernel code and data and user code and data. This occurs in `segmentation_init()`. This lets us protect a user processes from performing privileged operations, such as executing a privileged system instructions like disabling interrupts or disabling paging, as shown in Screenshot 1 below.

```
AMOS 0.6.0
--------------------------------------------------------------------------------
AMOS:>spawn /amos/test.bin
spawn: Spawned process 7
Test App
    1. Exit Gracefully!
    2. General Protection Fault
    3. Page Fault
    4. Divide By Zero
    5. Stack Overflow
    6. Invalid Opcode
    7. Loop Forever
Please enter your choice: 2
About to execute a privileged instruction...
[KERNEL] Exception "General Protection Fault" in process 7
[KERNEL]     CS:0x1B EIP:0x100002FE
[KERNEL]     DS:0x23 ES:0x23 FS:0x23 GS:0x23
[KERNEL]     EDI:0x0 ESI:0x0 EBP:0x20000FC8 ESP:0xD0037FF0
[KERNEL]     EBX:0x20000FB4 EDX:0x1000013D ECX:0x1000013C EAX:0x0
[KERNEL]     EFLAGS:0x206  SS0:0x23 ESP0:0x20000FA0
AMOS:>_
```

Screenshot 1

4.3.3 Paging - `/src/kernel/mm/paging.c`

Paging is used to define and manipulate a virtual address space. Each process will have its own virtual address space, unique to that process and as a result many of the paging functions take in a process structure (described in section 4.4) as one of their parameters in order to manipulate that processes address space. The key functions of the paging layer involve creating a page directory, mapping a virtual address to a physical address and destroying a page directory.

First I will discuss the typical layout of a virtual address space as shown in Figure 4 below.

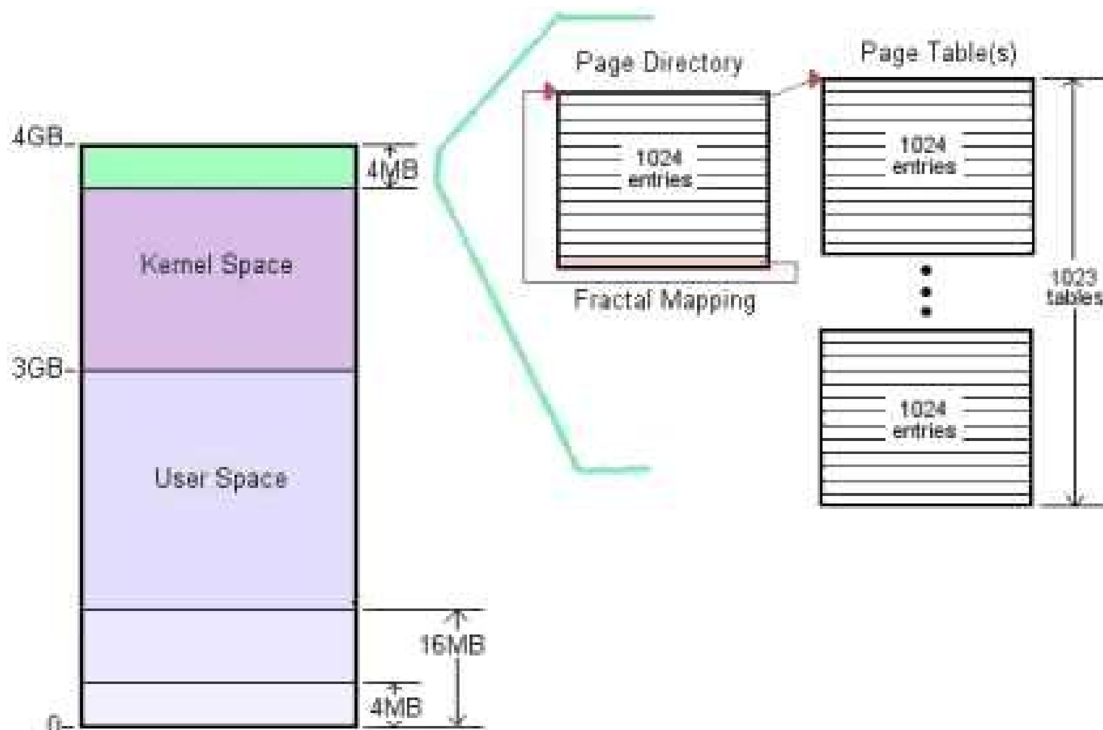

Figure 4

As we are operating in a 32bit architecture the maximum addressable memory location is 2^32 or 4GB's. The kernel is designed as a higher half kernel and is linked to be loaded at the 3GB mark in virtual memory and occupies the top gigabyte of the address space along with the kernel heap, kernel stacks and device drivers. The fractal mapping technique, discussed in detail later, lets you address the page directory and all its page tables for the current address space in a linear fashion via the top 4MB's in the 4GB address space. The top gigabyte is mapped across all address spaces as protected shared memory so as all processes have access to the kernel via the system calls, but are unable to directly modify or execute any of the kernel code or data. This means that when a system call occurs we do not need to switch address spaces which is an expensive operation as the processor would need to flush the entire Translation Look aside Buffer (TLB) and also any pointers to data in the user process would become unavailable and require the data to be copied across address spaces which is also an expensive operation. The bottom 3GB is unique to each process and will contain the user processes code, data, stack and heap.

