File: ./init/main.c

```c
#include <physical.h>
#include <virtual.h>
#include <heap.h>
#include <cpu.h>
#include <log.h>
#include <debug.h>
#include <assert.h>
#include <timer.h>
#include <irql.h>
#include <thread.h>
#include <panic.h>
#include <stdlib.h>
#include <process.h>
#include <progload.h>
#include <dev.h>
#include <vfs.h>
#include <diskcache.h>
#include <transfer.h>
#include <fcntl.h>
#include <console.h>
#include <swapfile.h>
#include <diskutil.h>
#include <string.h>
#include <filesystem.h>
#include <driver.h>

/*
 * Next steps:
 * - program loader / dynamic linker
 * - system call interface (KRNLAPI.LIB)
 * - C standard library
 * - complete-enough CLI OS
 *          terminal that supports pipes, redirection and background processes
 *          cd, ls/dir, type, mkdir, rm, more, rename, copy, tree, mkfifo, pause, rmtree, rmdir, cls, copytree, link,
 *              ...ttyname, sleep, exit
 *          port zlib, nasm
 * - floppy driver
 * - FAT32 driver
 * - floating point support (init and task switching)
 * - disk caching
 * - shutdown needs to close the entire VFS tree (e.g. so buffers can be flushed, etc).
 * - recycling vnodes if opening same file more than once
 * - initrd and boot system
 * - more syscalls
 * - VnodeOpWait, select/poll syscalls
 */

void DummyAppThread(void*) {
PutsConsole("drv0:/> ");

    struct open_file* con;
    OpenFile("con:", O_RDONLY, 0, &con);

    while (true) {
        char bf[302];
 inline_memset(bf, 0, 302);
        struct transfer tr = CreateKernelTransfer(bf, 301, 0, TRANSFER_READ);
 ReadFile(con, &tr);
 PutsConsole("Command not found: ");
 PutsConsole(bf);
        PutsConsole("\n");

        if (bf[0] == 'u' || bf[0] == 'U') {
            CreateUsermodeProcess(NULL, "sys:/init.exe");

        } else if (bf[0] == 'p' || bf[0] == 'P') {
            Panic(PANIC_MANUALLY_INITIATED);

        } else if (bf[0] == 'e' || bf[0] == 'E') {
            MapVirt(0, 0, 8 * 4096, VM_LOCK | VM_READ, NULL, 0);
        }

 PutsConsole("drv0:/> ");
    }
}

void InitUserspace(void) {
    size_t free = GetFreePhysKilobytes();
    size_t total = GetTotalPhysKilobytes();
    DbgScreenPrintf("NOS Kernel\nCopyright Alex Boxall 2022-2023\n\n%d / %d KB used (%d%% free)\n\n", total - free, total, 100 * (free) / total);
    CreateThread(DummyAppThread, NULL, GetVas(), "dummy app");
}

void InitThread(void*) {
    ReinitHeap();
    InitRandomDevice();
    InitNullDevice();
    InitConsole();
    InitProcess();
    InitDiskCaches();

    InitFilesystemTable();
    ArchInitDev(false);

    struct open_file* sys_folder;
    int res = OpenFile("drv0:/System", O_RDONLY, 0, &sys_folder);
    if (res != 0) {
        PanicEx(PANIC_NO_FILESYSTEM, "sys A");
    }
    res = AddVfsMount(sys_folder->node, "sys");
    if (res != 0) {
        PanicEx(PANIC_NO_FILESYSTEM, "sys B");
    }

    struct open_file* swapfile;
    res = OpenFile("raw-hd0:/part1", O_RDWR, 0, &swapfile);
    if (res != 0) {
        PanicEx(PANIC_NO_FILESYSTEM, "swapfile A");
    }
    res = AddVfsMount(swapfile->node, "swap");
    if (res != 0) {
        PanicEx(PANIC_NO_FILESYSTEM, "swapfile B");
    }

    InitSwapfile();
    InitSymbolTable();
    ArchInitDev(true);
    InitProgramLoader();
    InitUserspace();

    MarkTfwStartPoint(TFW_SP_ALL_CLEAR);

    while (true) {
        /*
         * We crash in strange and rare conditions if this thread's stack gets removed, so we will
         * ensure we don't terminate it.
         */
        SleepMilli(100000);
```

```
    }
}
#include <machine/portio.h>
static void InitSerialDebugging(void) {
    const int PORT = 0x3F8;
    outb(PORT + 1, 0x00);    // Disable all interrupts
    outb(PORT + 3, 0x80);    // Enable DLAB (set baud rate divisor)
    outb(PORT + 0, 0x03);    // Set divisor to 3 (lo byte) 38400 baud
    outb(PORT + 1, 0x00);    //                  (hi byte)
    outb(PORT + 3, 0x03);    // 8 bits, no parity, one stop bit
    outb(PORT + 2, 0xC7);    // Enable FIFO, clear them, with 14-byte threshold
    outb(PORT + 4, 0x0B);    // IRQs enabled, RTS/DSR set
    outb(PORT + 4, 0x1E);    // Set in loopback mode, test the serial chip
    outb(PORT + 0, 0xAE);    // Test serial chip (send byte 0xAE and check if serial returns same byte)

    // Check if serial is faulty (i.e: not same byte as sent)
    if(inb(PORT + 0) != 0xAE) {
        return;
    }

    // If serial is not faulty set it in normal operation mode
    // (not-loopback with IRQs enabled and OUT#1 and OUT#2 bits enabled)
    outb(PORT + 4, 0x0F);
}

void KernelMain(void) {
    InitSerialDebugging();

    LogWriteSerial("KernelMain: kernel is initialising...\n");

    /*
     * Allows us to call GetCpu(), which allows IRQL code to work. Anything which uses
     * IRQL (i.e. the whole system) relies on this, so this must be done first.
     */
    InitCpuTable();
    assert(GetIrql() == IRQL_STANDARD);

    /*
     * Initialise the testing framework if we're in debug mode.
     */
    InitTfw();
    MarkTfwStartPoint(TFW_SP_INITIAL);

    InitPhys();
    MarkTfwStartPoint(TFW_SP_AFTER_PHYS);

    /*
     * Allows deferments of functions to actually happen. IRQL is still usable beforehand though.
     */
    InitIrql();
    InitVfs();
    InitTimer();
    InitScheduler();
    InitDiskUtil();

    InitHeap();
    MarkTfwStartPoint(TFW_SP_AFTER_HEAP);

    InitBootstrapCpu();
    MarkTfwStartPoint(TFW_SP_AFTER_BOOTSTRAP_CPU);

    InitVirt();
    MarkTfwStartPoint(TFW_SP_AFTER_VIRT);

    ReinitPhys();
    MarkTfwStartPoint(TFW_SP_AFTER_PHYS_REINIT);

    InitOtherCpu();
    MarkTfwStartPoint(TFW_SP_AFTER_ALL_CPU);


    CreateThreadEx(InitThread, NULL, GetVas(), "init", NULL, SCHEDULE_POLICY_FIXED, FIXED_PRIORITY_KERNEL_NORMAL, 0);
    StartMultitasking();
}
```

File: ./adt/threadlist.c

```c
#include <common.h>
#include <threadlist.h>
#include <heap.h>
#include <assert.h>
#include <thread.h>
#include <panic.h>
#include <log.h>
#include <string.h>

void ThreadListInit(struct thread_list* list, int index) {
    inline_memset(list, 0, sizeof(struct thread_list));
    list->index = index;
}

void ThreadListInsert(struct thread_list* list, struct thread* thread) {
#ifndef NDEBUG
    if (ThreadListContains(list, thread)) {
        assert(!ThreadListContains(list, thread));
    }
#endif

    if (list->tail == NULL) {
        assert(list->head == NULL);
        list->head = thread;

    } else {
        list->tail->next[list->index] = thread;
    }

    list->tail = thread;
    thread->next[list->index] = NULL;
}

static int ThreadListGetIndex(struct thread_list* list, struct thread* thread) {
    struct thread* iter = list->head;
    int i = 0;
    while (iter != NULL) {
        if (iter == thread) {
            return i;
        }
        ++i;
        iter = iter->next[list->index];
    }
    return -1;
}

bool ThreadListContains(struct thread_list* list, struct thread* thread) {
    return ThreadListGetIndex(list, thread) != -1;
}

static void ProperDelete(struct thread_list* list, struct thread* iter, struct thread* prev) {
    if (iter == list->head) {
        list->head = list->head->next[list->index];
    } else {
        prev->next[list->index] = iter->next[list->index];
    }

    if (iter == list->tail) {
        list->tail = prev;
    }

    if (list->head == NULL) {
        assert(list->tail == NULL);
    }
}

static void ThreadListDeleteIndex(struct thread_list* list, int index) {
    struct thread* iter = list->head;
    struct thread* prev = NULL;

    assert(index >= 0);

    int i = 0;
    while (iter != NULL) {
        if (i == index) {
            ProperDelete(list, iter, prev);
            return;
        }
        ++i;
        prev = iter;
        iter = iter->next[list->index];
    }

    Panic(PANIC_THREAD_LIST);
}

struct thread* ThreadListDeleteTop(struct thread_list* list) {
    struct thread* top = list->head;
    if (list->head == list->tail) {
        list->tail = NULL;
    }
    list->head = list->head->next[list->index];

    if (list->head == NULL) {
        assert(list->tail == NULL);
    }

    return top;
}
void ThreadListDelete(struct thread_list* list, struct thread* thread) {
    ThreadListDeleteIndex(list, ThreadListGetIndex(list, thread));
}
```

File: ./adt/avl.c

```c
#include <common.h>
#include <assert.h>
#include <heap.h>
#include <avl.h>
#include <log.h>

struct avl_node {
    struct avl_node* left;
    struct avl_node* right;
    void* data;
};

struct avl_tree {
 int size;
 struct avl_node* root;
    avl_deletion_handler deletion_handler;
    avl_comparator equality_handler;
};

static struct avl_node* AvlCreateNode(void* data, struct avl_node* restrict left, struct avl_node* restrict right) {
```

```
    struct avl_node* tree = AllocHeap(sizeof(struct avl_node));
    tree->left = left;
    tree->right = right;
    tree->data = data;
    return tree;
}

static int AvlGetHeight(struct avl_node* tree) {
    if (tree == NULL) {
        return 0;
    }

    return 1 + MAX(AvlGetHeight(tree->left), AvlGetHeight(tree->right));
}

static int AvlGetBalance(struct avl_node* tree) {
    if (tree == NULL) {
        return 0;
    }

    return AvlGetHeight(tree->left) - AvlGetHeight(tree->right);
}

static struct avl_node* AvlRotateLeft(struct avl_node* tree) {
    struct avl_node* new_root = tree->right;
    struct avl_node* new_right = new_root->left;
    new_root->left = tree;
    tree->right = new_right;
    return new_root;
}

static struct avl_node* AvlRotateRight(struct avl_node* tree) {
    struct avl_node* new_root = tree->left;
    struct avl_node* new_left = new_root->right;
    new_root->right = tree;
    tree->left = new_left;
    return new_root;
}

static struct avl_node* AvlBalance(struct avl_node* tree) {
    if (tree == NULL) {
        return NULL;
    }

    int bf = AvlGetBalance(tree);
    assert(bf >= -2 && bf <= 2);

    if (bf == -2) {
        if (AvlGetBalance(tree->right) == 1) {
            tree->right = AvlRotateRight(tree->right);
        }
        return AvlRotateLeft(tree);

    } else if (bf == 2) {
        if (AvlGetBalance(tree->left) == -1) {
            tree->left = AvlRotateLeft(tree->left);
        }
        return AvlRotateRight(tree);

    } else {
        return tree;
    }
}

static struct avl_node* AvlInsert(struct avl_node* tree, void* data, avl_comparator comparator) {
    struct avl_node* new_tree;

    assert(comparator != NULL);
    assert(tree != NULL);

    if (comparator(data, tree->data) < 0) {
        struct avl_node* left_tree;
        if (tree->left == NULL) {
            left_tree = AvlCreateNode(data, NULL, NULL);
        } else {
            left_tree = AvlInsert(tree->left, data, comparator);
        }

        new_tree = AvlCreateNode(tree->data, left_tree, tree->right);

    } else {
        struct avl_node* right_tree;
        if (tree->right == NULL) {
            right_tree = AvlCreateNode(data, NULL, NULL);
        } else {
            right_tree = AvlInsert(tree->right, data, comparator);
        }

        new_tree = AvlCreateNode(tree->data, tree->left, right_tree);
    }

    FreeHeap(tree);

    return AvlBalance(new_tree);
}

static struct avl_node* AvlDelete(struct avl_node* tree, void* data, avl_comparator comparator) {
    if (tree == NULL) {
        return NULL;
    }

    struct avl_node* to_free = NULL;

    if (comparator(data, tree->data) < 0) {
        tree->left = AvlDelete(tree->left, data, comparator);

    } else if (comparator(data, tree->data) > 0) {
        tree->right = AvlDelete(tree->right, data, comparator);

    } else if (tree->left == NULL) {
        to_free = tree;
        tree = tree->right;

    } else if (tree->right == NULL) {
        to_free = tree;
        tree = tree->left;

    } else {
        struct avl_node* node = tree->right;
        while (node->left != NULL) {
            node = node->left;
        }

        tree->data = node->data;
        tree->right = AvlDelete(tree->right, node->data, comparator);
    }

    /*
     * If NULL is passed in to FreeHeap, nothing happens (which is want we want).
     */
```

```
    FreeHeap(to_free);

    return AvlBalance(tree);
}

/**
 * Given an object, find it in the AVL tree and return it. This is useful if the comparator only compares
 * part of the object, and so the entire object can be retrieved by searching for only part of it.
 */
static void* AvlGet(struct avl_node* tree, void* data, avl_comparator comparator) {
    if (tree == NULL) {
        return NULL;
    }

    if (comparator(tree->data, data) == 0) {
        /*
         * Must return `tree->data`, (and not `data`), as tree->data != data if there is a custom comparator.
         */
        return tree->data;
    }

    void* left = AvlGet(tree->left, data, comparator);
    if (left != NULL) {
        return left;
    }
    return AvlGet(tree->right, data, comparator);
}

static void AvlPrint(struct avl_node* tree, void(*printer)(void*)) {
    if (tree == NULL) {
        return;
    }
    AvlPrint(tree->left, printer);
    if (printer == NULL) {
        LogWriteSerial("[[0x%X]], \n", tree->data);
    } else {
        printer(tree->data);
    }
    AvlPrint(tree->right, printer);
}

static bool AvlContains(struct avl_node* tree, void* data, avl_comparator comparator) {
    if (tree == NULL) {
        return false;
    }

    if (comparator(tree->data, data) == 0) {
        return true;
    }

    return AvlContains(tree->left, data, comparator) || AvlContains(tree->right, data, comparator);
}

static void AvlDestroy(struct avl_node* tree, avl_deletion_handler handler) {
    if (tree == NULL) {
        return;
    }

    AvlDestroy(tree->left, handler);
    AvlDestroy(tree->right, handler);
    if (handler != NULL) {
        handler(tree->data);
    }
    FreeHeap(tree);
}

static int AvlDefaultComparator(void* a, void* b) {
    if (a == b) return 0;
    return (a < b) ? -1 : 1;
}

struct avl_tree* AvlTreeCreate(void) {
    struct avl_tree* tree = AllocHeap(sizeof(struct avl_tree));
    tree->size = 0;
    tree->root = NULL;
    tree->deletion_handler = NULL;
    tree->equality_handler = AvlDefaultComparator;
    return tree;
}

avl_deletion_handler AvlTreeSetDeletionHandler(struct avl_tree* tree, avl_deletion_handler handler) {
    avl_deletion_handler ret = tree->deletion_handler;
    tree->deletion_handler = handler;
    return ret;
}

avl_comparator AvlTreeSetComparator(struct avl_tree* tree, avl_comparator handler) {
    avl_comparator ret = tree->equality_handler;
    tree->equality_handler = handler;
    return ret;
}

void AvlTreeInsert(struct avl_tree* tree, void* data) {
    if (tree->root == NULL) {
        tree->root = AvlCreateNode(data, NULL, NULL);
    } else {
        tree->root = AvlInsert(tree->root, data, tree->equality_handler);
    }
    tree->size++;
}

void AvlTreeDelete(struct avl_tree* tree, void* data) {
    tree->root = AvlDelete(tree->root, data, tree->equality_handler);
    tree->size--;
}

bool AvlTreeContains(struct avl_tree* tree, void* data) {
    return AvlContains(tree->root, data, tree->equality_handler);
}

void* AvlTreeGet(struct avl_tree* tree, void* data) {
    return AvlGet(tree->root, data, tree->equality_handler);
}

int AvlTreeSize(struct avl_tree* tree) {
    return tree->size;
}

void AvlTreeDestroy(struct avl_tree* tree) {
    AvlDestroy(tree->root, tree->deletion_handler);
    FreeHeap(tree);
}

struct avl_node* AvlTreeGetRootNode(struct avl_tree* tree) {
    return tree->root;
}

struct avl_node* AvlTreeGetLeft(struct avl_node* node) {
    if (node == NULL) {
        return NULL;
```

```
    }

    return node->left;
}

struct avl_node* AvlTreeGetRight(struct avl_node* node) {
    if (node == NULL) {
        return NULL;
    }

    return node->right;
}

void* AvlTreeGetData(struct avl_node* node) {
    if (node == NULL) {
        return NULL;
    }

    return node->data;
}

void AvlTreePrint(struct avl_tree* tree, void(*printer)(void*)) {
    AvlPrint(tree->root, printer);
}
```

File: ./adt/priorityqueue.c

```

#include <heap.h>
#include <string.h>
#include <log.h>
#include <panic.h>
#include <assert.h>
#include <priorityqueue.h>

/*
 * Implements the max-heap and min-heap data structures. To avoid confusion with the
 * heap memory manager, it is referred to as a priority queue.
 */

struct priority_queue {
    int capacity;
    int size;
    int element_width;
    int qwords_per_element;       // includes the + 1 for the priority
    bool max;
    uint64_t* array;              // length is: capacity * qwords_per_element
};

struct priority_queue* PriorityQueueCreate(int capacity, bool max, int element_width) {
    assert(capacity > 0);
    assert(element_width > 0);

    struct priority_queue* queue = AllocHeap(sizeof(struct priority_queue));
    queue->capacity = capacity;
    queue->size = 0;
    queue->element_width = element_width;
    queue->qwords_per_element = 1 + (element_width + sizeof(uint64_t) - 1) / sizeof(uint64_t);
    queue->max = max;
    queue->array = AllocHeap(sizeof(uint64_t) * queue->qwords_per_element * capacity);
    return queue;
}

void PriorityQueueDestroy(struct priority_queue* queue) {
    FreeHeap(queue->array);
    FreeHeap(queue);
}

static void SwapElements(struct priority_queue* queue, int a, int b) {
    a *= queue->qwords_per_element;
    b *= queue->qwords_per_element;

    for (int i = 0; i < queue->qwords_per_element; ++i) {
        uint64_t tmp = queue->array[a];
        queue->array[a] = queue->array[b];
        queue->array[b] = tmp;
        ++a; ++b;
    }
}

static void Heapify(struct priority_queue* queue, int i) {
    int extreme = i;
    int left = i * 2 + 1;
    int right = left + 1;

    if (left < queue->size) {
        if ((queue->max && queue->array[left * queue->qwords_per_element] > queue->array[extreme * queue->qwords_per_element]) || (!queue->max && queue->array[le
            extreme = left;
        }
    }
    if (right < queue->size) {
        if ((queue->max && queue->array[right * queue->qwords_per_element] > queue->array[extreme * queue->qwords_per_element]) || (!queue->max && queue->array[r
            extreme = right;
        }
    }
    if (i != extreme) {
        SwapElements(queue, i, extreme);
        Heapify(queue, extreme);
    }
}

void PriorityQueueInsert(struct priority_queue* queue, void* elem, uint64_t priority) {
    if (queue->size == queue->capacity) {
        // I think this can happen when the OS is running too slowly! (the deferred function buffer actually fills up
        // and overflows). Testing on real H/W from 1996, debug prints took about a second per character!!
        PanicEx(PANIC_PRIORITY_QUEUE, "insert called when full");
    }

    int i = queue->size++;
    queue->array[i * queue->qwords_per_element] = priority;
    inline_memcpy(queue->array + i * queue->qwords_per_element + 1, elem, queue->element_width);

    if (queue->max) {
        while (i != 0 && queue->array[((i - 1) / 2) * queue->qwords_per_element] < queue->array[i * queue->qwords_per_element]) {
            SwapElements(queue, (i - 1) / 2, i);
            i = (i - 1) / 2;
        }
    } else {
        while (i != 0 && queue->array[((i - 1) / 2) * queue->qwords_per_element] > queue->array[i * queue->qwords_per_element]) {
            SwapElements(queue, (i - 1) / 2, i);
            i = (i - 1) / 2;
        }
    }
}

/*
```

```
 * returns the priority, and a REFERENCE to the data IN THE PRIORITY QUEUE. It is NOT a copy!
 * This is why Pop() can't return it, because popping it overwrites the data!
 */
struct priority_queue_result PriorityQueuePeek(struct priority_queue* queue) {
    if (queue->size == 0) {
        PanicEx(PANIC_PRIORITY_QUEUE, "peek called on empty");
    }

    struct priority_queue_result retv;
    retv.priority = queue->array[0];
    retv.data = (void*) (queue->array + 1);
    return retv;
}

/*
 * doesn't return the value, as the value would have been erased in the pop (PriorityQueue doesn't
 * allocate memory, as it needs to be used in high IRQL situations).
 */
void PriorityQueuePop(struct priority_queue* queue) {
    if (queue->size == 0) {
        PanicEx(PANIC_PRIORITY_QUEUE, "pop called on empty");
    }

    --queue->size;
    for (int i = 0; i < queue->qwords_per_element; ++i) {
        queue->array[i] = queue->array[queue->size * queue->qwords_per_element + i];
    }

    Heapify(queue, 0);
}

int PriorityQueueGetCapacity(struct priority_queue* queue) {
    return queue->capacity;
}

int PriorityQueueGetUsedSize(struct priority_queue* queue) {
    return queue->size;
}
```