The creation of a page directory is handled in the function `paging_createDirectory( struct PROCESS_INFO * p )`. We must

allocate a page of physical memory for the page directory itself and associate this with the process specified from the function call. The page directory must then have all its entries marked as not present as when we map in an address at a later stage the respective page table will be created if needed and mapped back into the page directory. This leads us to a problem of manipulating one processes address space from within another processes address space, for example the kernel process spawning a new user process. As the currently executing process can only address physical pages mapped into its current address space via their respective virtual address we are unable to manipulate another processes page directory or page tables unless we map them into the current address space. To preserve the layout of an address space we provide a function to quickly map a physical page into a fixed location in the current address space so we may access it. This function is called `paging_mapQuick()` and relies on the fractal mapping technique discussed next for its optimality.

Fractal mapping allows us to easily address the physical page table address of any page table in the current address space through a fixed virtual address in the top 4MB of the address space. By mapping the page directory as the last entry in itself as shown in Figure 4 we automatically place all the page tables and the page directory itself at virtual addresses in the top 4MB's. As a page table addresses 4MB's of virtual memory and a page directory has 1024 entries allowing it to address 4GB (1024*4MB = 4GB), therefore the last entry in the page directory addresses the top 4MB and by mapping the page directory into itself the page directory is then translated as a page table, automatically mapping each physical address of each page table entry in the page directory to a virtual address in the top 4MB's in a linear fashion. This is possible as the structure that defines a page directory is nearly identical to that of the structure that defines a page table (differing in only 2 bits).

We use the `paging_map()` function to map a physical address to a virtual address in a processes address space. This involves setting the correct entry in the correct page table. The algorithm is as follows:

```
1. We wish to map physical address P to virtual address V.
```

2. Get the page directory entry for V in the processes page directory. If present, this gives us the address of the correct page table to use.
3. If not present, we create a new page table and set the entry in the page directory to point to this page table, marking it as present.
4. We retrieve the correct page table entry for V in the newly found or created page table.
5. We set this page table entry as being present, with read write access, and point it to the physical address P.

The `paging_mapQuick()` must be used to provide read and write access to the retrieved page directory and page tables during the mapping process.

When destroying a page directory with the `paging_destroyDirectory()` function, we free the physical pages that the page directory and its page tables occupy. For a user process we can also free all the physical pages mapped in below the 3GB mark, automatically reclaiming the physical memory that the user processes code, data, stack and heap occupied.

Another important feature that the paging layer provides is the page fault handler which gets called upon the processor generating a page fault. To keep the system stable, the offending process is terminated allowing the system to continue to run. Should a page fault occur in the kernel process itself it is deemed an unrecoverable problem and a kernel panic is issued bringing the system to a halt after displaying as much error information as possible, such as the CPU state. The page fault handler `paging_pageFaultHandler()` is registered with the interrupt layer during the initialization of the paging layer in `paging_init()`. It is in the page fault handler that the page replacement code would be called.

4.3.4 Kernel Heap - `/src/kernel/mm/mm.c`

The kernel heap is where arbitrary sized portions of memory are allocated for use with kernel code. Each user process has a separate and unique user heap. A heap in AMOS

is page aligned and grows upwards in memory as needed via a call to `mm_morecore()`, user processes can expand their heap with the `morecore()` system call which is a wrapper to `mm_morecore()`. To allocate a portion of kernel memory we call `mm_kmalloc()` and to free a portion we call `mm_kfree()`.

Each process has a structure that defines that processes heap, including the base address of the heap and the address of the current top of the heap. To expand the heaps size `mm_morecore()` will calculate the number of pages required to fit the size requested and proceed to allocate pages of physical memory and map them onto the top of the heap, utilizing the underlying physical and paging layers.

The heap is composed of a forward linked list of heap items as shown in Figure 5 below. Each heap item has a pointer to the next item in the heap so we may traverse the heap. A used bit indicates whether the item is used or unused, unused items may be allocated and modified. The size indicated the size of the data portion that follows.

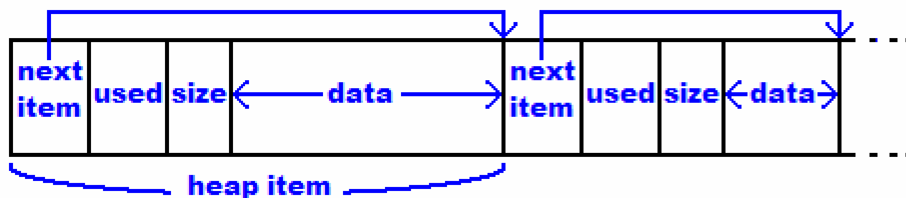

Figure 5

The first fit algorithm used in `mm_kmalloc()` is as follows:

```
1. Search the heap from the bottom upwards for an unused heap item
   X that will fit the request.
2. If we did not find one, expand the heap with morecore() and
   generate item X.
3. Split X into two items Y and Z. Set item Y as used and adjust
   its size to that of the request. Set item Z as being unused and
   set its size as the unused space left after Y.
```

Copyright © 2006 Stephen Fewer

When freeing a portion of memory previously allocated the appropriate heap item will be marked as unused and any adjacent unused items will be coalesced together to generate a single unused item.

4.4 Process Manager - `/src/kernel/pm/`

A process is a key concept in the AMOS kernel; it is defined by the `PROCESS_INFO` structure which contains the following information:

- **Name**
  This is the name of the process file name on secondary storage, e.g. "test.bin" or "kernel.elf".

- **ID**
  This is a unique ID number to identify the process. Many system calls such as `wake()`, `sleep()` and `kill()` use this process id number to identify the process they wish to perform the operation on. The id number is generated by incrementing the static integer variable `process_uniqueid`. While this guarantees each id number will be unique, the integer will overflow if we try to set a value greater then the maximum value possible for a 32bit integer. This value however is huge, over 2 billion.

- **State**
  This is the processes current state in the system and may be one of the following as specified in the design; CREATED, READY, RUNNING, BLOCKED or TERMINTED. The scheduler will use this value to determine if the process is eligible to be switched in.

- **Privilege**
  This is the processes privilege in the system and may be either USER or KERNEL. This value will be used by the paging layer to mark the processes pages with the correct privilege while also preventing lower privilege code accessing higher privilege code or data.

- **Heap**
  Defines the processes heap, specifying the base and top virtual address's of the heap. All calls to `mm_morecore()` will operate on this heap structure.

- **Handles**

  A table containing all the virtual file system handles opened and accessible by the process. The handles are of the `VFS_HANDLE` structure type (discussed in detail in the File System Section), which may be operated on through calls to the VFS, such as `vfs_read()` or `vfs_seek()`. In user mode though, handles are referenced via their integer offset in this table, e.g. handle 0 refers to the first `VFS_HANDLE` in the table, handle 1 to the second and so on. This gives a level of protection between processes so as they may not share or interfere with one another's file handles, e.g. handle number 88 in process A will refer to unique `VFS_HANDLE` in process A's handle table to that of process B, should process B also have a handle number 88. The processes default standard input and output is always handle 0 which is created upon spawning the process. The system calls for file access such as `open()`, `close()`, `read()`, and `write()` perform the handle table lookup and translation between `VFS_HANDLE`'s and their respective integer index. The handle table is also used to automatically close all remaining open file handles upon destroying the process.

- **Kernel Stack**

  The processes kernel stack is used when the process enters kernel mode from user mode, for example through a system call. Each process has a unique kernel stack allowing the kernel to be fully re-entrant (with critical sections such as access to global data protected by mutex's). When spawning the process, the initial state of the processes registers is setup in the kernel stack for the jump into user mode. This includes the location of the processes user mode stack.

- **Tick Slick**

  The tick slice represents the number of "ticks" the process has left before the scheduler selects another process to run. A tick is generated every time the timer interrupt, which is bound to the scheduler, fires. The duration between two consecutive ticks is a time slice.

- **Address Space**

The processes address space is defined by its page directory which is loaded into the CPU upon performing a context switch into the process. All operation in the paging layer such as mapping a virtual address to a physical one are performed on this address space.

The scheduler maintains a process table of all the process in the system as a forward linked list of `PROCESS_INFO` structures. Any access to the scheduler's process table is marked as a critical section and is protected with a mutex.

The kernel automatically becomes the first process in the system once the scheduler is initialized and will always have a process ID of zero. This enables the kernel to become the systems idle process and is always available for selection by the scheduler should no other process in the system be available to run. The kernel process could periodically run a deadlock detection routine as a future enhancement.