File: ./adt/stackadt.c

```
#include <common.h>
#include <linkedlist.h>
#include <stackadt.h>
#include <heap.h>
#include <assert.h>
#include <panic.h>

struct stack_adt {
    struct linked_list* list;
};

struct stack_adt* StackAdtCreate(void) {
    struct stack_adt* stack = AllocHeap(sizeof(struct stack_adt));
    stack->list = LinkedListCreate();
    return stack;
}

void StackAdtDestroy(struct stack_adt* stack) {
    LinkedListDestroy(stack->list);
    FreeHeap(stack);
}

void StackAdtPush(struct stack_adt* stack, void* data) {
    LinkedListInsertStart(stack->list, data);
}

void* StackAdtPeek(struct stack_adt* stack) {
    return LinkedListGetDataFromNode(LinkedListGetFirstNode(stack->list));
}

void* StackAdtPop(struct stack_adt* stack) {
    void* data = StackAdtPeek(stack);
    LinkedListDeleteIndex(stack->list, 0);
    return data;
}

int StackAdtSize(struct stack_adt* stack) {
    return LinkedListSize(stack->list);
}
```

File: ./adt/blockingbuffer.c

```c

#include <heap.h>
#include <assert.h>
#include <common.h>
#include <spinlock.h>
#include <semaphore.h>
#include <errno.h>
#include <thread.h>
#include <panic.h>
#include <log.h>
#include <irql.h>

struct blocking_buffer {
    uint8_t* buffer;
    int total_size;
    int used_size;
    int start_pos;
    int end_pos;
    struct semaphore* sem;
    struct semaphore* reverse_sem;
    struct spinlock lock;
};

struct blocking_buffer* BlockingBufferCreate(int size) {
    assert(size > 0);

    struct blocking_buffer* buffer = AllocHeap(sizeof(struct blocking_buffer));
    buffer->buffer = AllocHeap(size);
    buffer->total_size = size;
    buffer->used_size = 0;
    buffer->start_pos = 0;
    buffer->end_pos = 0;
    buffer->sem = CreateSemaphore("bb get", size, size);
    buffer->reverse_sem = CreateSemaphore("bb add", size, 0);
    InitSpinlock(&buffer->lock, "blocking buffer", IRQL_SCHEDULER);
    return buffer;
}

void BlockingBufferDestroy(struct blocking_buffer* buffer) {
    FreeHeap(buffer->sem);
    FreeHeap(buffer->buffer);
    FreeHeap(buffer);
}

int BlockingBufferAdd(struct blocking_buffer* buffer, uint8_t c, bool block) {
    int res = AcquireSemaphore(buffer->reverse_sem, block ? -1 : 0);

    if (!block && res != 0) {
        return ENOBUFS;
    }

    AcquireSpinlockIrql(&buffer->lock);

    assert(buffer->used_size != buffer->total_size);

    buffer->buffer[buffer->end_pos] = c;
    buffer->end_pos = (buffer->end_pos + 1) % buffer->total_size;
    buffer->used_size++;

    /*
     * Wake up someone waiting for a character to enter the buffer - or make it so
     * next time someone wants a character they can grab it straight away.
     */
    ReleaseSpinlockIrql(&buffer->lock);
    ReleaseSemaphore(buffer->sem);
    return 0;
}

static uint8_t BlockingBufferGetAfterAcquisition(struct blocking_buffer* buffer) {
    AcquireSpinlockIrql(&buffer->lock);

    uint8_t c = buffer->buffer[buffer->start_pos];
    buffer->start_pos = (buffer->start_pos + 1) % buffer->total_size;
    buffer->used_size--;

    ReleaseSpinlockIrql(&buffer->lock);
    ReleaseSemaphore(buffer->reverse_sem);

    return c;
}

uint8_t BlockingBufferGet(struct blocking_buffer* buffer) {
    /*
     * Wait for there to be something to actually read.
     */
    AcquireSemaphore(buffer->sem, -1);
    return BlockingBufferGetAfterAcquisition(buffer);
}

int BlockingBufferTryGet(struct blocking_buffer* buffer, uint8_t* c) {
    assert(c != NULL);

    int result = AcquireSemaphore(buffer->sem, 0);
    if (result == 0) {
        *c = BlockingBufferGetAfterAcquisition(buffer);
        return 0;

    } else {
        return result;
    }
}
```

File: ./adt/linkedlist.c

```c
#include <common.h>
#include <linkedlist.h>
#include <heap.h>
#include <assert.h>
#include <panic.h>

struct linked_list_node {
    void* data;
    struct linked_list_node* next;
};

struct linked_list {
    int size;
    struct linked_list_node* head;
    struct linked_list_node* tail;
};

struct linked_list* LinkedListCreate(void) {
    struct linked_list* list = AllocHeap(sizeof(struct linked_list));
    list->size = 0;
    list->head = NULL;
    list->tail = NULL;
```

```
23      return list;
24  }
25
26  void LinkedListInsertStart(struct linked_list* list, void* data) {
27      struct linked_list_node* node = AllocHeap(sizeof(struct linked_list_node));
28      node->data = data;
29      node->next = list->tail;
30
31      if (list->head == NULL) {
32          assert(list->tail == NULL);
33          list->tail = node;
34      }
35
36      list->head = node;
37      list->size++;
38  }
39
40  void LinkedListInsertEnd(struct linked_list* list, void* data) {
41      if (list->tail == NULL) {
42          assert(list->head == NULL);
43          list->tail = AllocHeap(sizeof(struct linked_list_node));
44          list->head = list->tail;
45
46      } else {
47          list->tail->next = AllocHeap(sizeof(struct linked_list_node));
48          list->tail = list->tail->next;
49      }
50
51      list->tail->data = data;
52      list->tail->next = NULL;
53      list->size++;
54  }
55
56  bool LinkedListContains(struct linked_list* list, void* data) {
57      return LinkedListGetIndex(list, data) != -1;
58  }
59
60  int LinkedListGetIndex(struct linked_list* list, void* data) {
61      struct linked_list_node* iter = list->head;
62      int i = 0;
63      while (iter != NULL) {
64          if (iter->data == data) {
65              return i;
66          }
67          ++i;
68          iter = iter->next;
69      }
70      return -1;
71  }
72
73  void* LinkedListGetData(struct linked_list* list, int index) {
74      struct linked_list_node* iter = list->head;
75      int i = 0;
76      while (iter != NULL) {
77          if (i == index) {
78              return iter->data;
79          }
80          ++i;
81          iter = iter->next;
82      }
83      Panic(PANIC_LINKED_LIST);
84  }
85
86  static void ProperDelete(struct linked_list* list, struct linked_list_node* iter, struct linked_list_node* prev) {
87      if (iter == list->head) {
88          list->head = list->head->next;
89      } else {
90          prev->next = iter->next;
91      }
92
93      if (iter == list->tail) {
94          list->tail = prev;
95      }
96
97      FreeHeap(iter);
98      list->size--;
99  }
100
101 bool LinkedListDeleteIndex(struct linked_list* list, int index) {
102     if (index >= list->size || index < 0) {
103         return false;
104     }
105
106     struct linked_list_node* iter = list->head;
107     struct linked_list_node* prev = NULL;
108
109     int i = 0;
110     while (iter != NULL) {
111         if (i == index) {
112             ProperDelete(list, iter, prev);
113             return true;
114         }
115         ++i;
116         prev = iter;
117         iter = iter->next;
118     }
119
120     return false;
121 }
122
123 bool LinkedListDeleteData(struct linked_list* list, void* data) {
124     return LinkedListDeleteIndex(list, LinkedListGetIndex(list, data));
125 }
126
127 int LinkedListSize(struct linked_list* list) {
128     return list->size;
129 }
130
131 void LinkedListDestroy(struct linked_list* list) {
132     while (list->size > 0) {
133         LinkedListDeleteIndex(list, 0);
134     }
135     FreeHeap(list);
136 }
137
138 struct linked_list_node* LinkedListGetFirstNode(struct linked_list* list) {
139     if (list == NULL) {
140         Panic(PANIC_LINKED_LIST);
141     }
142     return list->head;
143 }
144
145 struct linked_list_node* LinkedListGetNextNode(struct linked_list_node* prev_node) {
146     if (prev_node == NULL) {
147         Panic(PANIC_LINKED_LIST);
148     }
149     return prev_node->next;
150 }
151
152 void* LinkedListGetDataFromNode(struct linked_list_node* node) {
```

```
153     if (node == NULL) {
154         Panic(PANIC_LINKED_LIST);
155     }
156     return node->data;
157 }
```

File: ./vfs/openfile.c

```
1  #include <openfile.h>
2  #include <spinlock.h>
3  #include <assert.h>
4  #include <heap.h>
5  #include <irql.h>
6
7  /**
8   * Creates a new open file from an opened vnode. An open file is used to link a vnode with corresponding data,
9   * about a particular instance of opening a file: such as a seek position, and ability to read or write; and is
10  * used to maintain file descriptor tables for the C userspace library.
11  *
12  * Open files maintain a reference counter that can be incremented and decremented with ReferenceOpenFile and
13  * DereferenceOpenFile. A newly created open file has a reference count of 1. When the count reaches 0, the memory
14  * is freed.
15  *
16  * @param node      The already-open vnode to wrap with additional data
17  * @param mode      The Unix-permissions passed to the OpenFile when the vnode was opened. Ignored by the kernel
18  *                      so far.
19  * @param flag      The flags that were passed to OpenFile when the vnode was opened. Should be a bitfield consisting of
20  *                      zero or more of: O_RDONLY, O_WRONLY, O_TRUNC, O_CREAT. See OpenFile for details. The storage of these
21  *                      flags in an open file should only be of interest to usermode programs - the flags here may be be zero
22  *                      if the open file was created within the kernel, so the flags should be ignored within the kernel.
23  * @param can_read  Whether or not this open file is allowed to make read operations on the underlying vnode
24  * @param can_write Whether or not this open file is allowed to make write operations on the underlying vnode
25  *
26  * @return A pointer to the newly created open file.
27  */
28 struct open_file* CreateOpenFile(struct vnode* node, int mode, int flags, bool can_read, bool can_write) {
29     MAX_IRQL(IRQL_SCHEDULER);
30
31  struct open_file* file = AllocHeap(sizeof(struct open_file));
32  file->reference_count = 1;
33  file->node = node;
34  file->can_read = can_read;
35  file->can_write = can_write;
36  file->initial_mode = mode;
37  file->flags = flags;
38  file->seek_position = 0;
39  InitSpinlock(&file->reference_count_lock, "open file", IRQL_SCHEDULER);
40
41  return file;
42 }
43
44 /**
45  * Increments the reference counter for an opened file. This should be called everytime a reference to
46  * the open file is kept, so that its memory can be managed correctly.
47  *
48  * @param file The open file to reference
49  */
50 void ReferenceOpenFile(struct open_file* file) {
51     MAX_IRQL(IRQL_SCHEDULER);
52  assert(file != NULL);
53
54     AcquireSpinlockIrql(&file->reference_count_lock);
55     file->reference_count++;
56     ReleaseSpinlockIrql(&file->reference_count_lock);
57 }
58
59 /**
60  * Decrements the reference counter for an opened file. Should be called whenever a reference to the open
61  * file is removed. If the reference counter reaches zero, the memory behind the open file will be freed.
62  * The underlying vnode within the open file is not dereferenced - this should be done prior to calling this
63  * function.
64  *
65  * @param file The open file to dereference
66  */
67 void DereferenceOpenFile(struct open_file* file) {
68     MAX_IRQL(IRQL_SCHEDULER);
69     assert(file != NULL);
70
71  AcquireSpinlockIrql(&file->reference_count_lock);
72
73     assert(file->reference_count > 0);
74     file->reference_count--;
75
76     if (file->reference_count == 0) {
77         /*
78         * Must release the lock before we delete it so we can put interrupts back on
79         */
80         ReleaseSpinlockIrql(&file->reference_count_lock);
81         FreeHeap(file);
82         return;
83     }
84
85     ReleaseSpinlockIrql(&file->reference_count_lock);
86 }
```

File: ./vfs/diskutil.c

```
1   #include <diskutil.h>
2   #include <string.h>
3   #include <irql.h>
4   #include <assert.h>
5   #include <spinlock.h>
6   #include <vfs.h>
7   #include <errno.h>
8   #include <log.h>
9   #include <irql.h>
10  #include <partition.h>
11  #include <sys/stat.h>
12
13  /*
14   * Stores how many disks of each type have been allocated so far.
15   */
16  static int type_table[__DISKUTIL_NUM_TYPES];
17
18  /*
19   * Protects `type_table` and `next_mounted_disk_num`
20   */
21  static struct spinlock type_table_lock;
22
23  /*
24   * Filesystems get mounted to the VFS as drvX:, where X is an increasing number.
25   * This value stores the number the next disk gets.
26   */
27  static int next_mounted_disk_num = 0;
```

```
28
29 /*
30  * Maps a drive type to a string that will form part of the drive name.
31  */
32 static char* type_strings[__DISKUTIL_NUM_TYPES] = {
33     [DISKUTIL_TYPE_FIXED]      = "hd",
34     [DISKUTIL_TYPE_FLOPPY]     = "fd",
35     [DISKUTIL_TYPE_NETWORK]    = "net",
36     [DISKUTIL_TYPE_OPTICAL]    = "cd",
37     [DISKUTIL_TYPE_OTHER]      = "other",
38     [DISKUTIL_TYPE_RAM]        = "ram",
39     [DISKUTIL_TYPE_REMOVABLE]  = "rm",
40     [DISKUTIL_TYPE_VIRTUAL]    = "virt",
41 };
42
43 /**
44  * Initialises the disk utility functions. Must be called before any partitions
45  * are created or any drive names are generated.
46  */
47 void InitDiskUtil(void) {
48     EXACT_IRQL(IRQL_STANDARD);
49     InitSpinlock(&type_table_lock, "diskutil", IRQL_SCHEDULER);
50     memset(type_table, 0, sizeof(type_table));
51 }
52
53 /**
54  * Given a string and an integer less than 1000, it converts the integer to a
55  * string, and appends it to the end of the existing string, in place. The
56  * string should have enough buffer allocated to fit the number.
57  *
58  * Returns 0 on success, else EINVAL.
59  */
60 static int AppendNumberToString(char* str, int num) {
61     if (str == NULL || num >= 1000) {
62         return EINVAL;
63     }
64
65     char num_str[4];
66     memset(num_str, 0, 4);
67     if (num < 10) {
68         num_str[0] = num + '0';
69
70     } else if (num < 100) {
71         num_str[0] = (num / 10) + '0';
72         num_str[1] = (num % 10) + '0';
73
74     } else {
75         num_str[0] = (num / 100) + '0';
76         num_str[1] = ((num / 10) % 10) + '0';
77         num_str[2] = (num % 10) + '0';
78     }
79
80     strcat(str, num_str);
81     return 0;
82 }
83
84 /**
85  * Returns the name the next-mounted filesystem should receive (e.g. drv0,
86  * drv1, etc.) Each call to this function will return a different string. The
87  * caller is responsible for freeing the returned string.
88  *
89  * @return A caller-free string representing the drive name.
90  *
91  * @maxirql IRQL_SCHEDULER
92  */
93 char* GenerateNewMountedDiskName() {
94     MAX_IRQL(IRQL_SCHEDULER);
95
96     char name[16];
97     strcpy(name, "drv");
98
99     AcquireSpinlockIrql(&type_table_lock);
100     int disk_num = next_mounted_disk_num++;
101     ReleaseSpinlockIrql(&type_table_lock);
102
103     AppendNumberToString(name, disk_num);
104     return strdup(name);
105 }
106
107 /**
108  * Returns the name the next-mounted raw disk should receive, based on its type (e.g. raw-hd0, raw-hd1,
109  * raw-fd0). Each call to this function will return a different string. The caller is responsible for
110  * freeing the returned string.
111  *
112  * @param type The type of disk (one of DISKUTIL_TYPE_...)
113  * @return The caller-free string representing the drive name.
114  *
115  * @maxirql IRQL_SCHEDULER
116  */
117 char* GenerateNewRawDiskName(int type) {
118     MAX_IRQL(IRQL_SCHEDULER);
119
120     char name[16];
121     strcpy(name, "raw-");
122
123     if (type >= __DISKUTIL_NUM_TYPES || type < 0) {
124         type = DISKUTIL_TYPE_OTHER;
125     }
126
127     strcat(name, type_strings[type]);
128
129     AcquireSpinlockIrql(&type_table_lock);
130     int disk_num = type_table[type]++;
131     ReleaseSpinlockIrql(&type_table_lock);
132
133     AppendNumberToString(name, disk_num);
134     return strdup(name);
135 }
136
137 /**
138  * Generates and returns the name of a partition from its partition index within a drive (e.g. part0, part1)
139  * The caller is responsible for freeing the returned string.
140  */
141 static char* GetPartitionNameString(int index) {
142     char name[16];
143     strcpy(name, "part");
144     AppendNumberToString(name, index);
145     return strdup(name);
146 }
147
148 /**
149  * Given a disk, this function detects, creates and mounts partitions on that disk.
150  * For each detected partition, the filesystem is also detected, and that is mounted if it exists.
151  * If the disk has no partitions, a 'whole disk partition' will be created, the filesystem will
152  * still be detected. This should only be called once per disk, after it has been initialised.
153  *
154  * @param disk The disk to scan for partitions
155  *
156  * @maxirql IRQL_STANDARD
157  */
```

```
158 void CreateDiskPartitions(struct open_file* disk) {
159     EXACT_IRQL(IRQL_STANDARD);
160 
161     struct open_file** partitions = GetPartitionsForDisk(disk);
162 
163     if (partitions == NULL || partitions[0] == NULL) {
164         LogWriteSerial("no partition table...\n");
165         struct stat st;
166         int res = VnodeOpStat(disk->node, &st);
167         if (res != 0) {
168             return;
169         }
170 
171         struct vnode* whole_disk_partition = CreatePartition(disk, 0, st.st_size, 0, st.st_blksize, 0, false)->node;
172         VnodeOpCreate(disk->node, &whole_disk_partition, GetPartitionNameString(0), 0, 0);
173         return;
174     }
175 
176     for (int i = 0; partitions[i]; ++i) {
177         struct vnode* partition = partitions[i]->node;
178         char* str = GetPartitionNameString(i);
179         VnodeOpCreate(disk->node, &partition, str, 0, 0);
180     }
181 }
182 
183 void InitDiskPartitionHelper(struct disk_partition_helper* helper) {
184     helper->num_partitions = 0;
185 }
186 
187 int DiskFollowHelper(struct disk_partition_helper* helper, struct vnode** out, const char* name) {
188     for (int i = 0; i < helper->num_partitions; ++i) {
189         if (!strcmp(helper->partition_names[i], name)) {
190             *out = helper->partitions[i];
191             return 0;
192         }
193     }
194 
195     *out = NULL;
196     return EINVAL;
197 }
198 
199 int DiskCreateHelper(struct disk_partition_helper* helper, struct vnode** in, const char* name) {
200     if (helper->num_partitions >= MAX_PARTITIONS_PER_DISK) {
201         return EINVAL;
202     }
203 
204     helper->partitions[helper->num_partitions] = *in;
205     helper->partition_names[helper->num_partitions] = (char*) name;
206     helper->num_partitions++;
207     return 0;
208 }
```

File: ./vfs/transfer.c

```
1 
2  #include <transfer.h>
3  #include <assert.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <virtual.h>
7  #include <arch.h>
8  #include <log.h>
9 
10 static int ValidateCopy(const void* user_addr, size_t size, bool write) {
11     size_t initial_address = (size_t) user_addr;
12 
13     /*
14     * Check if the memory range starts in user memory.
15     */
16     if (initial_address < ARCH_USER_AREA_BASE || initial_address >= ARCH_USER_AREA_LIMIT) {
17         return EINVAL;
18     }
19 
20     size_t final_address = initial_address + size;
21 
22     /*
23     * Check for overflow when the initial address and size are added. If it would overflow,
24     * we cancel the operation, as the user is obviously outside their range.
25     */
26     if (final_address < initial_address) {
27         return EINVAL;
28     }
29 
30     /*
31     * Ensure the end of the memory range is in user memory. As user memory must be contiguous,
32     * this ensures the entire range is in user memory.
33     */
34     if (final_address < ARCH_USER_AREA_BASE || final_address >= ARCH_USER_AREA_LIMIT) {
35         return EINVAL;
36     }
37 
38     /*
39     * We must now check if the USER (and possibly WRITE) bits are set on the memory pages
40     * being accessed.
41     */
42     size_t initial_page = initial_address / ARCH_PAGE_SIZE;
43     size_t pages = BytesToPages(size);
44 
45     for (size_t i = 0; i < pages; ++i) {
46         size_t page = initial_page + i;
47         size_t permissions = GetVirtPermissions(page * ARCH_PAGE_SIZE);
48 
49         if (permissions == 0) {
50             return EINVAL;
51         }
52 
53         if (!(permissions & VM_READ)) {
54             return EINVAL;
55         }
56         if (!(permissions & VM_USER)) {
57             return EINVAL;
58         }
59         if (write && !(permissions & VM_WRITE)) {
60             return EINVAL;
61         }
62         if (write && (permissions & VM_EXEC)) {
63             return EINVAL;
64         }
65         if (write && (permissions & VM_EXEC)) {
66             return EINVAL;
67         }
68     }
69 
70     return 0;
71 }
72 
73 static int CopyIntoKernel(void* kernel_addr, const void* user_addr, size_t size) {
```

```
    int status = ValidateCopy(user_addr, size, false);
    if (status != 0) {
        return status;
    }

    inline_memcpy(kernel_addr, user_addr, size);
    return 0;
}

static int CopyOutOfKernel(const void* kernel_addr, void* user_addr, size_t size) {
    int status = ValidateCopy(user_addr, size, true);
    if (status != 0) {
        return status;
    }

    inline_memcpy(user_addr, kernel_addr, size);
    return 0;
}

int PerformTransfer(void* trusted_buffer, struct transfer* untrusted_buffer, uint64_t len) {
    assert(trusted_buffer != NULL);
    assert(untrusted_buffer != NULL && untrusted_buffer->address != NULL);
    assert(untrusted_buffer->direction == TRANSFER_READ || untrusted_buffer->direction == TRANSFER_WRITE);

    size_t amount_to_copy = MIN(len, untrusted_buffer->length_remaining);
    if (amount_to_copy == 0) {
        return 0;
    }

    if (untrusted_buffer->type == TRANSFER_INTRA_KERNEL) {
        if (untrusted_buffer->direction == TRANSFER_READ) {
            memmove(untrusted_buffer->address, trusted_buffer, amount_to_copy);

        } else {
            memmove(trusted_buffer, untrusted_buffer->address, amount_to_copy);
        }

    } else {
        int result;

        /*
        * This is from the kernel's perspective of the operations.
        */
        if (untrusted_buffer->direction == TRANSFER_READ) {
            result = CopyOutOfKernel((const void*) trusted_buffer, untrusted_buffer->address, amount_to_copy);

        } else {
            result = CopyIntoKernel(trusted_buffer, (const void*) untrusted_buffer->address, amount_to_copy);

        }

        if (result != 0) {
            return result;
        }
    }

    untrusted_buffer->length_remaining -= amount_to_copy;
    untrusted_buffer->offset += amount_to_copy;
    untrusted_buffer->address = ((uint8_t*) untrusted_buffer->address) + amount_to_copy;

    return 0;
}

int WriteStringToUsermode(const char* trusted_string, char* untrusted_buffer, uint64_t max_length) {
    struct transfer tr = CreateTransferWritingToUser(untrusted_buffer, max_length, 0);
    int result;

    /*
     * Limit the size of the string by the maximimum length. We use <, and a -1 in the other case,
     * as we need to ensure the null terminator fits.
     */
    uint64_t size = strlen(trusted_string) < max_length ? strlen(trusted_string) : max_length - 1;
    result = PerformTransfer((void*) trusted_string, &tr, size);

    if (result != 0) {
        return result;
    }

    uint8_t zero = 0;
    return PerformTransfer(&zero, &tr, 1);
}

int ReadStringFromUsermode(char* trusted_buffer, const char* untrusted_string, uint64_t max_length) {
    struct transfer tr = CreateTransferReadingFromUser(untrusted_string, max_length, 0);
    size_t i = 0;

    while (max_length-- > 1) {
        char c;
        int result = PerformTransfer(&c, &tr, 1);
        if (result != 0) {
            return result;
        }
        trusted_buffer[i++] = c;
        if (c == 0) {
            break;
        }
    }

    trusted_buffer[i] = 0;
    return 0;
}

int WriteWordToUsermode(size_t* location, size_t value) {
    struct transfer io = CreateTransferWritingToUser(location, sizeof(size_t), 0);
    int res = PerformTransfer(&value, &io, sizeof(size_t));
    if (io.length_remaining != 0) {
        return EINVAL;
    }
    return res;
}

int ReadWordFromUsermode(size_t* location, size_t* output) {
    struct transfer io = CreateTransferReadingFromUser(location, sizeof(size_t), 0);
    int res = PerformTransfer(output, &io, sizeof(size_t));
    if (io.length_remaining != 0) {
        return EINVAL;
    }
    return res;
}

static struct transfer CreateTransfer(void* addr, uint64_t length, uint64_t offset, int direction, int type) {
    struct transfer trans;
    trans.address = addr;
    trans.direction = direction;
    trans.length_remaining = length;
    trans.offset = offset;
    trans.type = type;
    return trans;
}

```

```
struct transfer CreateKernelTransfer(void* addr, uint64_t length, uint64_t offset, int direction) {
    return CreateTransfer(addr, length, offset, direction, TRANSFER_INTRA_KERNEL);
}

struct transfer CreateTransferWritingToUser(void* untrusted_addr, uint64_t length, uint64_t offset) {
    /*
     * When we "write to the user", we are doing so because the user is trying to *read* from the kernel.
     * i.e. someone is doing an "untrusted read" of kernel data (i.e. a TRANSFER_READ).
     */
    return CreateTransfer(untrusted_addr, length, offset, TRANSFER_READ, TRANSFER_USERMODE);
}

struct transfer CreateTransferReadingFromUser(const void* untrusted_addr, uint64_t length, uint64_t offset) {
    /*
     * When we "read from the user", we are doing so because the user is trying to *write* to the kernel.
     * i.e. as they are writing to the kernel, it is an "untrusted write" (i.e. a TRANSFER_WRITE).
     */
    return CreateTransfer((void*) untrusted_addr, length, offset, TRANSFER_WRITE, TRANSFER_USERMODE);
}
```

File: ./vfs/filedes.c

```
#include <filedes.h>
#include <errno.h>
#include <string.h>
#include <common.h>
#include <semaphore.h>
#include <vfs.h>
#include <fcntl.h>
#include <log.h>
#include <heap.h>
#include <irql.h>

struct filedes_entry {
    /*
    * Set to NULL if this entry isn't in use.
    */
    struct open_file* file;

    /*
    * The only flag that can live here is FD_CLOEXEC. All other flags live on the filesytem
    * level. This is because FD_CLOEXEC is a property of the file descriptor, not the underlying
    * file itself. (This is important in how dup() works.).
    *
    * Note that we set FD_CLOEXEC == O_CLOEXEC.
    */
    int flags;
};

/*
* The table of all of the file descriptors in use by a process.
*/
struct filedes_table {
    struct semaphore* lock;
    struct filedes_entry* entries;
};

struct filedes_table* CreateFileDescriptorTable(void) {
    struct filedes_table* table = AllocHeap(sizeof(struct filedes_table));

    table->lock = CreateMutex("filedes");
    table->entries = AllocHeapEx(sizeof(struct filedes_entry) * MAX_FD_PER_PROCESS, HEAP_ALLOW_PAGING);

    for (int i = 0; i < MAX_FD_PER_PROCESS; ++i) {
        table->entries[i].file = NULL;
    }

    return table;
}

struct filedes_table* CopyFileDescriptorTable(struct filedes_table* original) {
    struct filedes_table* new_table = CreateFileDescriptorTable();

    AcquireMutex(original->lock, -1);
    memcpy(new_table->entries, original->entries, sizeof(struct filedes_entry) * MAX_FD_PER_PROCESS);
    ReleaseMutex(original->lock);

    return new_table;
}

int CreateFileDescriptor(struct filedes_table* table, struct open_file* file, int* fd_out, int flags) {
    if ((flags & ~O_CLOEXEC) != 0) {
        return EINVAL;
    }

    AcquireMutex(table->lock, -1);

    for (int i = 0; i < MAX_FD_PER_PROCESS; ++i) {
        if (table->entries[i].file == NULL) {
            table->entries[i].file = file;
            table->entries[i].flags = flags;
            ReleaseMutex(table->lock);
            *fd_out = i;
            return 0;
        }
    }

    ReleaseMutex(table->lock);
    return EMFILE;
}

int RemoveFileDescriptor(struct filedes_table* table, struct open_file* file) {
    AcquireMutex(table->lock, -1);

    for (int i = 0; i < MAX_FD_PER_PROCESS; ++i) {
        if (table->entries[i].file == file) {
            table->entries[i].file = NULL;
            ReleaseMutex(table->lock);
            return 0;
        }
    }

    ReleaseMutex(table->lock);
    return EINVAL;
}

int GetFileFromDescriptor(struct filedes_table* table, int fd, struct open_file** out) {
    if (out == NULL || fd < 0 || fd >= MAX_FD_PER_PROCESS) {
        *out = NULL;
        return out == NULL ? EINVAL : EBADF;
    }

    AcquireMutex(table->lock, -1);
    struct open_file* result = table->entries[fd].file;
    ReleaseMutex(table->lock);

    *out = result;
```

```
    return result == NULL ? EBADF : 0;
}

int HandleFileDescriptorsOnExec(struct filedes_table* table) {
    AcquireMutex(table->lock, -1);

    for (int i = 0; i < MAX_FD_PER_PROCESS; ++i) {
        if (table->entries[i].file != NULL) {
            if (table->entries[i].flags & O_CLOEXEC) {
                struct open_file* file = table->entries[i].file;
                table->entries[i].file = NULL;
                int res = CloseFile(file);
                if (res != 0) {
                    ReleaseMutex(table->lock);
                    return res;
                }
            }
        }
    }

    ReleaseMutex(table->lock);
    return 0;
}

int DuplicateFileDescriptor(struct filedes_table* table, int oldfd, int* newfd) {
    AcquireMutex(table->lock, -1);

    struct open_file* original_file;
    int res = GetFileFromDescriptor(table, oldfd, &original_file);
    if (res != 0 || original_file == NULL) {
        ReleaseMutex(table->lock);
        return EBADF;
    }

    for (int i = 0; i < MAX_FD_PER_PROCESS; ++i) {
        if (table->entries[i].file == NULL) {
            table->entries[i].file = original_file;
            table->entries[i].flags = 0;
            ReleaseMutex(table->lock);
            *newfd = i;
            return 0;
        }
    }

    ReleaseMutex(table->lock);
    return EMFILE;
}

int DuplicateFileDescriptor2(struct filedes_table* table, int oldfd, int newfd, int flags) {
    if (flags & ~O_CLOEXEC) {
        return EINVAL;
    }

    AcquireMutex(table->lock, -1);

    struct open_file* original_file;
    int res = GetFileFromDescriptor(table, oldfd, &original_file);

    /*
    * "If oldfd is not a valid file descriptor, then the call fails,
    * and newfd is not closed."
    */
    if (res != 0 || original_file == NULL) {
        ReleaseMutex(table->lock);
        return EBADF;
    }

    /*
    * "If oldfd is a valid file descriptor, and newfd has the same
    * value as oldfd, then dup2() does nothing..."
    */
    if (oldfd == newfd) {
        ReleaseMutex(table->lock);
        return 0;
    }

    struct open_file* current_file;
    res = GetFileFromDescriptor(table, oldfd, &current_file);
    if (res == 0 && current_file != NULL) {
        /*
        * "If the file descriptor newfd was previously open, it is closed
        * before being reused; the close is performed silently (i.e., any
        * errors during the close are not reported by dup2())."
        */
        CloseFile(current_file);
    }

    table->entries[newfd].file = original_file;
    table->entries[newfd].flags = flags;

    ReleaseMutex(table->lock);
    return 0;
}
```

File: ./vfs/vnode.c

```
#include <vnode.h>
#include <spinlock.h>
#include <assert.h>
#include <log.h>
#include <spinlock.h>
#include <errno.h>
#include <heap.h>
#include <dirent.h>
#include <irql.h>

/*
* vfs/vnode.c - Virtual Filesystem Nodes
*
* Each vnode represents an abstract file, such as a file, directory or device.
*/

/*
* Allocate and initialise a vnode. The reference count is initialised to 1.
*/
struct vnode* CreateVnode(struct vnode_operations ops) {
    struct vnode* node = AllocHeap(sizeof(struct vnode));
    node->ops = ops;
    node->reference_count = 1;
    node->data = NULL;
    InitSpinlock(&node->reference_count_lock, "vnode refcnt", IRQL_SCHEDULER);
    return node;
}

/*
* Cleanup and free an abstract file node.
```

```
*/
static void DestroyVnode(struct vnode* node) {
    /*
    * The lock can't be held during this process, otherwise the lock will
    * get freed before it is released (which is bad, as we must release it
    * to get interrupts back on).
    */

    assert(node != NULL);
    assert(node->reference_count == 0);

    FreeHeap(node);
}

/*
* Ensures that a vnode is valid.
*/
static void CheckVnode(struct vnode* node) {
    assert(node != NULL);

    if (IsSpinlockHeld(&node->reference_count_lock)) {
        assert(node->reference_count > 0);
    } else {
        AcquireSpinlockIrql(&node->reference_count_lock);
        assert(node->reference_count > 0);
        ReleaseSpinlockIrql(&node->reference_count_lock);
    }
}


/*
* Increments a vnode's reference counter. Used whenever a vnode is 'given' to someone.
*/
void ReferenceVnode(struct vnode* node) {
    assert(node != NULL);

    AcquireSpinlockIrql(&node->reference_count_lock);
    node->reference_count++;
    ReleaseSpinlockIrql(&node->reference_count_lock);
}

/*
* Decrements a vnode's reference counter, destorying it if it reaches zero.
* It should be called to free a vnode 'given' to use when it is no longer needed.
*/
void DereferenceVnode(struct vnode* node) {
    CheckVnode(node);
    AcquireSpinlockIrql(&node->reference_count_lock);

    assert(node->reference_count > 0);
    node->reference_count--;

    if (node->reference_count == 0) {
        VnodeOpClose(node);

        /*
        * Must release the lock before we delete it so we can put interrupts back on
        */
        ReleaseSpinlockIrql(&node->reference_count_lock);

        DestroyVnode(node);
        return;
    }

    ReleaseSpinlockIrql(&node->reference_count_lock);
}


/*
* Wrapper functions for performing operations on a vnode. Also
* performs validation on the vnode.
*/
int VnodeOpCheckOpen(struct vnode* node, const char* name, int flags) {
    if (node == NULL) {
        return EINVAL;
    }
    if (node->ops.check_open == NULL) {
        return 0;
    }
    return node->ops.check_open(node, name, flags);
}

int VnodeOpRead(struct vnode* node, struct transfer* io) {
    if (node == NULL || node->ops.read == NULL || io->direction != TRANSFER_READ) {
        return EINVAL;
    }
    return node->ops.read(node, io);
}

int VnodeOpWrite(struct vnode* node, struct transfer* io) {
    if (node == NULL || node->ops.write == NULL || io->direction != TRANSFER_WRITE) {
        return EINVAL;
    }
    return node->ops.write(node, io);
}

int VnodeOpIoctl(struct vnode* node, int command, void* buffer) {
    if (node == NULL || node->ops.ioctl == NULL) {
        return EINVAL;
    }
    return node->ops.ioctl(node, command, buffer);
}

bool VnodeOpIsSeekable(struct vnode* node) {
    if (node == NULL || node->ops.is_seekable == NULL) {
        return false;
    }
    return node->ops.is_seekable(node);
}

int VnodeOpCheckTty(struct vnode* node) {
    if (node == NULL || node->ops.check_tty == NULL) {
        return ENOTTY;
    }
    return node->ops.check_tty(node);
}

int VnodeOpClose(struct vnode* node) {
    if (node == NULL || node->reference_count != 0) {
        return EINVAL;
    }
    if (node->ops.close == NULL) {
        return 0;
    }
    return node->ops.close(node);
}

int VnodeOpCreate(struct vnode* node, struct vnode** out, const char* name, int flags, mode_t mode) {
    if (node == NULL || node->ops.create == NULL) {
        return EINVAL;
```

```
    }
    return node->ops.create(node, out, name, flags, mode);
}

int VnodeOpTruncate(struct vnode* node, off_t offset) {
    if (node == NULL || node->ops.truncate == NULL) {
        return EINVAL;
    }
    return node->ops.truncate(node, offset);
}

int VnodeOpFollow(struct vnode* node, struct vnode** new_node, const char* name) {
    if (node == NULL || node->ops.follow == NULL) {
        return ENOTDIR;
    }
    return node->ops.follow(node, new_node, name);
}

uint8_t VnodeOpDirentType(struct vnode* node) {
    if (node == NULL || node->ops.dirent_type == NULL) {
        return DT_UNKNOWN;
    }
    return node->ops.dirent_type(node);
}

int VnodeOpStat(struct vnode* node, struct stat* stat) {
    if (node == NULL || node->ops.stat == NULL) {
        return EINVAL;
    }
    return node->ops.stat(node, stat);
}

int VnodeOpWait(struct vnode* node, int flags, uint64_t timeout_ms) {
    if (node == NULL) {
        return EINVAL;
    }
    if (node->ops.wait == NULL) {
        return 0;
    }
    return node->ops.wait(node, flags, timeout_ms);
}
```