4.4.1 Process Operations - `/src/kernel/pm/process.c`

To aid controlling processes in the system we have a number of operations that may be performed on a process as outlined below. All these operations have a respective system call wrapper so as they may be performed from user mode.

- **Spawn**
  Spawning a new process occurs in the `process_spawn()` function. It involves reading in its executable image from a file on secondary storage and creating its process structure (as outlined above). Once the process has been created it is ready to be added into the system via the scheduler `scheduler_addProcess()` function, this places it into the schedulers process table allowing it to be selected for execution at a later point in time. A new process will enter the system in a CREATED state. A complication in spawning a new process is that you must copy the new processes code from one address space, that of the parent process, to that of the new process. The function `mm_pmemcpyto()` allows us to copy a portion of memory from a virtual address in the current address space to any

physical address, utilising the `paging_mapQuick()` function discussed previously. To copy the new process code into its own address space we use the following algorithm:

```
1. Read in a page of code into virtual page V1.
2. Allocate a page of physical memory P1 with
   physical_pageAlloc().
3. Use mm_pmemcpyto() to copy V1 to P1.
4. Map P1 into the new processes virtual address space with
   paging_map().
5. Repeat until all the code has been copied.
```

We must also map the kernel into the top gigabyte of the new processes address space so it will be able to perform operations in the kernel via system calls. We do this with the `paging_mapKernel()` function which maps the kernels page tables into the new processes page directory. A process of USER privilege will not be able to access the mapped in kernel without generating a page fault as the kernels page tables have a privilege of KERNEL, as shown in Screenshot 2 below.

```
AMOS 0.6.0
--------------------------------------------------------------------------------
AMOS:>spawn /amos/test.bin
spawn: Spawned process 8
Test App
    1. Exit Gracefully!
    2. General Protection Fault
    3. Page Fault
    4. Divide By Zero
    5. Stack Overflow
    6. Invalid Opcode
    7. Loop Forever
Please enter your choice: 3
About to memset the kernel heap...
[KERNEL] Page Fault at CS:EIP 0x1B:0x10000E41 Address 0xD0000000
[KERNEL]      CS:0x1B EIP:0x10000E41
[KERNEL]      DS:0x23 ES:0x23 FS:0x23 GS:0x23
[KERNEL]      EDI:0x0 ESI:0x0 EBP:0x20000F98 ESP:0xD003CFF0
[KERNEL]      EBX:0xD0000000 EDX:0xD0000000 ECX:0xD0001000 EAX:0x0
[KERNEL]      EFLAGS:0x207  SS0:0x23 ESP0:0x20000F90
AMOS:>_
```

Screenshot 2

Copyright © 2006 Stephen Fewer

- **Kill**

  To kill a process we use the process_kill() function or kill() system call. We set the process as being in a TERMINATED state, marking it as ineligible for selection by the scheduler. The scheduler, as described in the next section, will conduct the removal and destruction of the process. If the currently executing process is to be killed we must also force a context switch after setting the process as terminated. The scheduler will call the function process_destroy() to destroy a process. This involves closing all open handles in the processes handle table, destroying the processes address space which also frees all the physical memory associated with that address space (excluding shared kernel memory). Finally the processes kernel stack and the process structure itself are destroyed.

- **Wait**

  When a process P1 wishes to halt its flow of execution until another process P2 has terminated it must use the wait() system call, which is a wrapper for the process_wait() function. Here the calling process P1 will continue to yield itself allowing the scheduler to select another process to run, until the process P2 has terminated, whereby control in P1 will resume. This makes use of the process_yield() function to forcibly give up execution of the current time slice. We must yield execution as opposed to placing the process in a BLOCKED state as the current version of AMOS has no concept of signals, therefore we would be unable to signal this process to place it back in a READY state once the condition of P2 terminating has been met. There incurs a mild performance hit as the use of process_yield() in this fashion can allow the process P1 to be switched in to execution only to force itself to be switched out again.

- **Wake**

  If a process has been put into a BLOCKED state, the process_wake() function will place it back in a READY state, making it eligible for selection by the scheduler.

Copyright © 2006 Stephen Fewer

- **Sleep**

  A process may be put into a BLOCKED state with the `process_sleep()` function.

4.4.2 Scheduler - `/src/kernel/pm/scheduler.c`

The preemptive scheduler works by installing a handler on the timer interrupt, so when it fires the `scheduler_handler()` function is called. The handler will decide it a new process should be selected to run if either the current process has no more tick slices left or if it has been placed in any state other then RUNNING. The selection of a new process to run is performed in the `scheduler_select()` function. The low level interrupt routine `isr_common` in the source code file `src/kernel/int.asm` performs the actual context switch if a new process is selected.

The scheduler selects the next process to run in a round robin fashion using the following algorithm:

1. The currently executing process is process C.
2. Retrieve the next process P from the process table. If we have come to the end of the process table we begin our search from the start.
3. If P is TERMINATED we remove it from the process table and destroy the process before continuing the search from the beginning of the process table.
4. Else if P is BLOCKED we loop to 2 and continue the search with the next process in the table.
5. Else if P is READY or CREATED we stop the search and select it.
6. Otherwise if we have not selected a process we loop to 2 and continue the search with the next process in the table.
7. If process P is not process C then we set C as being READY before we set P as RUNNING and reset its tick slick to a value greater then 0.
8. Perform a context switch into process P.

Copyright © 2006 Stephen Fewer

4.4.3 Synchronization - /src/kernel/pm/sync/mutex.c

All global data in the kernel must be protected against concurrent access from multiple processes, or multiple threads of execution to be precise but AMOS currently does not support multithreaded processes. These critical sections are synchronised via mutual exclusion, or mutex's as known in AMOS. On multiprocessor systems spin locks can be used to achieve mutual exclusion but currently AMOS is only implemented for uniprocessor systems and achieves mutual exclusion by temporarily disabling interrupts.

A `MUTEX` structure defines a mutex which must be first initialized with a call to `mutex_init()`. When entering a critical section you call `mutex_lock()`, passing in the required mutex. Upon leaving a critical section you call `mutex_unlock()`.

4.5 File System - `/src/kernel/fs/vfs.c`

The VFS is composed of a series of mount points, which are file systems mounted on to the virtual file system tree. The VFS provides a thin layer to marshal file system calls to the underlying file system drivers via the VFS mount points. A file system is defined by the `VFS_FILESYSTEM` structure. The main component of this structure is the `VFS_FILESYSTEM_CALLTABLE` structure which is a virtual function table that a file system driver will fill in with its corresponding calls, such as open and read. So when a call to `vfs_open()` occurs the correct mount point is retrieved based on the requested files path, we then pass on the open call to the file system driver associated with the mount point, e.g. `fat_open()` or `dfs_open()`. A file system driver must register itself with the VFS with a call to `vfs_register()` before we can mount a volume of that file system type. The basic structure of a file system driver in code is shown in Listing 1 below.

```
struct VFS_HANDLE * fat_open( struct VFS_HANDLE * handle, char * filename )
{
  // ...
}

int fat_close( struct VFS_HANDLE * handle )
{
  // ...
}

int fat_init( void )
{
  struct VFS_FILESYSTEM * fs;
  // create the file system structure
  fs = (struct VFS_FILESYSTEM *)mm_kmalloc( sizeof(struct VFS_FILESYSTEM) );
  // set the file system type
  fs->fstype = FAT_TYPE;
  // setup the file system calltable
  fs->calltable.open     = fat_open;
  fs->calltable.close    = fat_close;
  // ...
  // register the file system with the VFS
  return vfs_register( fs );
}
```

Listing 1

The subsystem is initialized with a call to `vfs_init()` where the DFS is initialized first and mounted to the location of `/amos/device/`, followed by the FAT file system driver.

A `VFS_HANDLE` structure defines a kernel handle to a file in the VFS. It stores the mount point that the file resides on along with a custom data pointer which the files file system driver can set to record specific information, such as the files FAT information if the file is on a FAT file system or the files device driver data if the file is on a device file system. Operations such as `vfs_read()` or `vfs_write()` operate on the file associated with a `VFS_HANDLE`.

The VFS supports opening a file in the following modes, some of which may be combined together:

- `VFS_MODE_READ`
  `Allow read access to the file but not write access.`
- `VFS_MODE_WRITE`
  `Allow write access to the file but not read access.`
- `VFS_MODE_READWRITE`
  `Allow both read and write access to the file.`
- `VFS_MODE_CREATE`
  `Create the file if it does not exist.`
- `VFS_MODE_TRUNCATE`
  `Upon opening the file, set its length to zero.`
- `VFS_MODE_APPEND`
  `Set the file position to the end of the file after each write.`

A function `vfs_clone()` was required to clone an already opened file handle, used when spawning a new process that wished to have the same standard input output console as the parent process. The cloned handles are identical upon the call to `vfs_clone()` returning but subsequent operations on each handle may be unique depending on the underlying implementation of clone by the relevant file system driver or device driver.

4.5.1 Device File System - `/src/kernel/fs/dfs.c`

To DFS is a thin layer that lets us access devices in the IO subsystem via the VFS, allowing us to treat all devices as files in the system.

By providing a file system interface to the VFS via exporting a `VFS_FILESYSTEM` structure and appropriate call table we can pass on file systems calls such as open, close, read and write to the IO subsystem, discussed in the section 4.6.