File: ./vfs/vfs.c

```
#include <vfs.h>
#include <spinlock.h>
#include <irql.h>
#include <log.h>
#include <assert.h>
#include <virtual.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
#include <heap.h>
#include <avl.h>
#include <fcntl.h>
#include <stackadt.h>

/*
* Try not to have non-static functions that return in any way a struct vnode*, as it
* probably means you need to use the reference/dereference functions.
*
*/

/*
* Maximum length of a component of a filepath (e.g. an file/directory's individual name).
*/
#define MAX_COMPONENT_LENGTH 128

/*
* Maximum total length of a path.
*/
#define MAX_PATH_LENGTH   2000

/*
* Maximum number of symbolic links to derefrence in a path before returning ELOOP.
*/
#define MAX_LOOP    5


/*
* A structure for mounted devices and filesystems.
*/
struct mounted_file {
 /* The vnode representing the device / root directory of a filesystem */
 struct open_file* node;

 /* What the device / filesystem mount is called */
 char* name;
};

static struct spinlock vfs_lock;
static struct avl_tree* mount_points = NULL;

int MountedDeviceComparator(void* a_, void* b_) {
    struct mounted_file* a = a_;
    struct mounted_file* b = b_;
    return strcmp(a->name, b->name);
}

void InitVfs(void) {
    InitSpinlock(&vfs_lock, "vfs", IRQL_SCHEDULER);
    mount_points = AvlTreeCreate();
    AvlTreeSetComparator(mount_points, MountedDeviceComparator);
}

static int CheckValidComponentName(const char* name)
{
 assert(name != NULL);

 if (name[0] == 0) {
  return EINVAL;
 }

 for (int i = 0; name[i]; ++i) {
  char c = name[i];

  if (c == '/' || c == '\\' || c == ':') {
   return EINVAL;
  }
 }

 return 0;
}

static int DoesMountPointExist(const char* name) {
    assert(name != NULL);
    assert(IsSpinlockHeld(&vfs_lock));
```

```
85
86      struct mounted_file target;
87      target.name = (char*) name;
88      if (AvlTreeContains(mount_points, &target)) {
89          return EEXIST;
90      }
91
92      return 0;
93  }
94
95  /*
96   * Given a filepath, and a pointer to an index within that filepath (representing where
97   * start searching), copies the next component into an output buffer of a given length.
98   * The index is updated to point to the start of the next component, ready for the next call.
99   *
100  * This also handles duplicated and trailing forward slashes.
101  */
102 static int GetPathComponent(const char* path, int* ptr, char* output_buffer, int max_output_length, char delimiter) {
103  int i = 0;
104
105  assert(path != NULL);
106  assert(ptr != NULL);
107  assert(max_output_length >= 1);
108  assert(delimiter != 0);
109  assert(output_buffer != NULL);
110  assert(strlen(path) >= 1);
111  assert(*ptr >= 0 && *ptr < (int) strlen(path));
112
113  /*
114   * These were meant to be caught at a higher level, so we can apply the current
115   * working directory or the current drive.
116   */
117  assert(path[0] != '/');
118  assert(path[0] != ':');
119
120  output_buffer[0] = 0;
121
122  while (path[*ptr] && path[*ptr] != delimiter) {
123   if (i >= max_output_length - 1) {
124    return ENAMETOOLONG;
125   }
126
127   /*
128    * Ensure we always have a null terminated string.
129    */
130   output_buffer[i++] = path[*ptr];
131   output_buffer[i] = 0;
132   (*ptr)++;
133  }
134
135  /*
136   * Skip past the delimiter (unless we are at the end of the string),
137   * as well as any trailing slashes (which could be after a slash delimiter, or
138   * after a colon).
139   */
140  if (path[*ptr]) {
141   do {
142    (*ptr)++;
143   } while (path[*ptr] == '/');
144  }
145
146  /*
147   * Ensure that there are no colons or backslashes in the filename itself.
148   */
149  return CheckValidComponentName(output_buffer);
150 }
151
152 static int GetFinalPathComponent(const char* path, char* output_buffer, int max_output_length) {
153  int path_ptr = 0;
154
155  int status = GetPathComponent(path, &path_ptr, output_buffer, max_output_length, ':');
156  if (status) {
157   return status;
158  }
159
160  while (path_ptr < (int) strlen(path)) {
161   status = GetPathComponent(path, &path_ptr, output_buffer, max_output_length, '/');
162   if (status) {
163    return status;
164   }
165  }
166
167  return 0;
168 }
169
170 /*
171  * Takes in a device name, without the colon, and returns its vnode.
172  * If no such device is mounted, it returns NULL.
173  */
174 static struct open_file* GetMountFromName(const char* name) {
175  assert(name != NULL);
176
177  if (mount_points == NULL) {
178   return NULL;
179  }
180
181  struct mounted_file target;
182     target.name = (char*) name;
183     struct mounted_file* mount = AvlTreeGet(mount_points, (void*) &target);
184     return mount->node;
185 }
186
187 int PAGEABLE_CODE_SECTION AddVfsMount(struct vnode* node, const char* name) {
188     MAX_IRQL(IRQL_PAGE_FAULT);
189
190     if (name == NULL || node == NULL) {
191   return EINVAL;
192  }
193
194  if (strlen(name) >= MAX_COMPONENT_LENGTH) {
195   return ENAMETOOLONG;
196  }
197
198     int status = CheckValidComponentName(name);
199  if (status != 0) {
200   return status;
201  }
202
203     AcquireSpinlockIrql(&vfs_lock);
204
205     if (DoesMountPointExist(name) == EEXIST) {
206         ReleaseSpinlockIrql(&vfs_lock);
207         return EEXIST;
208     }
209
210     struct mounted_file* mount = AllocHeap(sizeof(struct mounted_file));
211  mount->name = strdup(name);
212     mount->node = CreateOpenFile(node, 0, 0, true, true);
213
214     AvlTreeInsert(mount_points, (void*) mount);
```

```
215
216 LogWriteSerial("MOUNTED TO THE VFS: %s\n", name);
217
218     ReleaseSpinlockIrql(&vfs_lock);
219     return 0;
220 }
221
222 int PAGEABLE_CODE_SECTION RemoveVfsMount(const char* name) {
223     MAX_IRQL(IRQL_PAGE_FAULT);
224
225     if (name == NULL) {
226  return EINVAL;
227 }
228
229 if (CheckValidComponentName(name) != 0) {
230  return EINVAL;
231 }
232
233 AcquireSpinlockIrql(&vfs_lock);
234
235 /*
236 * Scan through the mount table for the device
237 */
238 struct mounted_file target;
239     target.name = (char*) name;
240
241     struct mounted_file* actual = AvlTreeGet(mount_points, &target);
242     if (actual == NULL) {
243         ReleaseSpinlockIrql(&vfs_lock);
244         return ENODEV;
245     }
246
247     assert(!strcmp(actual->name, name));
248
249     /*
250      * Decrement the reference that was initially created way back in
251      * vfs_add_device in the call to dev_create_vnode (the vnode dereference),
252      * and then the open file that was created alongside it.
253      */
254     DereferenceVnode(actual->node->node);
255     DereferenceOpenFile(actual->node);
256
257     AvlTreeDelete(mount_points, actual);
258     FreeHeap(actual->name);
259
260     ReleaseSpinlockIrql(&vfs_lock);
261     return 0;
262 }
263
264 static void CleanupVnodeStack(struct stack_adt* stack) {
265  /*
266  * We need to call dereference on each vnode in the stack before we
267  * can call StackAdtDestroy.
268  */
269  while (StackAdtSize(stack) > 0) {
270   struct vnode* node = StackAdtPop(stack);
271   DereferenceVnode(node);
272  }
273
274  StackAdtDestroy(stack);
275 }
276
277 /*
278 * Given an absolute filepath, returns the vnode representing
279 * the file, directory or device.
280 *
281 * Should only be called by vfs_open as the reference count will be incremented.
282 */
283 static int GetVnodeFromPath(const char* path, struct vnode** out, bool want_parent) {
284  assert(path != NULL);
285  assert(out != NULL);
286
287  LogWriteSerial("GetVnodeFromPath: path %s\n", path);
288
289  if (strlen(path) == 0) {
290   return EINVAL;
291  }
292  if (strlen(path) > MAX_PATH_LENGTH) {
293   return ENAMETOOLONG;
294  }
295
296  int path_ptr = 0;
297  char component_buffer[MAX_COMPONENT_LENGTH];
298
299  int err = GetPathComponent(path, &path_ptr, component_buffer, MAX_COMPONENT_LENGTH, ':');
300  if (err != 0) {
301   return err;
302  }
303
304  struct open_file* current_file = GetMountFromName(component_buffer);
305
306  /*
307  * No root device found, so we can't continue.
308  */
309  if (current_file == NULL || current_file->node == NULL) {
310   return ENODEV;
311  }
312
313  struct vnode* current_vnode = current_file->node;
314
315  /*
316  * This will be dereferenced either as we go through the loop, or
317  * after a call to vfs_close (this function should only be called
318  * by vfs_open).
319  */
320  ReferenceVnode(current_vnode);
321
322  char component[MAX_COMPONENT_LENGTH + 1];
323
324  /*
325  * To go back to a parent directory, we need to keep track of the previous component.
326  * As we can go back through many parents, we must keep track of all of them, hence we
327  * use a stack to store each vnode we encounter. We will not dereference the vnodes
328  * on the stack until the end using cleanup_vnode_stack.
329  */
330  struct stack_adt* previous_components = StackAdtCreate();
331
332  /*
333  * Iterate over the rest of the path.
334  */
335  while (path_ptr < (int) strlen(path)) {
336   int status = GetPathComponent(path, &path_ptr, component, MAX_COMPONENT_LENGTH, '/');
337   if (status != 0) {
338    DereferenceVnode(current_vnode);
339    CleanupVnodeStack(previous_components);
340    return status;
341   }
342
343   if (!strcmp(component, ".")) {
344    /*
```

```
   * This doesn't change where we point to.
   */
   continue;

  } else if (!strcmp(component, "..")) {
   if (StackAdtSize(previous_components) == 0) {
    /*
    * We have reached the root. Going 'further back' than the root
    * on Linux just keeps us at the root, so don't do anything here.
    */

   } else {
    /*
    * Pop the previous component and use it.
    */
    current_vnode = StackAdtPop(previous_components);
   }

   continue;
  }

  /*
  * Use a seperate pointer so that both inputs don't point to the same
  * location. vnode_follow either increments the reference count or creates
  * a new vnode with a count of one.
  */
  struct vnode* next_vnode = NULL;
  status = VnodeOpFollow(current_vnode, &next_vnode, component);
  if (status != 0) {
   DereferenceVnode(current_vnode);
   CleanupVnodeStack(previous_components);
   return status;
  }

  /*
  * We have a component that can be backtracked to, so add it to the stack.
  *
  * Also note that vnode_follow adds a reference count, so current_vnode
  * needs to be dereferenced. Conveniently, all components that need to be
  * put on the stack also need dereferencing, and vice versa.
  *
  * The final vnode we find will not be added to the stack and dereferenced
  * as we won't get here.
  */
  StackAdtPush(previous_components, current_vnode);
  current_vnode = next_vnode;
 }

 int status = 0;

 if (want_parent) {
  /*
  * Operations that require us to get the parent don't work if we are already
  * at the root.
  */
  if (StackAdtSize(previous_components) == 0) {
   status = EINVAL;

  } else {
   *out = StackAdtPop(previous_components);
  }
 } else {
  *out = current_vnode;
 }

 CleanupVnodeStack(previous_components);
 return status;
}

int OpenFile(const char* path, int flags, mode_t mode, struct open_file** out) {
    EXACT_IRQL(IRQL_STANDARD);

 if (path == NULL || out == NULL || strlen(path) <= 0) {
  return EINVAL;
 }

    int status;
    char name[MAX_COMPONENT_LENGTH + 1];
 status = GetFinalPathComponent(path, name, MAX_COMPONENT_LENGTH);
 if (status) {
  return status;
 }

    /*
 * Grab the vnode from the path.
 */
 struct vnode* node;

    /*
    * Lookup a (hopefully) existing file.
    */
    status = GetVnodeFromPath(path, &node, false);

    if (flags & O_CREAT) {
  if (status == ENOENT) {
   /*
   * Get the parent folder.
   */
   status = GetVnodeFromPath(path, &node, true);
   if (status) {
    return status;
   }

   struct vnode* child;
   status = VnodeOpCreate(node, &child, name, flags, mode);
   DereferenceVnode(node);

   if (status) {
    return status;
   }

   node = child;

  } else if (flags & O_EXCL) {
   /*
    * The file already exists (as we didn't get ENOENT), but we were passed O_EXCL so we
    * must give an error. If O_EXCL isn't passed, then O_CREAT will just open the existing file.
    */
   return EEXIST;
  }

    } else if (status != 0) {
        return status;
    }

 status = VnodeOpCheckOpen(node, name, flags & (O_ACCMODE | O_NONBLOCK));
  if (status) {
  DereferenceVnode(node);
  return status;
 }
```

```
475
476 bool can_read = (flags & O_ACCMODE) != O_WRONLY;
477 bool can_write = (flags & O_ACCMODE) != O_RDONLY;
478 uint8_t dirent_type = VnodeOpDirentType(node);
479
480 if (dirent_type == DT_DIR && can_write) {
481  /*
482  * You cannot write to a directory. This also prevents truncation.
483  */
484  DereferenceVnode(node);
485  return EISDIR;
486 }
487
488 if ((flags & O_TRUNC) && dirent_type == DT_REG) {
489  if (can_write) {
490   status = VnodeOpTruncate(node, 0);
491   if (status) {
492    return status;
493   }
494   return ENOSYS;
495  } else {
496   return EINVAL;
497  }
498 }
499
500    /* TODO: clear out the flags that don't normally get saved */
501
502 // TODO: may need to actually have a VnodeOpOpen, for things like FatFS.
503
504 *out = CreateOpenFile(node, mode, flags, can_read, can_write);
505    return 0;
506 }
507
508
509 static int FileAccess(struct open_file* file, struct transfer* io, bool write) {
510 EXACT_IRQL(IRQL_STANDARD);
511
512    if (io == NULL || io->address == NULL || file == NULL || file->node == NULL) {
513  return EINVAL;
514 }
515    if ((!write && !file->can_read) || (write && !file->can_write)) {
516        return EBADF;
517    }
518
519 if (file->flags & O_NONBLOCK) {
520  int block_status = VnodeOpWait(file->node, (write ? VNODE_WAIT_WRITE : VNODE_WAIT_READ) | VNODE_WAIT_NON_BLOCK, 0);
521  if (block_status != 0) {
522   return block_status;
523  }
524 }
525
526 if (write) {
527  return VnodeOpWrite(file->node, io);
528 } else {
529  return VnodeOpRead(file->node, io);
530 }
531 }
532
533 int ReadFile(struct open_file* file, struct transfer* io) {
534  return FileAccess(file, io, false);
535 }
536
537 int WriteFile(struct open_file* file, struct transfer* io) {
538  return FileAccess(file, io, true);
539 }
540
541 int CloseFile(struct open_file* file) {
542 EXACT_IRQL(IRQL_STANDARD);
543
544    if (file == NULL || file->node == NULL) {
545  return EINVAL;
546 }
547
548    DereferenceVnode(file->node);
549 DereferenceOpenFile(file);
550 return 0;
551 }
552
553 int GetFileSize(struct open_file* file, off_t* size) {
554 EXACT_IRQL(IRQL_STANDARD);
555
556 if (file == NULL || file->node == NULL || size == NULL) {
557  return EINVAL;
558 }
559
560 struct stat st;
561 int res = VnodeOpStat(file->node, &st);
562 if (res != 0) {
563  return res;
564 }
565
566 *size = st.st_size;
567 return 0;
568 }
```

File: ./.DS_Store

[binary]

File: ./util/log.c

```
1
2  #include <common.h>
3  #include <log.h>
4
5  #define REAL_HW 0
6
7  __attribute__((no_instrument_function)) static void IntToStr(uint32_t i, char* output, int base)
8  {
9   const char* digits = "0123456789ABCDEF";
10
11     /*
12     * Work out where the end of the string is (this is based on the number).
13     * Using the do...while ensures that we always get at least one digit
14     * (i.e. ensures a 0 is printed if the input was 0).
15     */
16  uint32_t shifter = i;
17  do {
18   ++output;
19   shifter /= base;
20  } while (shifter);
21
```

```
    /* Put in the null terminator. */
  *output = '\0';

    /*
    * Now fill in the digits back-to-front.
    */
  do {
   *--output = digits[i % base];
   i /= base;
  } while (i);
}

#if REAL_HW == 0
__attribute__((no_instrument_function)) static void outb(uint16_t port, uint8_t value)
{
 asm volatile ("outb %0, %1" : : "a"(value), "Nd"(port));
}

__attribute__((no_instrument_function)) static uint8_t inb(uint16_t port)
{
 uint8_t value;
 asm volatile ("inb %1, %0"
  : "=a"(value)
  : "Nd"(port));
 return value;
}
#endif

__attribute__((no_instrument_function)) static void logcnv(char c, bool screen)
{
 if (screen) {
  DbgScreenPutchar(c);
 }
#if REAL_HW == 0
 while ((inb(0x3F8 + 5) & 0x20) == 0) {
  ;
 }
 outb(0x3F8, c);
#endif
}

__attribute__((no_instrument_function)) static void logsnv(char* a, bool screen)
{
 while (*a) logcnv(*a++, screen);
}

__attribute__((no_instrument_function)) static void log_intnv(uint32_t i, int base, bool screen)
{
 char str[12];
    IntToStr(i, str, base);
 logsnv(str, screen);
}

__attribute__((no_instrument_function)) static void LogWriteSerialVa(const char* format, va_list list, bool screen) {
 if (format == NULL) {
  format = "NULL";
 }

 int i = 0;

 while (format[i]) {
  if (format[i] == '%') {
   switch (format[++i]) {
   case '%':
    logcnv('%', screen); break;
   case 'c':
    logcnv(va_arg(list, int), screen); break;
   case 's':
    logsnv(va_arg(list, char*), screen); break;
   case 'd':
    log_intnv(va_arg(list, signed), 10, screen); break;
   case 'x':
   case 'X':
    log_intnv(va_arg(list, unsigned), 16, screen); break;
   case 'l':
   case 'L':
    log_intnv(va_arg(list, unsigned long long), 16, screen); break;
   case 'u':
    log_intnv(va_arg(list, unsigned), 10, screen); break;
   default:
    logcnv('%', screen);
    logcnv(format[i], screen);
    break;
   }
  } else {
   logcnv(format[i], screen);
  }
  i++;
 }
}

__attribute__((no_instrument_function)) void LogWriteSerial(const char* format, ...)
{
 va_list list;
 va_start(list, format);
 LogWriteSerialVa(format, list, false);
 va_end(list);
}

__attribute__((no_instrument_function)) void LogDeveloperWarning(const char* format, ...) {
 va_list list;
 va_start(list, format);
 LogWriteSerial("\n!!!!!!!!!!!!!!!!!!!!\n\n>>> KERNEL DEVELOPER WARNING:\n    ");
 LogWriteSerialVa(format, list, false);
 va_end(list);
}

__attribute__((no_instrument_function)) void DbgScreenPrintf(const char* format, ...) {
 va_list list;
 va_start(list, format);
 LogWriteSerialVa(format, list, true);
 va_end(list);
}
```

File: ./util/unicode.c

```

 #include <unicode.h>
 #include <errno.h>

 /*
  * Converts UTF16 to UTF32. out_length is in/out - it must contain the output buffer's maximum length,
  * but will receive the result's length on success. NOT NULL TERMINATED ON INPUT OR OUTPUT!!
  */
 int Utf16ToCodepoints(uint16_t* utf16, uint32_t* codepoints, int in_length, int* out_length) {
     int in = 0;
     int out = 0;

```

```
    while (in < in_length) {
        if (out == *out_length) {
            return ENAMETOOLONG;
        }

        uint32_t codepoint = utf16[in++];
        if (codepoint >= 0xD800 && codepoint <= 0xDBFF) {
            if (in == in_length) {
                return EINVAL;
            }
            uint16_t low_surrogate = utf16[in++];
            if (low_surrogate >= 0xDC00 && low_surrogate <= 0xDFFF) {
                codepoint = (codepoint - 0xD800) * 0x400 + (low_surrogate - 0xDC00);
            } else {
                return EINVAL;
            }

        } else if (codepoint >= 0xDC00 && codepoint <= 0xDFFF) {
            return EINVAL;
        }

        if (codepoint >= 0xD800 && codepoint <= 0xDFFF) {
            return EINVAL;
        }

        codepoints[out++] = codepoint;
    }

    *out_length = out;
    return 0;
}

int Utf8ToCodepoints(uint8_t* utf8, uint32_t* codepoints, int in_length, int* out_length) {
    int in = 0;
    int out = 0;

    while (in < in_length) {
        if (out == *out_length) {
            return ENAMETOOLONG;
        }

        uint32_t codepoint = utf8[in++];
        if ((codepoint >> 3) == 0x1E) {
            if (in + 2 >= in_length) {
                return EINVAL;
            }
            codepoint &= 0x7;
            codepoint <<= 6;
            uint8_t next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;
            codepoint <<= 6;
            next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;
            next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;

        } else if ((codepoint >> 4) == 0xE) {
            if (in + 1 >= in_length) {
                return EINVAL;
            }
            codepoint &= 0xF;
            codepoint <<= 6;
            uint8_t next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;
            codepoint <<= 6;
            next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;

        } else if ((codepoint >> 5) == 0x6) {
            if (in >= in_length) {
                return EINVAL;
            }
            codepoint &= 0x1F;
            codepoint <<= 6;
            uint8_t next = utf8[in++];
            if ((next >> 6) != 0x2) {
                return EINVAL;
            }
            codepoint |= next & 0x3F;


        } else if (codepoint >= 0x80) {
            return EINVAL;
        }

        if (codepoint >= 0xD800 && codepoint <= 0xDFFF) {
            return EINVAL;
        }

        codepoints[out++] = codepoint;
    }

    *out_length = out;
    return 0;
}

int CodepointsToUtf16(uint32_t* codepoints, uint16_t* utf16, int in_length, int* out_length) {
    int in = 0;
    int out = 0;
    while (in < in_length) {
        if (out == *out_length) {
            return ENAMETOOLONG;
        }

        uint32_t codepoint = codepoints[in++];

        if (codepoint >= 0xD800 && codepoint <= 0xDFFF) {
            return EINVAL;
        }
        if (codepoint > 0x10FFFF) {
            return EINVAL;
        }

        if (codepoint >= 0x10000) {
            uint16_t high_surrogate = (codepoint / 0x400) + 0xD800;
```

```
            uint16_t low_surrogate = (codepoint % 0x400) + 0xDC00;
            utf16[out++] = high_surrogate;
            if (out == *out_length) {
                return ENAMETOOLONG;
            }
            utf16[out++] = low_surrogate;

        } else {
            utf16[out++] = codepoint;
        }
    }

    *out_length = out;
    return 0;
}

int CodepointsToUtf8(uint32_t* codepoints, uint8_t* utf8, int in_length, int* out_length) {
    int in = 0;
    int out = 0;
    while (in < in_length) {
        if (out == *out_length) {
            return ENAMETOOLONG;
        }

        uint32_t codepoint = codepoints[in++];

        if (codepoint >= 0xD800 && codepoint <= 0xDFFF) {
            return EINVAL;
        }

        if (codepoint <= 0x7F) {
            utf8[out++] = codepoint;

        } else if (codepoint <= 0x7FF) {
            if (out + 2 > *out_length) {
                return ENAMETOOLONG;
            }
            utf8[out++] = 0xC0 | (codepoint >> 6);
            utf8[out++] = 0x80 | (codepoint & 0x3F);

        } else if (codepoint <= 0xFFFF) {
            if (out + 3 > *out_length) {
                return ENAMETOOLONG;
            }
            utf8[out++] = 0xE0 | (codepoint >> 12);
            utf8[out++] = 0x80 | ((codepoint >> 6) & 0x3F);
            utf8[out++] = 0x80 | (codepoint & 0x3F);

        } else if (codepoint <= 0x10FFF) {
            if (out + 4 > *out_length) {
                return ENAMETOOLONG;
            }
            utf8[out++] = 0xF0 | (codepoint >> 18);
            utf8[out++] = 0x80 | ((codepoint >> 12) & 0x3F);
            utf8[out++] = 0x80 | ((codepoint >> 6) & 0x3F);
            utf8[out++] = 0x80 | (codepoint & 0x3F);
        } else {
            return EINVAL;
        }
    }

    *out_length = out;
    return 0;
}



```

File: ./util/video.c

```
#include <heap.h>
#include <assert.h>
#include <string.h>
#include <irql.h>
#include <video.h>
#include <virtual.h>
#include <errno.h>
#include <log.h>

static struct video_driver video_driver = {
    .putchar = NULL,
    .puts = NULL
};

void DeferPuts(void* v) {
    video_driver.puts((char*) v);
}

void DeferPutchar(void* v) {
    video_driver.putchar((char) (size_t) v);
}

void DbgScreenPutchar(char c) {
    if (video_driver.putchar != NULL) {
        DeferUntilIrql(IRQL_STANDARD, DeferPutchar, (void*) (size_t) c);
    }
}

void DbgScreenPuts(char* s) {
    if (video_driver.puts != NULL) {
        DeferUntilIrql(IRQL_STANDARD, DeferPuts, s);
    }
}

void InitVideoConsole(struct video_driver driver) {
    video_driver = driver;
}
```

File: ./util/panic.c

```
1
2  #include <panic.h>
3  #include <arch.h>
4  #include <log.h>
5  #include <errno.h>
6  #include <debug.h>
7  #include <irql.h>
8
9  static const char* message_table[_PANIC_HIGHEST_VALUE] = {
10  [PANIC_UNKNOWN]         = "unknown",
11  [PANIC_IMPOSSIBLE_RETURN]       = "impossible return",
12  [PANIC_MANUALLY_INITIATED]      = "manually initiated",
13  [PANIC_UNIT_TEST_OK]        = "unit test ok",
14  [PANIC_DRIVER_FAULT]        = "driver fault",
15  [PANIC_OUT_OF_HEAP]         = "out of heap",
16  [PANIC_OUT_OF_BOOTSTRAP_HEAP]     = "out of boostrap heap",
17  [PANIC_HEAP_REQUEST_TOO_LARGE]    = "heap request too large",
18  [PANIC_ASSERTION_FAILURE]       = "assertion failure",
19  [PANIC_NO_MEMORY_MAP]       = "no memory map",
20  [PANIC_NOT_IMPLEMENTED]       = "not implemented",
21  [PANIC_INVALID_IRQL]        = "invalid irq level",
22  [PANIC_SPINLOCK_WRONG_IRQL]     = "spinlock wrong irql",
23  [PANIC_PRIORITY_QUEUE]      = "invalid operation on a priority queue",
24  [PANIC_OUT_OF_PHYS]         = "no physical memory left",
25  [PANIC_LINKED_LIST]         = "invalid operation on a linked list",
26  [PANIC_CANARY_DIED]         = "kernel stack overflow detected",
27  [PANIC_SEMAPHORE_DESTROY_WHILE_HELD]   = "tried to destroy a held semaphore",
28  [PANIC_SEM_BLOCK_WITHOUT_THREAD]    = "semaphore acquisition would block before multithreading has started",
29  [PANIC_CANNOT_LOCK_MEMORY]      = "cannot lock virtual memory",
30  [PANIC_THREAD_LIST]         = "invalid operation on a thread list",
31  [PANIC_CANNOT_MALLOC_WITHOUT_FAULTING]  = "heap allocation cannot be completed without faulting",
32  [PANIC_NO_FILESYSTEM]       = "no driver can access the boot filesystem",
33  [PANIC_BAD_KERNEL]          = "kernel executable file is corrupted",
34  [PANIC_DISK_FAILURE_ON_SWAPFILE]    = "failed to read from or write to swapfile",
35  [PANIC_NEGATIVE_SEMAPHORE]      = "more semaphore releases than acquisitions",
36  [PANIC_NON_MASKABLE_INTERRUPT]    = "unrecoverable hardware error",
37  [PANIC_UNHANDLED_KERNEL_EXCEPTION]    = "unhandled cpu exception",
38  [PANIC_REQUIRED_DRIVER_MISSING_SYMBOL]  = "driver is missing a required symbol",
39  [PANIC_REQUIRED_DRIVER_NOT_FOUND]     = "a required driver could not be loaded",
40  [PANIC_NO_LOW_MEMORY]       = "not enough conventional memory to satisfy request",
41  [PANIC_OUT_OF_SWAPFILE]       = "out of swapfile space",
42  [PANIC_PROGRAM_LOADER]      = "failed to load the program loader",
43  [PANIC_VAS_TRIED_TO_SELF_DESTRUCT]    = "tried to destroy the current virtual address space",
44  [PANIC_ACPI_AML]            = "acpi aml is invalid",
45  [PANIC_SPINLOCK_DOUBLE_ACQUISITION]   = "spinlock attempted to be locked while currently held",
46  [PANIC_SPINLOCK_RELEASED_BEFORE_ACQUIRED] = "spinlock attempted to be released while not held",
47  [PANIC_DOUBLE_FREE_DETECTED]      = "heap memory freed twice",
48  [PANIC_CONFLICTING_ALLOCATION_REQUIREMENTS] = "conflicting heap allocation requirements"
49 };
50
51 static void (*graphical_panic_handler)(int, const char*) = NULL;
52
53 int SetGraphicalPanicHandler(void (*handler)(int, const char*)) {
54  if (graphical_panic_handler == NULL) {
55   graphical_panic_handler = handler;
56   return 0;
57
58  } else {
59   return EALREADY;
60  }
61 }
62
63 const char* GetPanicMessageFromCode(int code) {
64  return code < _PANIC_HIGHEST_VALUE ? message_table[code] : "";
65 }
66
67 [[noreturn]] void Panic(int code)
68 {
69  PanicEx(code, GetPanicMessageFromCode(code));
70 }
71
72 [[noreturn]] void PanicEx(int code, const char* message) {
73  LogWriteSerial("PANIC %d %s\n", code, message);
74  if (IsInTfwTest()) {
75   LogWriteSerial("in test.\n");
76   FinishedTfwTest(code);
77   ArchSetPowerState(ARCH_POWER_STATE_REBOOT);
78  }
79
80  RaiseIrql(IRQL_HIGH);
81  LogWriteSerial("\n\n *** KERNEL PANIC ***\n\n0x%X - %s\n", code, message);
82
83  if (graphical_panic_handler != NULL) {
84   graphical_panic_handler(code, message);
85  }
86
87  while (1) {
88   ArchDisableInterrupts();
89   ArchStallProcessor();
90  }
91 }
```