4.5.2 FAT File System - `/src/kernel/fs/fat.c`

A FAT file system is composed of a boot sector followed by the FAT, followed by the root directory and then the data area where file and directory data is stored. FAT operates on data segments called clusters which are a series of sectors on secondary storage. The FAT itself is a table of cluster entries and is used to compose a logical cluster chain as shown in Figure 6 below.

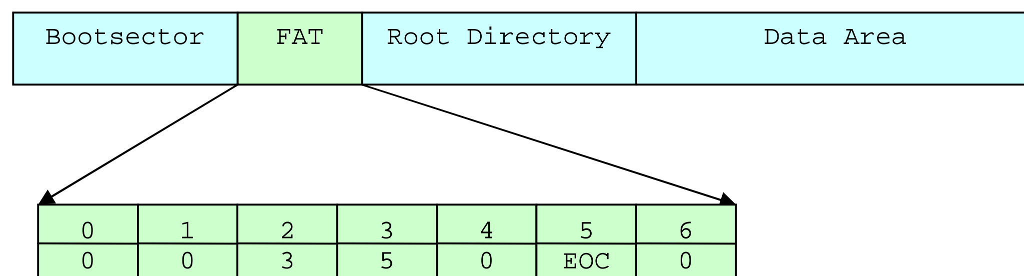

Figure 6

Here we see a logical cluster chain of cluster 2 → 3 → 5 where it is terminated with an End of Cluster (EOC) value. A files entry in a directory will specify its starting cluster, we can then lookup this cluster index in the FAT and proceed to traverse the files cluster chain.

We must provide a number of utility functions to help manipulate the FAT and clusters. We have three functions to work directly with the FAT:

- `fat_setFATCluster()`
  This function lets us set a specific entry in the FAT to a specified value, and also allows us to optionally commit the FAT, which we cache in memory, back to secondary storage. If we are going to perform a series of `fat_setFATCluster()` operation we only need to commit the FAT at the end which is why it is an optional feature.

- `fat_getFATCluster()`
  This function lets us retrieve an entry in the FAT, and is used to traverse cluster chains.

- `fat_getFreeCluster()`
  This function will return the number of a free cluster in the FAT volume by searching the FAT for an entry marked as free. We use this function when expanding a files size and we need to allocate more data clusters.

When we open a file we need to locate the files entry which is a `FAT_ENTRY` structure. This structure is for both files and directory's and stores attributes such the name, the size and starting cluster of the entry. A directory is a special file that is composed of a series of `FAT_ENTRY` structures, one for each file or sub directory in that directory. The function `fat_file2entry()` will find a files `FAT_ENTRY` based on a file path. Once found we store it in a `FAT_FILE` structure which also records the current file position for read and write operations along with the FAT mount that the file resides on.

As we can mount multiple FAT volumes with the same FAT driver we need a structure to store the volume specific information such as the device in the VFS that the volume is located on, e.g. `/amos/device/floppy1`. We also cache the volumes FAT and the root directory. The `FAT_MOUNTPOINT` structure is used for this purpose.

To read from a file we use the following algorithm, as implemented in the `fat_rw()` function.

1. Traverse the files cluster chain to retrieve the correct cluster to begin reading from.
2. If we are reading from a point inside a cluster, calculate the correct offset into that cluster.
3. Calculate the amount of bytes to read in this iteration of the loop. If we are attempting to read across two clusters we reduce the bytes to read in order to only read from the first cluster.
4. Read in the cluster from the FAT volume and copy the data over to the user's buffer.
5. Retrieve the next cluster in the chain.
6. If we still have bytes to read, loop to 2.
7. Update the files current position by the amount of bytes read.

To `fat_rw()` function can also perform write operations as the algorithm is nearly identical. Instead of reading in the cluster data in point 4 we write to the current cluster, retrieving the data to write from the user's buffer.

To create a new file we must find the directory entry that we wish to create the new file in. The directory will be a file itself containing a series of `FAT_ENTRY`'s as mentioned previously. We create a new entry corresponding to the file we wish to create in the directory. If we are creating a new directory as opposed to a file the operation is the same except we must also allocate a new cluster for the new directory and fill it full of empty file entries.

To delete a file we simply set its corresponding `FAT_ENTRY` in the volume as deleted.

To rename a file we simply change the name attribute of the files corresponding `FAT_ENTRY` in the volume.

Copyright © 2006 Stephen Fewer

4.6 Input Output - `/src/kernel/io/io.c`

The IO subsystem is composed of the IO controller layer and the IO device drivers of which we have four by default. They are for the bit bucket, floppy drive, virtual consoles and keyboard. When the subsystem is initialized in `io_init()` we initialize these four device drivers aswell. The IO controller provides two functions `io_add()` and `io_remove()` to add and remove devices in the system. These functions communicate with the DFS to add and remove a device from the VFS. A device driver will call `io_add()` upon initializing to add itself into the system. The IO controller will forward calls to the correct device driver from the DFS. There are 7 calls supported, namely: open, close, read, write, clone, seek and control.

It is in the IO controller that we could provide for buffering and caching of data which is why a driver will register itself as either a character or block device.

4.6.1 Bit Bucket - `/src/kernel/io/dev/bitbucket.c`

The bit bucket is a very simple device driver and a good example of the AMOS device driver framework. It is analogous with the `/dev/null` device in UNIX or the `NUL:` device in DOS or the `NIL:` device in Amiga OS. The entire code for the device driver is as shown below in Listing 2.

```
struct IO_HANDLE * bitbucket_open(struct IO_HANDLE * handle,char * filename)
{
  // return the handle
  return handle;
}

int bitbucket_close( struct IO_HANDLE * handle )
{
  // return success
  return SUCCESS;
}

int bitbucket_write( struct IO_HANDLE * handle, BYTE * buffer, DWORD size  )
{
  // return the number of bytes we were asked to write
  return size;
}

int bitbucket_init( void )
{
  struct IO_CALLTABLE * calltable;
  // setup the calltable for this driver
  calltable=(struct IO_CALLTABLE *)mm_kmalloc(sizeof(struct IO_CALLTABLE));
  calltable->open    = bitbucket_open;
  calltable->close   = bitbucket_close;
  calltable->clone   = NULL;
  calltable->read    = NULL;
  calltable->write   = bitbucket_write;
  calltable->seek    = NULL;
  calltable->control = NULL;
  // add the bitbucket device
  io_add( "bitbucket", calltable, IO_CHAR );
  return SUCCESS;
}
```

Listing 2

Here we see the structure of a device driver is much like that of a file system driver. Upon initialization we construct a call table and fill in the values for the 7 calls possible. There are two mandatory calls that all device drivers must support, namely open and close, the others are optional, as depending on the device it may make no sense to perform that operation, e.g. writing to the keyboard. Once the call table is created we register the device driver via a call to `io_add()` passing in the call table, the name we wish to give the device and whether it is a character of block device. By

adding in the "`bitbucket`" device we may now access it via the VFS through the path name of "`/amos/device/bitbucket`", as the DFS is mounted to the location of "`/amos/device/`" as described previously in section 4.5.

4.6.2 Keyboard - `/src/kernel/io/dev/keyboard.c`

The keyboard device driver will install an interrupt handler on the keyboard interrupt so upon every key press (or release) we will generate a call the `keyboard_handler()` function. Instead of exporting a read function the keyboard driver uses a feature in the virtual console driver to send characters types to the currently active virtual console by writing the character to the `console0` device.

A key press will generate a scan code which represents the key that was pushed, we lookup the ASCII character that the scan code represents via a keyboard map. A keyboard map is represented by both an upper map and a lower map, for example character "5" on the lower map would be character "%" on the upper map. We can detect if either the shift key or the caps lock key is active and use the upper map; otherwise we use the lower map. The current version of AMOS uses a standard 101 key UK keyboard map. A future enhancement would be to load a custom keyboard map at boot time. This action could be performed in the user mode init process.

A number of keys have a special purpose in AMOS. The F1 through to F4 keys when pressed will activate a virtual console, e.g. pressing F2 will activate `console2`. We can do this by using the `vfs_control()` function operating on the `console0` device. We pass in a command specific to the virtual console driver informing it to set another console as active. The F5 key has been chosen to display some system information such as the state of the scheduler's process table as shown in Screenshot 3 below.

Copyright © 2006 Stephen Fewer

```
AMOS 0.6.0
----------------------------------------------------------------
Shell Version 1.0
AMOS:>
[KERNEL] Scheduler Process Table:
[KERNEL]      9 test.bin (User) is Ready, ticks: 0
[KERNEL]      8 hanoi.bin (User) is Running, ticks: 1
[KERNEL]      7 hanoi.bin (User) is Ready, ticks: 0
[KERNEL]      6 hanoi.bin (User) is Ready, ticks: 0
[KERNEL]      5 shell.bin (User) is Ready, ticks: 0
[KERNEL]      4 shell.bin (User) is Ready, ticks: 0
[KERNEL]      3 shell.bin (User) is Ready, ticks: 0
[KERNEL]      2 shell.bin (User) is Ready, ticks: 0
[KERNEL]      0 kernel.elf (Kernel) is Ready, ticks: 0

AMOS:>_
```

Screenshot 3

4.6.3 Virtual Console - /src/kernel/io/dev/console.c

The virtual console device driver creates four virtual console devices in the system (console1 through to console4) along with a fifth console, console0, which is always the currently active console in the system. The function console_create() is used to create the four unique virtual consoles. A virtual console is represented by the CONSOLE structure which is composed of a number of parts as shown below in Figure 7.