File: ./util/assert.c

```
1  #include <assert.h>
2  #include <panic.h>
3  #include <arch.h>
4  #include <log.h>
5  #include <debug.h>
6
7  _Noreturn void AssertionFail(const char* file, const char* line, const char* condition, const char* msg) {
8   ArchDisableInterrupts();
9   LogWriteSerial("Assertion failed: %s %s [%s: %s]\n", condition, msg, file, line);
10  Panic(PANIC_ASSERTION_FAILURE);
11 }
```

File: ./util/console.c

```

#include <console.h>
#include <pty.h>
#include <vfs.h>
#include <thread.h>
#include <virtual.h>
#include <string.h>
#include <log.h>

static struct vnode* console_master;
static struct vnode* console_subordinate;

static struct open_file* open_console_master;
static struct open_file* open_console_subordinate;

static bool console_initialised = false;

/*
 * NOTE: the console only echos input when there's someone waiting for input
 *   (which should be fine most of the time - the only people input is when it's waiting for input!)
 */

static void ConsoleDriverThread(void*) {
 AddVfsMount(console_subordinate, "con");

    while (true) {
        char c;
        struct transfer tr = CreateKernelTransfer(&c, 1, 0, TRANSFER_READ);
  ReadFile(open_console_master, &tr);
  DbgScreenPutchar(c);
    }
}

void InitConsole(void) {
 CreatePseudoTerminal(&console_master, &console_subordinate);
 open_console_master = CreateOpenFile(console_master, 0, 0, true, true);
 open_console_subordinate = CreateOpenFile(console_subordinate, 0, 0, true, true);
   CreateThread(ConsoleDriverThread, NULL, GetVas(), "con");
 console_initialised = true;
}

void SendKeystrokeConsole(char c) {
 if (!console_initialised) return;

 struct transfer tr = CreateKernelTransfer(&c, 1, 0, TRANSFER_WRITE);
 WriteFile(open_console_master, &tr);
}

char GetcharConsole(void) {
 if (!console_initialised) return 0;

 char c;
 struct transfer tr = CreateKernelTransfer(&c, 1, 0, TRANSFER_READ);
 ReadFile(open_console_subordinate, &tr);
 return c;
}

void PutcharConsole(char c) {
 if (!console_initialised) return;
 struct transfer tr = CreateKernelTransfer(&c, 1, 0, TRANSFER_WRITE);
 WriteFile(open_console_subordinate, &tr);
}

void PutsConsole(const char* s) {
 for (int i = 0; s[i]; ++i) {
  PutcharConsole(s[i]);
 }
}
```

File: ./util/driver.c

```

 #include <arch.h>
 #include <driver.h>
 #include <semaphore.h>
 #include <irql.h>
 #include <avl.h>
 #include <string.h>
 #include <vfs.h>
 #include <errno.h>
 #include <heap.h>
 #include <panic.h>
 #include <log.h>
 #include <stdlib.h>
 #include <virtual.h>
 #include <fcntl.h>
 #include <ctype.h>
 #include <assert.h>

 static struct semaphore* driver_table_lock;
 static struct semaphore* symbol_table_lock;
 static struct avl_tree* loaded_drivers;
 static struct avl_tree* symbol_table;

 struct symbol {
     const char* name;
     size_t addr;
 };

 struct loaded_driver {
     char* filename;
     size_t relocation_point;
     struct quick_relocation_table* quick_relocation_table;
 };

 static int SymbolComparator(void* a_, void* b_) {
     struct symbol* a = a_;
     struct symbol* b = b_;
     return strcmp(a->name, b->name);
 }

 static int DriverTableComparatorByRelocationPoint(void* a_, void* b_) {
     struct loaded_driver* a = a_;
     struct loaded_driver* b = b_;
     return COMPARE_SIGN(a->relocation_point, b->relocation_point);
 }

 static int DriverTableComparatorByName(void* a_, void* b_) {
     struct loaded_driver* a = a_;
     struct loaded_driver* b = b_;
     return strcmp(a->filename, b->filename);
 }

 static int QuickRelocationTableComparator(const void* a_, const void* b_) {
  struct quick_relocation a = *((struct quick_relocation*) a_);
  struct quick_relocation b = *((struct quick_relocation*) b_);
     return COMPARE_SIGN(a.address, b.address);
 }
```

```
57
58 static size_t GetDriverAddressWithLockHeld(const char* name) {
59     struct loaded_driver dummy;
60     dummy.filename = (char*) name;
61
62     AvlTreeSetComparator(loaded_drivers, DriverTableComparatorByName);
63     struct loaded_driver* res = AvlTreeGet(loaded_drivers, &dummy);
64     if (res == NULL) {
65         return 0;
66     }
67     return res->relocation_point;
68 }
69
70 static struct loaded_driver* GetDriverFromAddress(size_t relocation_point) {
71     AcquireMutex(driver_table_lock, -1);
72
73     struct loaded_driver dummy;
74     dummy.relocation_point = relocation_point;
75
76     AvlTreeSetComparator(loaded_drivers, DriverTableComparatorByRelocationPoint);
77     struct loaded_driver* res = AvlTreeGet(loaded_drivers, &dummy);
78     ReleaseMutex(driver_table_lock);
79     return res;
80 }
81
82 size_t GetDriverAddress(const char* name) {
83     EXACT_IRQL(IRQL_STANDARD);
84
85     AcquireMutex(driver_table_lock, -1);
86     size_t res = GetDriverAddressWithLockHeld(name);
87     ReleaseMutex(driver_table_lock);
88     return res;
89 }
90
91 void InitSymbolTable(void) {
92     driver_table_lock = CreateMutex("drv table");
93     symbol_table_lock = CreateMutex("sym table");
94
95     loaded_drivers = AvlTreeCreate();
96     symbol_table = AvlTreeCreate();
97     AvlTreeSetComparator(symbol_table, SymbolComparator);
98
99     struct open_file* kernel_file;
100     if (OpenFile("sys:/kernel.exe", O_RDONLY, 0, &kernel_file)) {
101         Panic(PANIC_NO_FILESYSTEM);
102     }
103     ArchLoadSymbols(kernel_file, 0);
104     CloseFile(kernel_file);
105 }
106
107 static bool DoesSymbolContainIllegalCharacters(const char* symbol) {
108     for (int i = 0; symbol[i]; ++i) {
109         if (!isalnum(symbol[i]) && symbol[i] != '_') {
110             return true;
111         }
112     }
113     return strlen(symbol) == 0;
114 }
115
116 void AddSymbol(const char* symbol, size_t address) {
117     EXACT_IRQL(IRQL_STANDARD);
118
119     if (DoesSymbolContainIllegalCharacters(symbol)) {
120         return;
121     }
122
123     struct symbol* entry = AllocHeap(sizeof(struct symbol));
124     entry->name = strdup(symbol);
125     entry->addr = address;
126
127     AcquireMutex(symbol_table_lock, -1);
128     if (AvlTreeContains(symbol_table, entry)) {
129         /*
130          * The kernel has some symbols declared 'static' to file scope, with
131          * duplicate names (e.g. in /dev each file has its own 'Stat'). These
132          * get exported for some reason so we end up with duplicate names. We
133          * must ignore these to avoid AVL issues. They are safe to ignore, as
134          * they were meant to be 'static' anyway.
135          *
136          * TODO: there may be issues if device drivers try to create their own
137          * methods with those names (?) e.g. they use the standard template
138          * and have their own 'Stat'.
139          */
140         FreeHeap(entry);
141     } else {
142         AvlTreeInsert(symbol_table, entry);
143     }
144     ReleaseMutex(symbol_table_lock);
145 }
146
147 /*
148  * Do not name a (global) function pointer you receive from this the same as the
149  * actual function - this may cause issues with duplicate symbols.
150  *
151  * e.g. don't do this at global scope:
152  *
153  * void (*MyFunc)(void) = GetSymbolAddress("MyFunc");
154  *
155  * Instead, do void (*_MyFunc)(void) or something.
156  */
157 size_t GetSymbolAddress(const char* symbol) {
158     EXACT_IRQL(IRQL_STANDARD);
159
160     struct symbol dummy;
161     dummy.name = symbol;
162
163     AcquireMutex(symbol_table_lock, -1);
164     struct symbol* result = AvlTreeGet(symbol_table, &dummy);
165     ReleaseMutex(symbol_table_lock);
166
167     if (result == NULL) {
168         return 0;
169
170     } else {
171         assert(!strcmp(result->name, symbol));
172         return result->addr;
173     }
174 }
175
176 static int LoadDriver(const char* name) {
177     struct open_file* file;
178     int res;
179     if ((res = OpenFile(name, O_RDONLY, 0, &file))) {
180         return res;
181     }
182
183     struct loaded_driver* drv = AllocHeap(sizeof(struct loaded_driver));
184     drv->filename = strdup_pageable(name);
185     drv->quick_relocation_table = NULL;
186     if ((res = ArchLoadDriver(&drv->relocation_point, file, &drv->quick_relocation_table))) {
```

```
187         return res;
188     }
189 
190     assert(drv->quick_relocation_table != NULL);
191 
192     AvlTreeInsert(loaded_drivers, drv);
193     ArchLoadSymbols(file, drv->relocation_point - 0xD0000000); // TODO: @@@ GET RID OF ARCH SPECIFIC DETAILS (0xD0000000)
194     return 0;
195 }
196 
197 int RequireDriver(const char* name) {
198     EXACT_IRQL(IRQL_STANDARD);
199 
200     LogWriteSerial("Requiring driver: %s\n", name);
201 
202     AcquireMutex(driver_table_lock, -1);
203 
204     if (GetDriverAddressWithLockHeld(name) != 0) {
205         ReleaseMutex(driver_table_lock);
206 
207         /*
208          * Not an error that it's already loaded - ideally no one should care if it has already been loaded
209          * or not. Hence we give 0 (success), not EALREADY.
210          */
211         return 0;
212     }
213 
214     int res = LoadDriver(name);
215     ReleaseMutex(driver_table_lock);
216     return res;
217 }
218 
219 void SortQuickRelocationTable(struct quick_relocation_table* table) {
220  struct quick_relocation* entries = table->entries;
221  qsort_pageable((void*) entries, table->used_entries, sizeof(struct quick_relocation), QuickRelocationTableComparator);
222 }
223 
224 void AddToQuickRelocationTable(struct quick_relocation_table* table, size_t addr, size_t val) {
225  assert(table->used_entries < table->total_entries);
226  table->entries[table->used_entries].address = addr;
227  table->entries[table->used_entries].value = val;
228  table->used_entries++;
229 }
230 
231 struct quick_relocation_table* CreateQuickRelocationTable(int count) {
232     struct quick_relocation_table* table = AllocHeap(sizeof(struct quick_relocation_table));
233     struct quick_relocation* entries = (struct quick_relocation*) MapVirt(0, 0, count * sizeof(struct quick_relocation), VM_READ | VM_WRITE, NULL, 0);
234     table->entries = entries;
235     table->used_entries = 0;
236     table->total_entries = count;
237     return table;
238 }
239 
240 static int BinarySearchComparator(const void* a_, const void* b_) {
241     struct quick_relocation a = *((struct quick_relocation*) a_);
242  struct quick_relocation b = *((struct quick_relocation*) b_);
243 
244     size_t page_a = a.address / ARCH_PAGE_SIZE;
245     size_t page_b = b.address / ARCH_PAGE_SIZE;
246 
247     if (page_a > page_b) {
248         return 1;
249 
250     } else if (page_a < page_b) {
251         return -1;
252 
253     } else {
254         return 0;
255     }
256 }
257 
258 static void ApplyRelocationsToPage(struct quick_relocation_table* table, size_t virtual) {
259     struct quick_relocation target;
260     target.address = virtual;
261     LogWriteSerial("ApplyRelocationsToPage: looking for 0x%X\n", virtual);
262 
263     struct quick_relocation* entry = bsearch(&target, table->entries, table->used_entries, sizeof(struct quick_relocation), BinarySearchComparator);
264     if (entry == NULL) {
265         PanicEx(PANIC_ASSERTION_FAILURE, "quick relocation table doesn't contain lookup - bsearch or qsort is probably bugged");
266     }
267 
268     /*
269      * As we only did a binary search to look for anything on that page, we might be halfway through the page!
270      * Move back through the entries until we find the start of page (or the first entry in the table).
271      */
272     while ((entry->address / ARCH_PAGE_SIZE == virtual / ARCH_PAGE_SIZE
273         || (entry->address - sizeof(size_t) + 1) == virtual / ARCH_PAGE_SIZE) && entry != table->entries) {
274         entry -= 1;
275     }
276 
277     /*
278      * We went past the last one, so need to forward onto it again - unless we hit the start of the table.
279      */
280     if (entry != table->entries) {
281         entry += 1;
282     }
283 
284 
285     /*
286      * We also need to lock these, as (and YES this has actually happened before):
287      *   - if the relocation is on the boundary of the next one, and the next one is not present
288      *   - and the next one is relocatable
289      * Then:
290      *   - we make the next page writable
291      *   - we try to do the straddle relocation
292      *   - that causes a fault on the next page (not present)
293      *   - that one is also relocatable, so it enables writing on that page
294      *   - it finishes, and unmarks it as writable
295      *   - we fail to do the relocation, as it is no longer writable
296      *
297      * By locking it first, we force the relocations on the second page to happen first, and then we can
298      * mark it as writable.
299      */
300 
301     bool needs_write_low = (GetVirtPermissions(virtual) & VM_WRITE) == 0;
302     bool needs_write_high = false;
303     bool need_unlock_high = false;
304 
305  if (needs_write_low) {
306      SetVirtPermissions(virtual, VM_WRITE, 0);
307  }
308 
309     size_t final_address = table->entries[table->used_entries - 1].address;
310 
311     while (entry->address / ARCH_PAGE_SIZE == virtual / ARCH_PAGE_SIZE
312         || (entry->address - sizeof(size_t) + 1) == virtual / ARCH_PAGE_SIZE) {
313 
314         if ((entry->address + sizeof(size_t) + 1) / ARCH_PAGE_SIZE != virtual / ARCH_PAGE_SIZE) {
315             need_unlock_high = !LockVirt(virtual + ARCH_PAGE_SIZE);
316             needs_write_high = (GetVirtPermissions(virtual + ARCH_PAGE_SIZE) & VM_WRITE) == 0;
```

```
            if (needs_write_high) {
                SetVirtPermissions(virtual + ARCH_PAGE_SIZE, VM_WRITE, 0);
            }
        }

        size_t* ref = (size_t*) entry->address;
        *ref = entry->value;

        if (entry->address == final_address) {
            break;
        }
        entry += 1;
    }

    if (needs_write_low) {
  SetVirtPermissions(virtual, 0, VM_WRITE);
 }
 if (needs_write_high) {
  SetVirtPermissions(virtual + ARCH_PAGE_SIZE, 0, VM_WRITE);
 }
    if (need_unlock_high) {
        UnlockVirt(virtual + ARCH_PAGE_SIZE);
    }
}

void PerformDriverRelocationOnPage(struct vas*, size_t relocation_base, size_t virt) {
    LogWriteSerial("PerformDriverRelocationOnPage A\n");
    struct loaded_driver* drv = GetDriverFromAddress(relocation_base);
    if (drv == NULL) {
        PanicEx(PANIC_ASSERTION_FAILURE, "PerformDriverRelocationOnPage");
    }
    LogWriteSerial("PerformDriverRelocationOnPage B. driver at 0x%X\n", drv);

    ApplyRelocationsToPage(drv->quick_relocation_table, virt);
}
```

File: ./Makefile

```
CC = /Users/alex/Desktop/NOS/toolchain/output/bin/i386-elf-gcc
AS = nasm
FAKE_CROSS_COMPILER = -m32 -I"." -Iinclude -Imachine/include -Iinclude/openlibm
COMPILE_FLAGS = -c -Os -fipa-icf -std=gnu2x -ffunction-sections -fno-strict-aliasing -DCOMPILE_KERNEL -Wall -Wextra -Wpedantic -Werror -Wcast-align=strict -Wpointer-arith -fmax-errors=5 -ffreestanding $(FAKE_CROSS_COMPILER) -Wno-infinite-recursion -fomit-frame-pointer
#-flto -finstrument-functions

LINK_FLAGS = -fuse-ld=gold -Wl,--icf=all -Wl,-Map=kernel.map -nostartfiles -nostdlib -lgcc

# Set by the higher level Makefile before calling us - changes depending on whether we are compiling the debug or release build
LINKER_STRIP =
CPPDEFINES =

COBJECTS = $(patsubst %.c, %.o, $(wildcard *.c) $(wildcard */*.c) $(wildcard */*/*.c) $(wildcard */*/*/*.c) $(wildcard */*/*/*/*.c) $(wildcard **/*.c))
ASMOBJECTS = $(patsubst %.s, %.oo, $(wildcard *.s) $(wildcard */*.s) $(wildcard */*/*.s) $(wildcard */*/*/*.s) $(wildcard **/*.s))

oskernel: $(COBJECTS) $(ASMOBJECTS) $(HOBJECTS)
$(CC) -T machine/linker.ld -o KERNEL.EXE $^ $(LINK_FLAGS) $(LINKER_STRIP)
# rm -r include
# rm -r machine
rm Makefile

%.o: %.c
$(CC) $(CPPDEFINES) $(COMPILE_FLAGS) $^ -o $@
rm $^

%.oo: %.s
$(AS) -felf32 $^ -o $@
rm $^
```

File: ./include/filedes.h

```
#pragma once

#include <common.h>

#define MAX_FD_PER_PROCESS 1024

struct open_file;
struct filedes_table;

int CreateFileDescriptor(struct filedes_table* table, struct open_file* file, int* fd_out, int flags);
int RemoveFileDescriptor(struct filedes_table* table, struct open_file* file);
int GetFileFromDescriptor(struct filedes_table* table, int fd, struct open_file** out);

int HandleFileDescriptorsOnExec(struct filedes_table* table);

struct filedes_table* CreateFileDescriptorTable(void);
struct filedes_table* CopyFileDescriptorTable(struct filedes_table* original);
int DuplicateFileDescriptor(struct filedes_table* table, int oldfd, int* newfd);
int DuplicateFileDescriptor2(struct filedes_table* table, int oldfd, int newfd, int flags);
```

File: ./include/filesystem.h

```
#pragma once

#include <common.h>

struct open_file;

typedef int(*fs_mount_creator)(struct open_file*, struct open_file**);

void InitFilesystemTable(void);
int RegisterFilesystem(char* fs_name, fs_mount_creator mount);
int MountFilesystemForDisk(struct open_file* partition);
```

File: ./include/semaphore.h

```
#pragma once

#include <common.h>

struct semaphore;
struct thread;

#define SEM_BIG_NUMBER (1 << 30)

#define SEM_DONT_CARE 0
#define SEM_REQUIRE_ZERO 1
#define SEM_REQUIRE_FULL 2

struct semaphore* CreateSemaphore(const char* name, int max_count, int initial_count);
int AcquireSemaphore(struct semaphore* sem, int timeout_ms);
void ReleaseSemaphore(struct semaphore* sem);
int DestroySemaphore(struct semaphore* sem, int mode);
int GetSemaphoreCount(struct semaphore* sem);

#define CreateMutex(name) CreateSemaphore(name, 1, 0)
#define AcquireMutex(mtx, timeout_ms) AcquireSemaphore(mtx, timeout_ms)
#define ReleaseMutex(mtx) ReleaseSemaphore(mtx)
#define DestroyMutex(mtx, mode) DestroySemaphore(mtx, mode)

void CancelSemaphoreOfThread(struct thread* thr);
```

File: ./include/console.h

```
#pragma once

#include <common.h>

void InitConsole(void);
void SendKeystrokeConsole(char c);
char GetcharConsole(void);
void PutcharConsole(char c);
void PutsConsole(const char* s);
```

File: ./include/stackadt.h

```
#pragma once

#include <common.h>

struct stack_adt;

struct stack_adt* StackAdtCreate(void);
void StackAdtDestroy(struct stack_adt* stack);
void StackAdtPush(struct stack_adt* stack, void* data);
void* StackAdtPeek(struct stack_adt* stack);
void* StackAdtPop(struct stack_adt* stack);
int StackAdtSize(struct stack_adt* stack);
```

File: ./include/spinlock.h

```
#pragma once

#include <common.h>

struct thread;

struct spinlock {
size_t lock;
struct thread* owner;
char name[16];
int irql;
int prev_irql;
};

void InitSpinlock(struct spinlock* lock, const char* name, int irql);

int AcquireSpinlockIrql(struct spinlock* lock);
void ReleaseSpinlockIrql(struct spinlock* lock);

void AcquireSpinlockDirect(struct spinlock* lock);
void ReleaseSpinlockDirect(struct spinlock* lock);

bool IsSpinlockHeld(struct spinlock* lock);
```

File: ./include/pty.h

```
#pragma once

#include <common.h>

struct vnode;

void CreatePseudoTerminal(struct vnode** master, struct vnode** subordinate);
```

File: ./include/debug.h

```
#include <debug/tfw.h>
```

File: ./include/progload.h

```
#pragma once

#include <common.h>
```

```
void InitProgramLoader(void);
int CopyProgramLoaderIntoAddressSpace(void);
```

File: ./include/irql.h

```
#pragma once

#include <assert.h>

struct irql_deferment {
void (*handler)(void*);
void* context;
};

/**
* SIMPLE TABLE
*
* Can page fault? Can task switch? Can use drivers? Can have IRQs?
* IRQL_STANDARD YES YES YES YES
* IRQL_PAGE_FAULT SORT OF YES YES YES (only the page fault handler can generate a nested page fault, e.g. handling some COW stuff)
* IRQL_SCHEDULER NO SORT OF YES YES (only the scheduler can make a task switch occur, others get postponed)
* IRQL_DRIVER NO NO SORT OF YES (only higher priority drivers can be used)
* IRQL_TIMER NO NO NO NO (but the timer handler jumps up to this level)
* IRQL_HIGH NO NO NO NO
*
*/

/*
* Scheduler works. Page faults are allowed.
*/
#define IRQL_STANDARD 0
#define IRQL_STANDARD_HIGH_PRIORITY 1

/*
* Scheduler still works at this point. Cannot page fault.
*/
#define IRQL_PAGE_FAULT 2

/*
* This is the scheduler (and therefore things won't be scheduled out 'behind its back'). Cannot page fault.
*/
#define IRQL_SCHEDULER 3

/*
* Scheduling will be postponed. Cannot page fault. Cannot use lower-priority devices.
*/
#define IRQL_DRIVER 4 // 3...39 is the driver range

/*
* No scheduling, no page faulting, no using other hardware devices (no other irqs)
*/
#define IRQL_TIMER 40

/*
* No interrupts from here.
*/
#define IRQL_HIGH 41

#include <common.h>
#include <assert.h>

void PostponeScheduleUntilStandardIrql(void);
void DeferUntilIrql(int irql, void(*handler)(void*), void* context);
int GetIrql(void);
int RaiseIrql(int level);
void LowerIrql(int level);
int GetNumberInDeferQueue(void);

void InitIrql(void);

#define MAX_IRQL(l) assert(GetIrql() <= l)
#define MIN_IRQL(l) assert(GetIrql() >= l)
#define EXACT_IRQL(l) assert(GetIrql() == l)
```

File: ./include/vnode.h

```
#pragma once

#include <common.h>
#include <sys/types.h>
#include <transfer.h>
#include <spinlock.h>
#include <sys/stat.h>

struct vnode;

/*
* Operations which can be performed on an abstract file. They may be left NULL,
* in this case, a default return value is supplied.
*
* check_open: default 0
* Called just before a file is opened to ensure that the flags and the filename
* are valid. Flags that can be passed in are O_RDONLY, O_WRONLY and O_RDWR, and
* O_NONBLOCK. A filename may be invalid if the name is too long for the filesystem,
* or if the filesystem contains other reserved characters.
*
* read: default EINVAL
* Reads data from the file. If the file gives DT_DIR when asked for
* dirent_type, then it should read in chunks of sizeof(struct dirent),
* with the last being full of null bytes.
*
* write: default EINVAL
* Writes data to the file. Fails on directories (EISDIR).
*
```

```
* ioctl: default EINVAL
* Performs a miscellaneous operation on a file.
*
* is_seekable: default EINVAL
* Returns true if seek can be called on the file.
*
* check_tty: default ENOTTY
* Returns 0 if a terminal, or ENOTTY otherwise.
*
* close: default 0
* Frees the vnode, as its reference count has hit zero.
*
* truncate: default EINVAL
* Truncates the file to the given size. Fails on directories (EISDIR).
*
* create: default EINVAL
* Creates a new file under a given parent, with a given name.
* The flags specifies O_RDWR, O_RDONLY, O_WRONLY, O_EXCL and O_APPEND.
*
* follow: default ENOTDIR
* Returns the vnode associated with a child of the current vnode.
* Fails on files (ENOTDIR).
*
* dirent_type: default DT_UNKNOWN
* Returns the type of file, either DT_DIR, DT_REG, DT_BLK, DT_CHR, DT_FIFO, DT_LNK, or DT_UNKNOWN.
*
*/

#define VNODE_WAIT_READ (1 << 0)
#define VNODE_WAIT_WRITE (1 << 1)
#define VNODE_WAIT_ERROR (1 << 2)
#define VNODE_WAIT_HAVE_TIMEOUT (1 << 3)
#define VNODE_WAIT_NON_BLOCK (1 << 4)

struct vnode_operations {
int (*check_open)(struct vnode* node, const char* name, int flags);
int (*read)(struct vnode* node, struct transfer* io);
int (*write)(struct vnode* node, struct transfer* io);
int (*ioctl)(struct vnode* node, int command, void* buffer);
int (*close)(struct vnode* node); // release the fileystem specific data
int (*truncate)(struct vnode* node, off_t offset);
int (*create)(struct vnode* node, struct vnode** out, const char* name, int flags, mode_t mode);
int (*follow)(struct vnode* node, struct vnode** out, const char* name);
int (*stat)(struct vnode* node, struct stat* st);
int (*check_tty)(struct vnode* node);
int (*wait)(struct vnode* node, int flags, uint64_t timeout_ms);

bool (*is_seekable)(struct vnode* node);
uint8_t (*dirent_type)(struct vnode* node);
};

struct vnode {
struct vnode_operations ops;
void* data;
int reference_count;
struct spinlock reference_count_lock;
};


/*
* Allocates a new vnode for a given set of operations.
*/
struct vnode* CreateVnode(struct vnode_operations ops);
void ReferenceVnode(struct vnode* node);
void DereferenceVnode(struct vnode* node);

/*
* Wrapper functions to check the vnode is valid, and then call the driver.
*/
int VnodeOpCheckOpen(struct vnode* node, const char* name, int flags);
int VnodeOpRead(struct vnode* node, struct transfer* io);
int VnodeOpWrite(struct vnode* node, struct transfer* io);
int VnodeOpIoctl(struct vnode* node, int command, void* buffer);
bool VnodeOpIsSeekable(struct vnode* node);
int VnodeOpCheckTty(struct vnode* node);
int VnodeOpClose(struct vnode* node);
int VnodeOpTruncate(struct vnode* node, off_t offset);
uint8_t VnodeOpDirentType(struct vnode* node);
int VnodeOpCreate(struct vnode* node, struct vnode** out, const char* name, int flags, mode_t mode);
int VnodeOpFollow(struct vnode* node, struct vnode** out, const char* name);
int VnodeOpStat(struct vnode* node, struct stat* st);
int VnodeOpWait(struct vnode* node, int flags, uint64_t timeout_ms);
```

File: ./include/blockingbuffer.h

```
#pragma once

#include <common.h>

struct blocking_buffer;

struct blocking_buffer* BlockingBufferCreate(int size);
void BlockingBufferDestroy(struct blocking_buffer* buffer);
int BlockingBufferAdd(struct blocking_buffer* buffer, uint8_t c, bool block);
uint8_t BlockingBufferGet(struct blocking_buffer* buffer);
int BlockingBufferTryGet(struct blocking_buffer* buffer, uint8_t* c);
```

File: ./include/virtual.h

```
#pragma once

#include <stddef.h>
#include <sys/types.h>
#include <stdbool.h>
#include <common.h>
```

```
#define VM_READ 1
#define VM_WRITE 2
#define VM_USER 4
#define VM_EXEC 8
#define VM_LOCK 16
#define VM_FILE 32
#define VM_FIXED_VIRT 64
#define VM_MAP_HARDWARE 128 /* map a physical page that doesn't live within the physical memoery manager*/
#define VM_LOCAL 256 /* indicates it's local to the VAS - i.e. not in kernel global memory */
#define VM_RECURSIVE 512 /* assumes the VAS is already locked, so won't lock or unlock it */
#define VM_RELOCATABLE 1024 /* needs driver fixups whenever swapped back in*/
#define VM_EVICT_FIRST 2048

#define VAS_NO_ARCH_INIT 1

struct open_file;

struct vas_entry {
size_t virtual;

uint8_t in_ram : 1; /* Whether it is backed by a physical page or not. (i.e. does it have a real page table entry) */
uint8_t allocated : 1; /* Whether or not to free a physical page on deallocation. Differs from in_ram when VM_MAP_HARDWARE is set. */
uint8_t file : 1; /* Whether or not the page is file-mapped. */
uint8_t cow : 1; /* */
uint8_t swapfile : 1; /* Whether or not the page has been moved to a swapfile. Will not occur if 'file' is set (will back to that file instead)*/
uint8_t lock : 1;
uint8_t read : 1;
uint8_t write : 1;

uint8_t exec : 1;
uint8_t user : 1;
uint8_t global : 1;
uint8_t allow_temp_write: 1; /* used internally - allows the system to write to otherwise read-only pages to, e.g. reload from disk */
uint8_t relocatable : 1; /* from a relocated driver file */
uint8_t first_load : 1;
uint8_t load_in_progress: 1; /* someone else is deferring a read into this page - keep trying the access until flag clears */

uint8_t times_swapped : 4;
uint8_t evict_first : 1;
uint8_t : 3;

int num_pages; /* only used for non-allocated or hardware mapped to reduce the number of AVL entries */

off_t file_offset;
struct open_file* file_node;
size_t physical;
union {
size_t swapfile_offset;
size_t relocation_base;
};

int ref_count;
};

struct vas;

size_t BytesToPages(size_t bytes);

bool LockVirt(size_t virtual);
void UnlockVirt(size_t virtual);
bool LockVirtEx(struct vas* vas, size_t virtual);
void UnlockVirtEx(struct vas* vas, size_t virtual);

void SetVirtPermissions(size_t virtual, int set, int clear);
int GetVirtPermissions(size_t virtual);
size_t MapVirt(size_t physical, size_t virtual, size_t bytes, int flags, struct open_file* file, off_t pos);
int UnmapVirt(size_t virtual, size_t bytes);
int UnmapVirtEx(struct vas* vas, size_t virtual, size_t pages);
size_t GetPhysFromVirt(size_t virtual);

struct vas* GetKernelVas(void); // a kernel vas
struct vas* GetVas(void); // current vas

struct vas* CreateVas(void);
void CreateVasEx(struct vas* vas, int flags);
void DestroyVas(struct vas* vas);

struct vas* CopyVas(void);
void SetVas(struct vas* vas);
void InitVirt(void);
bool IsVirtInitialised(void);
void EvictVirt(void);
void HandleVirtFault(size_t faulting_virt, int fault_type);

#include <arch.h>
#include <spinlock.h>

struct vas {
struct avl_tree* mappings;
platform_vas_data_t* arch_data;
struct spinlock lock;
};
```

File: ./include/linkedlist.h

```
#pragma once

#include <common.h>

struct linked_list;
struct linked_list_node;

struct linked_list* LinkedListCreate(void);
void LinkedListInsertStart(struct linked_list* list, void* data);
void LinkedListInsertEnd(struct linked_list* list, void* data);
bool LinkedListContains(struct linked_list* list, void* data);
```

```
int LinkedListGetIndex(struct linked_list* list, void* data);
void* LinkedListGetData(struct linked_list* list, int index);
bool LinkedListDeleteIndex(struct linked_list* list, int index);
bool LinkedListDeleteData(struct linked_list* list, void* data);
int LinkedListSize(struct linked_list* list);
void LinkedListDestroy(struct linked_list* list);

struct linked_list_node* LinkedListGetFirstNode(struct linked_list* list);
struct linked_list_node* LinkedListGetNextNode(struct linked_list_node* prev_node);
void* LinkedListGetDataFromNode(struct linked_list_node* node);
```

File: ./include/driver.h

```
#pragma once

#include <common.h>

struct quick_relocation {
size_t address;
size_t value;
};

struct quick_relocation_table {
int total_entries;
int used_entries;
struct quick_relocation* entries;
};

struct vas;

void InitSymbolTable(void);
int RequireDriver(const char* name);
size_t GetDriverAddress(const char* name);
size_t GetSymbolAddress(const char* symbol);
void AddSymbol(const char* symbol, size_t address);

void SortQuickRelocationTable(struct quick_relocation_table* table);
void AddToQuickRelocationTable(struct quick_relocation_table* table, size_t addr, size_t val);
struct quick_relocation_table* CreateQuickRelocationTable(int count);
void PerformDriverRelocationOnPage(struct vas*, size_t relocation_base, size_t virt);
```

File: ./include/dev.h

```
#pragma once

void InitNullDevice(void);
void InitRandomDevice(void);
```

File: ./include/vfs.h

```
#pragma once

#include <common.h>
#include <sys/types.h>
#include <openfile.h>
#include <vnode.h>
#include <transfer.h>

void InitVfs(void);
int AddVfsMount(struct vnode* node, const char* name);
int RemoveVfsMount(const char* name);

int OpenFile(const char* path, int flags, mode_t mode, struct open_file** out);
int ReadFile(struct open_file* file, struct transfer* io);
int WriteFile(struct open_file* file, struct transfer* io);
int CloseFile(struct open_file* file);
int GetFileSize(struct open_file* file, off_t* size);
```

File: ./include/diskcache.h

```
#pragma once

#include <common.h>

#define DISKCACHE_NORMAL 0
#define DISKCACHE_REDUCE 1
#define DISKCACHE_TOSS 2

void InitDiskCaches(void);
void SetDiskCaches(int mode);

struct open_file* CreateDiskCache(struct open_file* underlying_disk);
```

File: ./include/timer.h

```
#pragma once

#include <common.h>

export uint64_t GetSystemTimer(void);

void ReceivedTimer(uint64_t nanos);
void InitTimer(void);

/*
* Internal functions to do shenanigans
*/
struct thread;
void QueueForSleep(struct thread* thr);
bool TryDequeueForSleep(struct thread* thr);
```

File: ./include/unicode.h

```
#pragma once

#include <common.h>

int Utf16ToCodepoints(uint16_t* utf16, uint32_t* codepoints, int in_length, int* out_length);
int Utf8ToCodepoints(uint8_t* utf8, uint32_t* codepoints, int in_length, int* out_length);

int CodepointsToUtf16(uint32_t* codepoints, uint16_t* utf16, int in_length, int* out_length);
int CodepointsToUtf8(uint32_t* codepoints, uint8_t* utf8, int in_length, int* out_length);
```

File: ./include/swapfile.h

```
#pragma once

#include <common.h>

void InitSwapfile(void);

struct open_file* GetSwapfile(void);
uint64_t AllocateSwapfileIndex(void);
void DeallocateSwapfileIndex(uint64_t index);
int GetNumberOfPagesOnSwapfile(void);
```

File: ./include/voidptr.h

```
#pragma once
#include <stdint.h>

#define AddVoidPtr(ptr, offset) ((void*) (((uint8_t*) ptr) + offset))
#define SubVoidPtr(ptr, offset) ((void*) (((uint8_t*) ptr) - offset))
```

File: ./include/thread.h

```
#pragma once

#include <common.h>

struct semaphore;
struct process;

#define THREAD_STATE_RUNNING 0
#define THREAD_STATE_READY 1
#define THREAD_STATE_SLEEPING 2
#define THREAD_STATE_WAITING_FOR_SEMAPHORE 3
#define THREAD_STATE_WAITING_FOR_SEMAPHORE_WITH_TIMEOUT 4
#define THREAD_STATE_TERMINATED 5

#define SCHEDULE_POLICY_FIXED 0
#define SCHEDULE_POLICY_USER_HIGHER 1
#define SCHEDULE_POLICY_USER_NORMAL 2
#define SCHEDULE_POLICY_USER_LOWER 3

#define FIXED_PRIORITY_KERNEL_HIGH 0
#define FIXED_PRIORITY_KERNEL_NORMAL 30
#define FIXED_PRIORITY_IDLE 255

/*
* Determines which of the 'next' pointers are used to manage the list.
* A thread can be on multiple lists so long as they are different numbers.
* Can increase the number of 'next' pointers in the thread struct to make them distinct if needed.
*/
#define NEXT_INDEX_READY 0
#define NEXT_INDEX_SLEEP 1
#define NEXT_INDEX_SEMAPHORE 2
#define NEXT_INDEX_TERMINATED 0 // terminated can share the ready list

struct thread {
/*
* These first two values must be in this order.
*/
size_t kernel_stack_top;
size_t stack_pointer;

struct vas* vas;
size_t kernel_stack_size;
void (*initial_address)(void*);

/*
* Allows a thread to be on a timer and a semaphore list at the same time.
* Very sketchy stuff.
*/
struct thread* next[3];

int thread_id;
int state;
void* argument;
uint64_t time_used;
char* name;
int priority;
int schedule_policy;
size_t canary_position;
bool timed_out;
bool needs_termination;

struct semaphore* waiting_on_semaphore;

struct process* process;

/*
* The system time at which this task's time has expired. If this is 0, then the task will not have a set time limit.
* This value is set to GetSystemTimer() + TIMESLICE_LENGTH_MS when the task is scheduled in, and doesn't change until
* the next time it is switched in.
```

```
*/
uint64_t timeslice_expiry;

uint64_t sleep_expiry;
};

void Schedule(void);
void LockSchedulerX(void);
void UnlockSchedulerX(void);
#define LockScheduler() /*LogWriteSerial("LOCKING SCHEDULER: %s, %s, %d\n", __FILE__, __func__, __LINE__);*/ LockSchedulerX()
#define UnlockScheduler() /*LogWriteSerial("UNLOCKING SCHEDULER: %s, %s, %d\n", __FILE__, __func__, __LINE__);*/ UnlockSchedulerX()

void InitScheduler(void);
void StartMultitasking(void);

void AssertSchedulerLockHeld(void);

struct thread* GetThread(void);
void TerminateThread(struct thread* thr);
void TerminateThreadLockHeld(struct thread* thr);

struct thread* CreateThreadEx(void(*entry_point)(void*), void* argument, struct vas* vas, const char* name, struct process* prcss, int policy, int priority, int kernel_stack_kb);
struct thread* CreateThread(void(*entry_point)(void*), void* argument, struct vas* vas, const char* name);

void BlockThread(int reason);
void UnblockThread(struct thread* thr);
int SetThreadPriority(struct thread* thread, int policy, int priority);

void SleepUntil(uint64_t system_time_ns);
void SleepNano(uint64_t delta_ns);
void SleepMilli(uint32_t delta_ms);

void HandleSleepWakeups(void* sys_time_ptr); // used internally between timer.c and thread.c
void InitIdle(void);
void InitCleaner(void);

struct process* CreateUsermodeProcess(struct process* parent, const char* filename);

/*
* A thread can lock itself onto the current cpu. Task switches *STILL OCCUR*, but we ensure that
* next time this task runs, it will go back to this cpu.
*
* This is not a spinlock nor mutex, it's literally should just set a flag in the thread struct (sure, that
* will spin while setting variable, but that's it). Between AssignThreadToCpu and UnassignThreadToCpu we remain
* at IRQL_STANDARD.
*/
void AssignThreadToCpu(void);
void UnassignThreadToCpu(void);
```

File: ./include/common.h

```
#pragma once

#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdarg.h>

#define export __attribute__((used))

#ifndef NULL
#define NULL ((void*) 0)
#endif

#define warn_unused __attribute__((warn_unused_result))
#define always_inline __attribute__((always_inline)) inline

#define PAGEABLE_CODE_SECTION __attribute__((__section__(".pageablektext")))
#define PAGEABLE_DATA_SECTION __attribute__((__section__(".pageablekdata")))

#define NO_EXPORT __attribute__((visibility("hidden")))
#define EXPORT __attribute__((visibility("default")))

#define LOCKED_DRIVER_CODE __attribute__((__section__(".lockedtext")))
#define LOCKED_DRIVER_DATA __attribute__((__section__(".lockeddata")))
#define LOCKED_DRIVER_RODATA __attribute__((__section__(".lockedrodata")))

#define inline_memcpy(dst, src, n) __builtin_memcpy(dst, src, n)
#define inline_memset(dst, v, n) __builtin_memset(dst, v, n)

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define CLAMP(val, min, max) MAX(MIN(val, max), min)
#define COMPARE_SIGN(a, b) ((a) > (b) ? 1 : ((a) < (b) ? -1 : 0))
```

File: ./include/priorityqueue.h

```
#pragma once

#include <stdbool.h>

struct priority_queue;

struct priority_queue_result {
uint64_t priority;
void* data;
};

struct priority_queue* PriorityQueueCreate(int capacity, bool max, int element_width);
void PriorityQueueInsert(struct priority_queue* queue, void* elem, uint64_t priority);
struct priority_queue_result PriorityQueuePeek(struct priority_queue* queue);
void PriorityQueuePop(struct priority_queue* queue);
int PriorityQueueGetCapacity(struct priority_queue* queue);
int PriorityQueueGetUsedSize(struct priority_queue* queue);
void PriorityQueueDestroy(struct priority_queue* queue);
```

File: ./include/avl.h

```
#pragma once

#include <common.h>

struct avl_tree;
struct avl_node;

typedef void (*avl_deletion_handler)(void*);
typedef int (*avl_comparator)(void*, void*);

void AvlTreePrint(struct avl_tree* tree, void(*printer)(void*));
struct avl_tree* AvlTreeCreate(void);
void AvlTreeInsert(struct avl_tree* tree, void* data);
void AvlTreeDelete(struct avl_tree* tree, void* data);
bool AvlTreeContains(struct avl_tree* tree, void* data);
void* AvlTreeGet(struct avl_tree* tree, void* data);
int AvlTreeSize(struct avl_tree* tree);
void AvlTreeDestroy(struct avl_tree* tree);
struct avl_node* AvlTreeGetRootNode(struct avl_tree* tree);
struct avl_node* AvlTreeGetLeft(struct avl_node* node);
struct avl_node* AvlTreeGetRight(struct avl_node* node);
void* AvlTreeGetData(struct avl_node* node);
avl_deletion_handler AvlTreeSetDeletionHandler(struct avl_tree* tree, avl_deletion_handler handler);
avl_comparator AvlTreeSetComparator(struct avl_tree* tree, avl_comparator comparator);
```

File: ./include/openfile.h

```
#pragma once

#include <common.h>
#include <sys/types.h>
#include <spinlock.h>

struct vnode;

struct open_file {
bool can_read;
bool can_write;
mode_t initial_mode;
size_t seek_position;
int flags;
int reference_count;
struct spinlock reference_count_lock;
struct vnode* node;
};

struct open_file* CreateOpenFile(struct vnode* node, int mode, int flags, bool can_read, bool can_write);
void ReferenceOpenFile(struct open_file* file);
void DereferenceOpenFile(struct open_file* file);
```