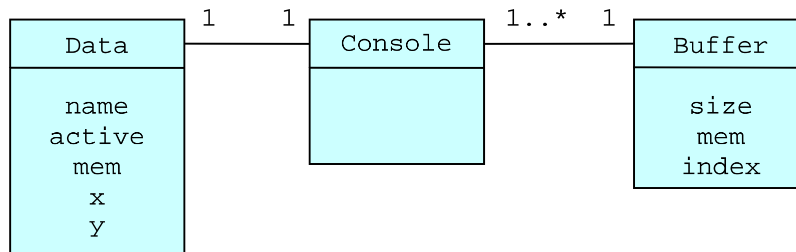

Figure 7

A console is associated with both a data structure called CONSOLE_DATA aswell as one or more buffer structures called CONSOLE_BUFFER. The data structure contains

the name of the console it represents, the memory area where the console data is stored along with some metadata such as the current cursor location and whether this is the active console. A new buffer structure is created every time a handle to the console is obtained, this allows for unique read operations to be performed by multiple handles on the same virtual console, for example if there are three handles opened on `console1`, handle 1 might wish to read exactly 128 bytes, while handle 2 might wish to only read up until a new line and handle 3 might not wish to read any bytes yet. By providing multiple buffers we can accommodate this. The function `console_putchBuffer()` handles placing bytes being typed in on a console into any buffers that are awaiting input. The function `console_read()` initializes the buffer to handle the new read operation, including specifying how many bytes to read and the memory buffer to place them in. It is a blocking function and while input is waiting to be received we use `process_yield()` to give up the current time slice to another process. We cannot use `process_sleep()` as discussed in section 4.4.1 as the current version of AMOS does not support signal handling.

Cloning a console handle in `console_clone()` involves creating a new `CONSOLE` structure and associating it with the `CONSOLE_DATA` structure of the console to be cloned while creating a new `CONSOLE_BUFFER` structure unique to the newly cloned console handle.

When writing bytes to a virtual console in `console_putChar()` we test to see If the we are operating on the currently active virtual console and if so we copy the consoles data area over to the VGA memory mapped area in system so as to place it on the computers monitor. This must also be done when setting a virtual console as active in `console_activate()`.

The `console_control()` as mentioned briefly in the keyboard section above, allows us to pass in commands specific to the virtual console driver. These commands are:

CONSOLE_SETECHO

Sending this command to a console lets us specify whether we want characters typed on the keyboard to be echoed onto the consoles screen. Typically only the currently active console would want to do this.

- **CONSOLE_SETACTIVE**

Sending this command lets us activate a virtual console.

- **CONSOLE_SENDCHAR**

We can input bytes to a console with this command.

- **CONSOLE_SETBREAK**

This command lets us specify a break byte for a consoles buffer. When this byte is received as input by the console any blocked read operation on the buffer will be allowed to complete. This is useful if we only wish to read a buffer of data up to a new line as we can specify the new line character '`\n`' as the break byte.

4.6.4 Floppy Drive - `/src/kernel/io/dev/floppy.c`

The floppy driver will create a device for each floppy drive present in the system, e.g. `floppy1` and `floppy2`. The floppy driver has complete read and write support and follows the specifications laid out in the Intel 82077AA CHMOS Single-Chip Floppy Disk Controller which are 100% PC AT and PS/2 Compatible.

Copyright © 2006 Stephen Fewer

4.7 Portability

The Kernel may be ported to architectures other then x86. The requirements of the new architecture would be that it had a memory management unit (MMU) that could support virtual memory. The key layers that would need to be ported would be as follows:

- **The Interrupt Layer**

The interrupt layer is also responsible for the software task switching as discussed in section 4.2.1 and would need to be ported in a way that supports this. The architecture specific portions of the process layer would also need to be ported, primarily the state of the processes kernel stack for its initial context switch.

- **The Paging Layer**

The paging layer would need to be rewritten for the target architecture, exporting the mapping functions for the other layers to use.

- **The Device Drivers**

The default device drivers included with AMOS would need to be rewritten for the target architecture.

- **The Kernel Loader**

The kernel loader stub which bootstraps the kernel proper would need to be ported to the target architecture.

The tool chain would also need to be available on the target architecture in order to build the system from its source code. As the GCC is the main component of the tool chain this is no problem as GCC is available for many different architectures already.

Copyright © 2006 Stephen Fewer