File: ./include/log.h

```
#pragma once

#include <common.h>

export void LogWriteSerial(const char* format, ...);
export void LogDeveloperWarning(const char* format, ...);

void DbgScreenPrintf(const char* format, ...);
void DbgScreenPuts(char* str);
void DbgScreenPutchar(char c);
```

File: ./include/threadlist.h

```
#pragma once

#include <common.h>

struct thread;

struct thread_list {
struct thread* head;
struct thread* tail;
int index;
};

void ThreadListInit(struct thread_list* list, int index);
void ThreadListInsert(struct thread_list* list, struct thread* thread);
bool ThreadListContains(struct thread_list* list, struct thread* thread);
void ThreadListDelete(struct thread_list* list, struct thread* thread);
struct thread* ThreadListDeleteTop(struct thread_list* list);
```

File: ./include/partition.h

```
#pragma once

#include <stdint.h>
#include <stddef.h>

struct open_file;
struct vnode;

struct open_file* CreatePartition(struct open_file* disk, uint64_t start, uint64_t length, int id, int sector_size, int media_type, bool boot);
struct open_file** GetPartitionsForDisk(struct open_file* disk);
```

File: ./include/_stdckdint.h

```
#if __has_include(<stdckdint.h>)
# include <stdckdint.h>
#else
# ifdef __GNUC__
# define ckd_add(R, A, B) __builtin_add_overflow ((A), (B), (R))
# define ckd_sub(R, A, B) __builtin_sub_overflow ((A), (B), (R))
# define ckd_mul(R, A, B) __builtin_mul_overflow ((A), (B), (R))
# else
# error "we need a compiler extension for this"
# endif
#endif
```

File: ./include/cpu.h

```
#pragma once

#include <common.h>
#include <arch.h>
#include <spinlock.h>

struct vas;
struct thread;
struct avl_tree;

struct cpu {
struct vas* current_vas;
struct thread* current_thread;
platform_cpu_data_t* platform_specific;
size_t cpu_number;
int irql;

struct priority_queue* deferred_functions;
struct priority_queue* irq_deferred_functions;
bool init_irql_done;
bool postponed_task_switch;

struct avl_tree* global_vas_mappings;
struct spinlock global_mappings_lock;
};

void InitCpuTable(void);
void InitBootstrapCpu(void);
void InitOtherCpu(void);

struct cpu* GetCpu(void);
int GetCpuCount(void);
struct cpu* GetCpuAtIndex(int index);
```

File: ./include/video.h

```
#pragma once

struct video_driver {
void (*putchar)(char);
void (*puts)(char*);
};

void InitVideoConsole(struct video_driver driver);
```

File: ./include/heap.h

```
#pragma once

#include <common.h>

/*
* Allocation will not fault. If this is impossible to achieve, the system will panic.
* Must not be set with HEAP_ALLOW_PAGING.
*/
#define HEAP_NO_FAULT 1

/*
* Clears allocated memory to zero.
*/
#define HEAP_ZERO 2

/*
* Indicates that the allocated region is allowed to be swapped onto disk.
* Must not be set with HEAP_NO_FAULT. Data allocated with this flag set can only be
* accessed when IRQL = IRQL_STANDARD.
*/
#define HEAP_ALLOW_PAGING 4

void* AllocHeap(size_t size);
void* AllocHeapEx(size_t size, int flags);
void* ReallocHeap(void* ptr, size_t size);
void* AllocHeapZero(size_t size);
void FreeHeap(void* ptr);
void ReinitHeap(void);
void InitHeap(void);

#ifndef NDEBUG
int DbgGetOutstandingHeapAllocations(void);
#endif


#define malloc(x) AllocHeap(x)
#define free(x) FreeHeap(x)
```

File: ./include/irq.h

```
#pragma once

#include <arch.h>
#include <common.h>

typedef int(*irq_handler_t)(platform_irq_context_t*);

int RegisterIrqHandler(int irq_num, irq_handler_t handler);
void RespondToIrq(int irq_num, int required_irql, platform_irq_context_t* context);
void UnhandledFault(void);
```

File: ./include/process.h

```
#pragma once

#include <common.h>
#include <sys/types.h>

struct filedes_table;
struct process;

void InitProcess(void);
struct process* CreateProcess(pid_t parent_pid);
struct process* ForkProcess(void);
pid_t WaitProcess(pid_t pid, int* status, int flags);
void KillProcess(int retv);

struct process* GetProcessFromPid(pid_t pid);
struct process* GetProcess(void);
pid_t GetPid(struct process* prcss);

struct filedes_table* GetFileDescriptorTable(struct process* prcss);

void AddThreadToProcess(struct process* prcss, struct thread* thr);
struct process* CreateProcessWithEntryPoint(pid_t parent, void(*entry_point)(void*), void* arg);
```

File: ./include/transfer.h

```
#pragma once

#include <common.h>

enum transfer_type {
TRANSFER_INTRA_KERNEL,
TRANSFER_USERMODE,
};

enum transfer_direction {
TRANSFER_READ,
TRANSFER_WRITE,
};

/*
* A data structure for performing file read and write operations, potentially
* between the kernel and the user.
*
* TODO: userspace handling
*/
struct transfer {
void* address;
uint64_t length_remaining; /* In bytes. Will be modified on copying */
uint64_t offset; /* In bytes. Will be modified on copying */

enum transfer_direction direction;
enum transfer_type type;
};


int PerformTransfer(void* trusted_buffer, struct transfer* untrusted_buffer, uint64_t len);

/*
* max_length of 0 means unbounded
*/
int WriteStringToUsermode(const char* trusted_string, char* untrusted_buffer, uint64_t max_length);
int ReadStringFromUsermode(char* trusted_buffer, const char* untrusted_string, uint64_t max_length);

int WriteWordToUsermode(size_t* location, size_t value);
int ReadWordFromUsermode(size_t* location, size_t* output);

struct transfer CreateKernelTransfer(void* addr, uint64_t length, uint64_t offset, int direction);
struct transfer CreateTransferWritingToUser(void* untrusted_addr, uint64_t length, uint64_t offset);
struct transfer CreateTransferReadingFromUser(const void* untrusted_addr, uint64_t length, uint64_t offset);
```

File: ./include/panic.h

```
#pragma once

#include <common.h>

enum {
PANIC_UNKNOWN,

/*
* A non-returnable function or infinite loop was exited out of.
*/
PANIC_IMPOSSIBLE_RETURN,

/*
* The panic was requested by the debugger.
*/
PANIC_MANUALLY_INITIATED,

/*
```

```
* A unit test succeeded. Only to be used with the unit testing framework,
* which panics to either succeed or fail (via assertion fails).
*/
PANIC_UNIT_TEST_OK,

/*
* Used by drivers to report unrecoverable faults.
*/
PANIC_DRIVER_FAULT,

/*
* The kernel heap is out of memory and cannot request any more.
*/
PANIC_OUT_OF_HEAP,

/*
* Too much heap memory has been allocated before the virtual memory manager has
* been initialised.
*/
PANIC_OUT_OF_BOOTSTRAP_HEAP,

/*
* A request for a block on the heap was too large.
*/
PANIC_HEAP_REQUEST_TOO_LARGE,

/*
* The kernel or a driver has caused an illegal page fault.
*/
PANIC_PAGE_FAULT_IN_NON_PAGED_AREA,

/*
* An assertion failure within the kernel or driver.
*/
PANIC_ASSERTION_FAILURE,

/*
* The bootloader failed to provide a usable memory map.
*/
PANIC_NO_MEMORY_MAP,

/*
* The given section of kernel code is not implemented yet.
*/
PANIC_NOT_IMPLEMENTED,

/*
* Wrong IRQL
*/
PANIC_INVALID_IRQL,

/*
* Spinlock acquired from the wrong IRQL level.
*/
PANIC_SPINLOCK_WRONG_IRQL,

/*
* No more physical memory, even after evicting old pages.
*/
PANIC_OUT_OF_PHYS,

PANIC_PRIORITY_QUEUE,
PANIC_LINKED_LIST,

/*
* Kernel stack overflow
*/
PANIC_CANARY_DIED,

PANIC_SEMAPHORE_DESTROY_WHILE_HELD,
PANIC_SEM_BLOCK_WITHOUT_THREAD,
PANIC_CANNOT_LOCK_MEMORY,
PANIC_THREAD_LIST,
PANIC_CANNOT_MALLOC_WITHOUT_FAULTING,
PANIC_NO_FILESYSTEM,
PANIC_BAD_KERNEL,
PANIC_DISK_FAILURE_ON_SWAPFILE,
PANIC_NEGATIVE_SEMAPHORE,
PANIC_NON_MASKABLE_INTERRUPT,
PANIC_UNHANDLED_KERNEL_EXCEPTION,
PANIC_REQUIRED_DRIVER_MISSING_SYMBOL,
PANIC_REQUIRED_DRIVER_NOT_FOUND,
PANIC_NO_LOW_MEMORY,
PANIC_OUT_OF_SWAPFILE,
PANIC_PROGRAM_LOADER,
PANIC_VAS_TRIED_TO_SELF_DESTRUCT,
PANIC_ACPI_AML,
PANIC_SPINLOCK_DOUBLE_ACQUISITION,
PANIC_SPINLOCK_RELEASED_BEFORE_ACQUIRED,
PANIC_DOUBLE_FREE_DETECTED,
PANIC_CONFLICTING_ALLOCATION_REQUIREMENTS,

_PANIC_HIGHEST_VALUE
};


_Noreturn void PanicEx(int code, const char* message);
_Noreturn void Panic(int code);

const char* GetPanicMessageFromCode(int code);
int SetGraphicalPanicHandler(void (*handler)(int, const char*));
```

File: ./include/diskutil.h

```
#pragma once

#include <common.h>
```

```
struct vnode;
struct open_file;

#define MAX_PARTITIONS_PER_DISK 8

#define DISKUTIL_TYPE_FIXED 0
#define DISKUTIL_TYPE_FLOPPY 1
#define DISKUTIL_TYPE_OPTICAL 2
#define DISKUTIL_TYPE_REMOVABLE 3
#define DISKUTIL_TYPE_NETWORK 4
#define DISKUTIL_TYPE_VIRTUAL 5
#define DISKUTIL_TYPE_RAM 6
#define DISKUTIL_TYPE_OTHER 7

#define __DISKUTIL_NUM_TYPES 8

struct disk_partition_helper {
struct vnode* partitions[MAX_PARTITIONS_PER_DISK];
char* partition_names[MAX_PARTITIONS_PER_DISK];
int num_partitions;
};

void InitDiskUtil(void);
char* GenerateNewRawDiskName(int type);
char* GenerateNewMountedDiskName();
void CreateDiskPartitions(struct open_file* disk);
void InitDiskPartitionHelper(struct disk_partition_helper* helper);

int DiskFollowHelper(struct disk_partition_helper* helper, struct vnode** out, const char* name);
int DiskCreateHelper(struct disk_partition_helper* helper, struct vnode** in, const char* name);
```

File: ./include/fs/internal/fat.h

```
#pragma once

#include <common.h>

struct open_file;

#define LFN_SHORT_ONLY 0
#define LFN_BOTH 1
#define LFN_ERROR 2

#define FAT12 0
#define FAT16 2 // the value of 2 is relied on, as it means 2 bytes per FAT (is used for calcs)
#define FAT32 4 // as above, we use fat_type to do calcs, so required that FAT32 == 4

struct fat_data {
int num_fats;
int fat_sectors[4];
int sectors_per_fat;
union {
uint64_t first_root_dir_sector_12_16;
uint64_t root_dir_cluster_32;
};
uint64_t root_dir_num_sectors_12_16;
int total_clusters;
uint64_t first_data_sector;
uint64_t first_fat_sector;
int fat_type; // FAT12 or FAT16 or FAT32
int sectors_per_cluster;
int bytes_per_sector;

struct open_file* disk; // TODO! points to a vnode for the partition

uint8_t* cluster_buffer_a;
uint8_t* cluster_buffer_b;
};

int GetFatShortFilename(char* lfn, char* output, char* directory);
void FormatFatShortName(char* with_dot, char* without_dot);
void UnformatFatShortName(char* without_dot, char* with_dot);

int ReadFatCluster(struct fat_data* fat, int cluster, bool buffer);
int WriteFatCluster(struct fat_data* fat, int cluster, bool buffer);
int ReadFatEntry(struct fat_data* fat, int entry, uint32_t* output);
int WriteFatEntry(struct fat_data* fat, int entry, uint32_t value);

struct fat_data LoadFatData(uint8_t* boot_sector, struct open_file* disk);
```

File: ./include/fs/fat.h

```
#pragma once

#include <common.h>

int DetectFatPartition(void* partition);
```

File: ./include/syscall.h

```
#pragma once

#include <common.h>

int HandleSystemCall(int call, size_t a, size_t b, size_t c, size_t d, size_t e);

int SysYield(size_t, size_t, size_t, size_t, size_t);
int SysTerminate(size_t, size_t, size_t, size_t, size_t);
int SysMapVirt(size_t, size_t, size_t, size_t, size_t);
int SysUnmapVirt(size_t, size_t, size_t, size_t, size_t);
int SysOpen(size_t, size_t, size_t, size_t, size_t);
int SysReadWrite(size_t, size_t, size_t, size_t, size_t);
int SysClose(size_t, size_t, size_t, size_t, size_t);
```

```
int SysSeek(size_t, size_t, size_t, size_t, size_t);
int SysDup(size_t, size_t, size_t, size_t, size_t);
```

File: ./include/physical.h

```
#pragma once

#include <common.h>

void DeallocPhys(size_t addr);
void DeallocPhysContiguous(size_t addr, size_t bytes);
size_t AllocPhys(void);
size_t AllocPhysContiguous(size_t bytes, size_t min_addr, size_t max_addr, size_t boundary);
size_t GetTotalPhysKilobytes(void);
size_t GetFreePhysKilobytes(void);

void InitPhys(void);
void ReinitPhys(void);
```

File: ./include/debug/tfw_tests.h

```
#pragma once

#ifndef NDEBUG

void RegisterTfwPhysTests(void);
void RegisterTfwInitTests(void);
void RegisterTfwIrqlTests(void);
void RegisterTfwAVLTreeTests(void);
void RegisterTfwPriorityQueueTests(void);
void RegisterTfwSemaphoreTests(void);
void RegisterTfwWaitTests(void);

#endif
```

File: ./include/debug/tfw.h

```
#pragma once

#include <common.h>

enum {
TFW_SP_INITIAL,
TFW_SP_AFTER_PHYS,
TFW_SP_AFTER_HEAP,
TFW_SP_AFTER_BOOTSTRAP_CPU,
TFW_SP_AFTER_VIRT,
TFW_SP_AFTER_PHYS_REINIT,
TFW_SP_AFTER_ALL_CPU,

TFW_SP_ALL_CLEAR,
};

#ifdef NDEBUG

#define IsInTfwTest() false
#define FinishedTfwTest(x)
#define MarkTfwStartPoint(x)
#define InitTfw()

#else

bool IsInTfwTest(void);

// this can probably go up to around 150,000 or so in theory (in what the transfer format supports), or about 20,000 on a 4MB RAM system.
// but bigger is slower, so only increase as we need to
#define MAX_TWF_TESTS 100
#define MAX_NAME_LENGTH 96 // If this changes the python must do too


struct tfw_test {
char name[MAX_NAME_LENGTH];
void (*code)(struct tfw_test*, size_t context);
int start_point;
int expected_panic_code;
bool nightly_only;
size_t context;
};

#define TFW_IGNORE_UNUSED (void) test; (void) context;
#define TFW_CREATE_TEST(name) static void name (struct tfw_test* test, size_t context)

void RegisterTfwTest(const char* name, int start_point, void (*code)(struct tfw_test*, size_t), int expected_panic, size_t context);
void RegisterNightlyTfwTest(const char* name, int start_point, void (*code)(struct tfw_test*, size_t), int expected_panic, size_t context);

void FinishedTfwTest(int panic_code);
void MarkTfwStartPoint(int id);
void InitTfw(void);

#endif
```

File: ./include/debug/hostio.h

```
#pragma once

#ifdef NDEBUG

#else

#include <common.h>
```

```
#define DBGPKT_TFW 0

void DbgWritePacket(int type, uint8_t* data, int size);
void DbgReadPacket(int* type, uint8_t* data, int* size);

#endif
```

File: ./include/arch.h

```
#pragma once

/*
* arch.h - Architecture-specific wrappers
*
*
* Functions relating to hardware devices that must be implemented by
* any platform supporting the operating system.
*
*/

/*
* config.h needs to define the following:
* - ARCH_PAGE_SIZE
* - ARCH_MAX_CPU_ALLOWED
* - ARCH_MAX_RAM_KBS
* - ARCH_BIG_ENDIAN or ARCH_LITTLE_ENDIAN
* - the address in the kernel area, ARCH_PROG_LOADER_BASE, where the program loader lives, and
* - ARCH_PROG_LOADER_ENTRY, the entry point of the prog loader
* - the valid user area, via ARCH_USER_AREA_BASE and ARCH_USER_AREA_LIMIT
* - the valid kernel area, via ARCH_KRNL_SBRK_BASE and ARCH_KRNL_SBRK_LIMIT
* (the kernel and user areas must not overlap, but ARCH_USER_AREA_LIMIT may equal ARCH_KRNL_SBRK_BASE
or ARCH_KRNL_SBRK_LIMIT may equal ARCH_USER_AREA_BASE)
* - the user stack area, via ARCH_USER_STACK_BASE and ARCH_USER_STACK_LIMIT
* (may overlap with ARCH_USER_AREA_BASE and ARCH_USER_AREA_LIMIT)
* - a typedef for platform_cpu_data_t
* - a typedef for platform_irq_context_t
* - a typedef for platform_vas_data_t
*/


#include <machine/config.h>

#if ARCH_USER_STACK_BASE < ARCH_USER_AREA_BASE
#error "ARCH_USER_STACK_BASE must be greater than or equal to ARCH_USER_AREA_BASE"
#elif ARCH_USER_STACK_LIMIT > ARCH_USER_AREA_LIMIT
#error "ARCH_USER_STACK_LIMIT must be less than or equal to ARCH_USER_AREA_LIMIT"
#endif

#include <common.h>

struct arch_memory_range
{
size_t start;
size_t length;
};

struct vas;
struct vas_entry;
struct thread;
struct open_file;
struct cpu;
struct quick_relocation_table;

struct arch_driver_t;

/*
* Only to be called in very specific places, e.g. turning interrupts
* on for the first time, the panic handler.
*/
void ArchEnableInterrupts(void);
void ArchDisableInterrupts(void);

/*
* Do nothing until (maybe) the next interrupt. If this is not supported by the
* system it may just return without doing anything.
*/
void ArchStallProcessor(void);

#define ARCH_POWER_STATE_REBOOT 1
#define ARCH_POWER_STATE_SHUTDOWN 2
#define ARCH_POWER_STATE_SLEEP 3
int ArchSetPowerState(int power_state);

void ArchSpinlockAcquire(volatile size_t* lock);
void ArchSpinlockRelease(volatile size_t* lock);

/*
* To be called repeatedly until it returns NULL. Each time will return a new memory
* range. An address of a static local object is permitted to be returned.
*
* NULL is returned if there is no more memory. No more calls to this function
* will be made after a NULL is returned.
*/
struct arch_memory_range* ArchGetMemory() warn_unused;

uint64_t ArchReadTimestamp(void);

void ArchFlushTlb(struct vas* vas);
void ArchAddMapping(struct vas* vas, struct vas_entry* entry);
void ArchUpdateMapping(struct vas* vas, struct vas_entry* entry);
void ArchUnmap(struct vas* vas, struct vas_entry* entry);
void ArchSetVas(struct vas* vas);

void ArchGetPageUsageBits(struct vas* vas, struct vas_entry* entry, bool* accessed, bool* dirty);
void ArchSetPageUsageBits(struct vas* vas, struct vas_entry* entry, bool accessed, bool dirty);
```

```
// responsible for loading all symbols. should not close the file!
int ArchLoadDriver(size_t* relocation_point, struct open_file* file, struct quick_relocation_table** table);
void ArchLoadSymbols(struct open_file* file, size_t adjust);
void ArchSwitchThread(struct thread* old, struct thread* new);
size_t ArchPrepareStack(size_t addr);

void ArchSwitchToUsermode(size_t entry_point, size_t user_stack, void* arg);

void ArchInitDev(bool fs);


/*
* Used only if the AVL tree is insufficient, e.g. for deallocating part of the kernel region to, e.g.
* reclaim the physical memory bitmap. Works only for the current VAS. Returns 0 on no mapping.
*/
size_t ArchVirtualToPhysical(size_t virtual);

/*
* Initialises a given VAS with platform specific data (e.g. mapping the kernel in).
*/
void ArchInitVas(struct vas* vas);

/*
* Initialises virtual memory in general, i.e. creates the first VAS.
*/
void ArchInitVirt(void);

int ArchGetCurrentCpuIndex(void);
void ArchSendEoi(int irq_num);
/*
* Sets the CPUs interrupt state (and mask devices) based on an IRQL. This function
* will always be called with interrupts completely disabled.
*/
void ArchSetIrql(int irql);

void ArchInitBootstrapCpu(struct cpu* cpu);

/*
* If possible, initialises the next CPU, and returns true. If there are no more CPUs
* to initialise, returns false.
*/
bool ArchInitNextCpu(struct cpu* cpu);
```

File: ./irq/irq.c

```
1
2  #include <arch.h>
3  #include <irql.h>
4  #include <log.h>
5  #include <irq.h>
6  #include <thread.h>
7  #include <linkedlist.h>
8  #include <errno.h>
9  #include <process.h>
10 #include <panic.h>
11
12 #define HIGHEST_IRQ_NUM 256
13
14 static struct linked_list* irq_table[HIGHEST_IRQ_NUM] = {0};
15
16 int RegisterIrqHandler(int irq_num, irq_handler_t handler) {
17     if (irq_num < 0 || irq_num >= HIGHEST_IRQ_NUM || handler == NULL) {
18         return EINVAL;
19     }
20
21     if (irq_table[irq_num] == NULL) {
22         irq_table[irq_num] = LinkedListCreate();
23     }
24
25     LinkedListInsertEnd(irq_table[irq_num], (void*)(size_t) handler);
26     return 0;
27 }
28
29 void RespondToIrq(int irq_num, int required_irql, platform_irq_context_t* context) {
30     int irql = RaiseIrql(required_irql);
31     ArchSendEoi(irq_num);   /* this must be done after raising the IRQL */
32
33     if (irq_table[irq_num] != NULL) {
34         struct linked_list_node* iter = LinkedListGetFirstNode(irq_table[irq_num]);
35         while (iter != NULL) {
36             irq_handler_t handler = (irq_handler_t)(size_t) LinkedListGetDataFromNode(iter);
37             assert(handler != NULL);
38
39             /*
40             * Interrupt handlers return 0 if they could handle the IRQ (i.e. stop trying to handle it).
41             * Non-zero means 'leave this one for someone else'.
42             */
43             if (handler(context) == 0) {
44                 break;
45             }
46
47             iter = LinkedListGetNextNode(iter);
48         }
49     }
50
51     LowerIrql(irql);
52 }
53
54 void UnhandledFault(void) {
55     if (GetProcess() != NULL) {
56         LogWriteSerial("unhandled fault...\n");
57         TerminateThread(GetThread());
58
59     } else {
60         Panic(PANIC_UNHANDLED_KERNEL_EXCEPTION);
61     }
62 }
```

File: ./irq/cpu.c

```c
#include <cpu.h>
#include <arch.h>
#include <assert.h>
#include <heap.h>
#include <string.h>
#include <spinlock.h>
#include <irql.h>

static struct cpu cpu_table[ARCH_MAX_CPU_ALLOWED];
static int num_cpus_running = 1;

static void InitCpuTableEntry(int index) {
    /*
     * The boot CPU can't use dynamic memory, as this happens before we have a heap.
     */
    static platform_cpu_data_t boot_cpu_data;

    cpu_table[index].cpu_number = index;
    cpu_table[index].platform_specific = index != 0 ? AllocHeapZero(sizeof(platform_cpu_data_t)) : &boot_cpu_data;
    cpu_table[index].irql = IRQL_STANDARD;
    cpu_table[index].global_vas_mappings = NULL;
    cpu_table[index].current_vas = NULL;
    cpu_table[index].current_thread = NULL;
    cpu_table[index].init_irql_done = false;
    cpu_table[index].postponed_task_switch = false;
    InitSpinlock(&cpu_table[index].global_mappings_lock, "glbl vas map", IRQL_SCHEDULER);
}

/*
 * Initialises the CPU table (`cpu_table`) for the bootstrap processor. This does *not* do any
 * platform-specific initialisation, as that requires other features to be set up first.
 */
void InitCpuTable(void) {
    assert(num_cpus_running == 1);
    InitCpuTableEntry(0);
}

/*
 * Performs platform-specific initialisation of the bootstrap CPU (e.g. initialising interrupts,
 * system timers, segments, etc.).
 */
void InitBootstrapCpu(void) {
    EXACT_IRQL(IRQL_STANDARD);
    ArchInitBootstrapCpu(cpu_table);
}

void InitOtherCpu(void) {
    /*
     * We initialise the next entry, even before we know there's a CPU there. If there isn't a
     * CPU there, we don't increment `num_cpus_running` so the entry won't get used.
     */
    InitCpuTableEntry(1);

    while (ArchInitNextCpu(cpu_table + num_cpus_running)) {
        ++num_cpus_running;
        InitCpuTableEntry(num_cpus_running);
    }
}

int GetCpuCount(void) {
    return num_cpus_running;
}

struct cpu* GetCpuAtIndex(int index) {
    assert(index >= 0 && index < GetCpuCount());
    return cpu_table + index;
}

struct cpu* GetCpu(void) {
    return cpu_table + ArchGetCurrentCpuIndex();
}
```

File: ./irq/irql.c

```c

#include <panic.h>
#include <cpu.h>
#include <irql.h>
#include <log.h>
#include <priorityqueue.h>
#include <thread.h>
#include <assert.h>

/**
 * Runs a function at an IRQL lower than or equal to the current IRQL. If the IRQLs match,
 * the function will be run immediately. If the target IRQL is lower than the current IRQL,
 * it will be deferred until the IRQL level drops below IRQL_SCHEDULER. Deferred function
 * calls will be run in order from highest IRQL to lowest. If InitIrql() has not been called
 * prior to calling this function, any requests meant to be deferred will instead be silently
 * ignored (this is needed to bootstrap the physical memory manager, et al.).
 *
 * @param irql      The IRQL level to run the function at. This IRQL must be lower than the
 *                  current IRQL, or a panic will occur.
 * @param handler   The function to be run.
 * @param context   An arguement given to the handler function.
 */
void DeferUntillIrql(int irql, void(*handler)(void*), void* context) {
    if (irql == GetIrql() || (irql == IRQL_STANDARD_HIGH_PRIORITY && GetIrql() == IRQL_STANDARD)) {
        handler(context);

    } else if (irql > GetIrql()) {
        PanicEx(PANIC_INVALID_IRQL, "invalid irql on DeferUntilIrql");

    } else if (GetCpu()->init_irql_done) {
        struct irql_deferment deferment = {.context = context, .handler = handler};
        PriorityQueueInsert(GetCpu()->deferred_functions, (void*) &deferment, irql);
    }
}

int GetIrql(void) {
    return GetCpu()->irql;
}

int RaiseIrql(int level) {
    ArchDisableInterrupts();

    struct cpu* cpu = GetCpu();
    int existing_level = cpu->irql;

    if (level < existing_level) {
        PanicEx(PANIC_INVALID_IRQL, "invalid irql on RaiseIrql");
    }

    cpu->irql = level;
    ArchSetIrql(level);

    return existing_level;
}

void LowerIrql(int target_level) {
    struct cpu* cpu = GetCpu();
    struct priority_queue* deferred_functions = cpu->deferred_functions;

    int current_level = cpu->irql;

    if (target_level > current_level) {
        PanicEx(PANIC_INVALID_IRQL, "invalid irql on LowerIrql");
    }

    while (cpu->init_irql_done && PriorityQueueGetUsedSize(deferred_functions) > 0) {
        struct priority_queue_result next = PriorityQueuePeek(deferred_functions);
        assert((int) next.priority <= current_level || (next.priority == IRQL_STANDARD_HIGH_PRIORITY && current_level == IRQL_STANDARD));

        if ((int) next.priority >= target_level) {
            current_level = next.priority;
            if (current_level == IRQL_STANDARD_HIGH_PRIORITY) {
                current_level = IRQL_STANDARD;
            }

            /*
             * Must Pop() before we call the handler (otherwise if the handler does a raise/lower, it will
             * retrigger itself and cause a recursion loop), and must also get data off the queue before we Pop().
             * Also must only actually lower the IRQL after doing this, so we don't get interrupted in between
             * (as someone else could then Raise/Lower, and mess us up.)
             */
            struct irql_deferment* deferred_call = (struct irql_deferment*) next.data;
            void* context = deferred_call->context;
            void (*handler)(void*) = deferred_call->handler;
            if (handler == NULL) {
                ArchSetIrql(current_level);
                continue;
            }
            PriorityQueuePop(deferred_functions);
            cpu->irql = current_level;
            ArchSetIrql(current_level);
            handler(context);

        } else {
            break;
        }
    }

    current_level = target_level;
    if (current_level == IRQL_STANDARD_HIGH_PRIORITY) {
        current_level = IRQL_STANDARD;
    }
    cpu->irql = current_level;
    ArchSetIrql(current_level);

    if (current_level == IRQL_STANDARD && cpu->postponed_task_switch) {
        cpu->postponed_task_switch = false;
        Schedule();
    }
}

void PostponeScheduleUntilStandardIrql(void) {
    // TODO: does this function need its own lock ? (e.g. just for setting postponed_task_switch)
    GetCpu()->postponed_task_switch = true;
}

/**
 * Requires TFW_SP_AFTER_HEAP or later.
 */
void InitIrql(void) {
    GetCpu()->deferred_functions = PriorityQueueCreate(32, true, sizeof(struct irql_deferment));
    GetCpu()->init_irql_done = true;
}

int GetNumberInDeferQueue(void) {
    return PriorityQueueGetUsedSize(GetCpu()->deferred_functions);
}
```

File: ./sys/calls/mapvirt.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <transfer.h>
#include <process.h>
#include <filedes.h>
#include <virtual.h>
#include <_stdckdint.h>
#include <sys/mman.h>

int SysMapVirt(size_t flags, size_t bytes, size_t fd, size_t offset, size_t virtual_) {
size_t* userptr_virt = (size_t*) virtual_;

if (flags & ~(VM_READ | VM_WRITE | VM_EXEC | VM_FILE | VM_FIXED_VIRT)) {
 return EINVAL;
}

size_t target_virtual;
int res = ReadWordFromUsermode(userptr_virt, &target_virtual);
if (res != 0) {
 return res;
}

if (target_virtual < ARCH_USER_AREA_BASE) {
 return EINVAL;
}

size_t end_of_virtual;
bool overflow = ckd_add(&end_of_virtual, target_virtual, bytes);

if (overflow || (end_of_virtual >= ARCH_USER_AREA_LIMIT)) {
 return EINVAL;
}

struct open_file* file = NULL;
if (flags & VM_FILE) {
 res = GetFileFromDescriptor(GetFileDescriptorTable(GetProcess()), fd, &file);
 if (file == NULL || res != 0) {
  return res;
 }
}

size_t output_virtual = MapVirt(0, target_virtual, bytes, flags | VM_USER | VM_LOCAL, file, offset);
if (output_virtual == 0) {
 return EINVAL;
}

return WriteWordToUsermode(userptr_virt, output_virtual);
}
```

File: ./sys/calls/seek.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <log.h>
#include <vfs.h>
#include <process.h>
#include <dirent.h>
#include <filedes.h>
#include <unistd.h>
#include <transfer.h>

int SysSeek(size_t fd, size_t pos_ptr, size_t whence, size_t, size_t) {
struct open_file* file;
int res = GetFileFromDescriptor(GetFileDescriptorTable(GetProcess()), fd, &file);

if (file == NULL || res != 0) {
 return res;
}

struct transfer io = CreateTransferReadingFromUser((void*) pos_ptr, sizeof(off_t), 0);
   off_t offset;
   if ((res = PerformTransfer(&offset, &io, sizeof(off_t)))) {
       return res;
   }

   int type = VnodeOpDirentType(file->node);
   if (type == DT_FIFO || type == DT_SOCK) {
       return ESPIPE;
   }

   if (whence == SEEK_CUR) {
       offset += file->seek_position;

   } else if (whence == SEEK_END) {
       struct stat st;
       if ((res = VnodeOpStat(file->node, &st))) {
           return res;
       }
       offset += st.st_size;

   } else if (whence != SEEK_SET) {
       return EINVAL;
   }

   file->seek_position = offset;

   io = CreateTransferWritingToUser((void*) pos_ptr, sizeof(off_t), 0);
   return PerformTransfer(&offset, &io, sizeof(off_t));
}
```

File: ./sys/calls/unmapvirt.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <transfer.h>
#include <virtual.h>
#include <_stdckdint.h>
#include <sys/mman.h>

int SysUnmapVirt(size_t virtual, size_t bytes, size_t, size_t, size_t) {
 if (virtual < ARCH_USER_AREA_BASE) {
  return EINVAL;
 }

 size_t end_of_virtual;
 bool overflow = ckd_add(&end_of_virtual, virtual, bytes);

 if (overflow || (end_of_virtual >= ARCH_USER_AREA_LIMIT)) {
  return EINVAL;
 }

 return UnmapVirt(virtual, bytes);
}
```

File: ./sys/calls/readwrite.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <log.h>
#include <vfs.h>
#include <transfer.h>
#include <fcntl.h>
#include <process.h>
#include <filedes.h>

int SysReadWrite(size_t fd, size_t size, size_t buffer, size_t br_out, size_t write) {
 struct filedes_table* table = GetFileDescriptorTable(GetProcess());
 struct open_file* file;
 int res = GetFileFromDescriptor(table, fd, &file);

 if (file == NULL || res != 0) {
  return res;
 }

 if (write && (file->flags & O_APPEND)) {
  struct stat st;
        if ((res = VnodeOpStat(file->node, &st))) {
            return res;
        }

        file->seek_position = st.st_size;
 }

 struct transfer io = CreateTransferWritingToUser((uint8_t*) buffer, size, file->seek_position);
 io.direction = write ? TRANSFER_WRITE : TRANSFER_READ;

 if ((res = (write ? WriteFile : ReadFile)(file, &io))) {
  return res;
 }

 size_t br = size - io.length_remaining;
 file->seek_position += br;

 return WriteWordToUsermode((size_t*) br_out, br);
}
```

File: ./sys/calls/terminate.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>

int SysTerminate(size_t, size_t, size_t, size_t, size_t) {
 TerminateThread(GetThread());
 return EFAULT;
}
```

File: ./sys/calls/dup.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <log.h>
#include <vfs.h>
#include <process.h>
#include <filedes.h>
#include <fcntl.h>

int SysDup(size_t dup_num, size_t old_fd, size_t new_fd, size_t flags, size_t) {
 struct filedes_table* table = GetFileDescriptorTable(GetProcess());

 if ((flags & ~O_CLOEXEC) != 0) {
  return EINVAL;
 }

 if (dup_num == 1) {
  int result_fd;
  int res = DuplicateFileDescriptor(table, old_fd, &result_fd);
  if (res != 0) {
   return res;
  }

  return WriteWordToUsermode((size_t*) new_fd, result_fd);

 } else if (dup_num == 2) {
  return DuplicateFileDescriptor2(table, old_fd, new_fd, flags);

 } else {
  return EINVAL;
 }
}
```

File: ./sys/calls/yield.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <log.h>

int SysYield(size_t, size_t, size_t, size_t, size_t) {
 Schedule();
 return 0;
}
```

File: ./sys/calls/open.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <transfer.h>
#include <log.h>
#include <fcntl.h>
#include <thread.h>
#include <process.h>
#include <vfs.h>
#include <filedes.h>

#define ALLOWABLE_FLAGS (O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC | O_APPEND | O_NONBLOCK | O_CLOEXEC | O_DIRECT | O_ACCMODE)

int SysOpen(size_t filename, size_t flags, size_t mode, size_t fdout, size_t) {
 char path[400];
 int fd;
 int res;

 if (flags & ~ALLOWABLE_FLAGS) {
  return EINVAL;
 }

 if ((res = ReadStringFromUsermode(path, (const char*) filename, 399))) {
  return res;
 }

 struct open_file* file;
 if ((res = OpenFile(path, flags, mode, &file))) {
  return res;
 }

 struct filedes_table* table = GetFileDescriptorTable(GetProcess());
 if ((res = CreateFileDescriptor(table, file, &fd, flags & O_CLOEXEC))) {
  CloseFile(file);
  return res;
 }

 if ((res = WriteWordToUsermode((size_t*) fdout, fd))) {
  CloseFile(file);
  RemoveFileDescriptor(table, file);
  return res;
 }

 return 0;
}
```

File: ./sys/calls/close.c

```c
#include <syscall.h>
#include <errno.h>
#include <_syscallnum.h>
#include <thread.h>
#include <log.h>
#include <vfs.h>
#include <process.h>
#include <filedes.h>

int SysClose(size_t fd, size_t, size_t, size_t, size_t) {
 struct open_file* file;
 int res;

 if ((res = GetFileFromDescriptor(GetFileDescriptorTable(GetProcess()), fd, &file))) {
  return res;
 }

 return CloseFile(file);
}
```

File: ./sys/.DS_Store

[binary]

File: ./sys/syscalls.c

```c
#include <syscall.h>
#include <log.h>
#include <errno.h>
#include <_syscallnum.h>

typedef int (*system_call_t)(size_t, size_t, size_t, size_t, size_t);

const char* syscall_names[_SYSCALL_NUM_ENTRIES] = {
 [SYSCALL_YIELD] = "yield",
 [SYSCALL_TERMINATE] = "terminate",
 [SYSCALL_MAPVIRT] = "map_virt",
 [SYSCALL_UNMAPVIRT] = "unmap_virt",
 [SYSCALL_OPEN] = "open",
 [SYSCALL_READWRITE] = "read/write",
 [SYSCALL_CLOSE] = "close",
 [SYSCALL_SEEK] = "seek",
 [SYSCALL_DUP] = "dup",
};

static const system_call_t system_call_table[_SYSCALL_NUM_ENTRIES] = {
 [SYSCALL_YIELD] = SysYield,
 [SYSCALL_TERMINATE] = SysTerminate,
 [SYSCALL_MAPVIRT] = SysMapVirt,
 [SYSCALL_UNMAPVIRT] = SysUnmapVirt,
 [SYSCALL_OPEN] = SysOpen,
 [SYSCALL_READWRITE] = SysReadWrite,
 [SYSCALL_CLOSE] = SysClose,
 [SYSCALL_SEEK] = SysSeek,
 [SYSCALL_DUP] = SysDup,
};

int HandleSystemCall(int call, size_t a, size_t b, size_t c, size_t d, size_t e) {
 if (call >= _SYSCALL_NUM_ENTRIES) {
  return ENOSYS;
 }

 return system_call_table[call](a, b, c, d, e);
}
```

File: ./arch/.DS_Store

[binary]

File: ./arch/x86/.DS_Store

[binary]

File: ./arch/x86/boot/kernel_entry.s

```
;
;
; x86/kernel_entry.asm - Kernel Entry Point
;
; We want the kernel to conform to the Multiboot 2 standard, by doing this
; the kernel can be loaded by common bootloaders, such as GRUB2, or directly
; by the QEMU emulator
;
;

; Therefore, we must first define a few constants and flags required by Multiboot
;
MBALIGN  equ 1 << 0
MEMINFO  equ 1 << 1
FLAGS  equ MBALIGN | MEMINFO
MAGIC  equ 0x1BADB002
CHECKSUM equ -(MAGIC + FLAGS)

; We put the multiboot in a special section which gets placed at the start of
; the binary. This allows the bootloader to find the Multiboot header.
;
section .multiboot.data
align 4
 dd MAGIC
 dd FLAGS
 dd CHECKSUM


; We need a stack, so for the bootstrap process. We can define a quick 16KB
; stack in the BSS section, which is always initialised to zeros
;
section .bss
align 16
stack_bottom:
resb 4 * 1024
stack_top:


; The kernel is being loaded to 0xC0100000. We need temporary bootstrap paging
; structures handled here so that we can get the kernel to 0xC0100000 in virtual
; memory.
;
; We will allocate one page directory, and one page table. With this, we can map
; 4MB of memory. As the kernel starts at 1MB, we can actually have a kernel
; of at most 3MB. We will replace these paging structures later once we get into
; the proper kernel (and we'll release the physical memory behind it too!)
;
align 4096
global boot_page_directory
global boot_page_table1
boot_page_directory: resb 4096
boot_page_table1: resb 4096

; The start of the kernel itself - this will be called by the bootloader
; We must place it in a special section so it appears at the start of the binary
;
section .multiboot.text
global _start:function (_start.end - _start) ; calculate size of the _start function
extern KernelMain
extern _kernel_end
_start:
 cli
 cld

    ; GRUB puts pointer in ebx, so we need to save it

    ; Work out how many pages in the first 4MB need to be mapped
    ; (we map the low 1MB, and then the kernel)
    mov ecx, _kernel_end
    add ecx, 0xFFF
    and ecx, 0x0FFFF000
    shr ecx, 12
```

```
    mov eax, 1024
    sub eax, ecx

; Get ready to loop over the page table
mov edi, boot_page_table1 - 0xC0000000
xor esi, esi
mov ecx, 1024       ; 1024 assumes 4MB of memory exists - we will only set the first 2MB as present
                    ; (as the kernel loads at 1MB, the kernel can be at most 1MB large)
                    ; (the page swapper will *hate you* if you 'invent' physical memory here)

.mapNextPage:
; Combine the address with the present and writable flags
mov edx, esi
or edx, 3
    cmp ecx, eax
    jg .keep          ; ** remember, the loop counter is going down **
    xor edx, edx      ; not present
.keep:
mov [edi], edx

.incrementPage:
; Move onto the next page table entry, and the next corresponding physical page
add esi, 4096
add edi, 4
loop .mapNextPage

.endMapping:
; Identity map and put the mappings at 0xC0000000
; This way we won't page fault before we jump over to the kernel in high memory
; (we are still in low memory)
mov [boot_page_directory - 0xC0000000 + 0], dword boot_page_table1 - 0xC0000000 + 3 + 256
mov [boot_page_directory - 0xC0000000 + 768 * 4], dword boot_page_table1 - 0xC0000000 + 3 + 256

; Set the page directory
mov ecx, boot_page_directory - 0xC0000000
mov cr3, ecx

; Enable paging
mov ecx, cr0
or ecx, (1 << 31)
or ecx, (1 << 16)  ; enforce read-only pages in ring 0
mov cr0, ecx

; This is why identity paging was required earlier, as paging is on, but we
; are still in low memory (i.e. at 0x100000-ish)

; Now jump to the higher half
lea ecx, KernelEntryPoint
jmp ecx
.end:

section .text

global vesa_pitch
global vesa_width
global vesa_height
global vesa_depth
global vesa_framebuffer

vesa_depth db 0
vesa_framebuffer dd 0
vesa_width dw 0
vesa_height dw 0
vesa_pitch dw 0

global x86_grub_table
x86_grub_table dd 0

; The proper entry point of the kernel. Assumes the kernel is mapped into memory
; at 0xC0100000.
KernelEntryPoint:
    ; GRUB puts the address of a table in EBX, which we must use to find the
; memory table. Note that we haven't trashed EBX up until this point.

    ; TODO: kernel assumes the table is below 4MB, and that it is paged in
    ; (which atm is only the case when it is below 1MB).
mov [x86_grub_table], ebx

    ; Grab the video data the bootloader put into memory.
    mov ax, [0x1000 + 16]
    mov [vesa_pitch], ax

    mov ax, [0x1000 + 18]
    mov [vesa_width], ax

    mov ax, [0x1000 + 20]
    mov [vesa_height], ax

    mov al, [0x1000 + 25]
    mov [vesa_depth], al

    mov eax, [0x1000 + 40]
    mov [vesa_framebuffer], eax

; Remove the identity paging and flush the TLB so the changes take effect
mov [boot_page_directory], dword 0
mov ecx, cr3
mov cr3, ecx

; On x86, we'll store the current CPU number in the DR3 register (so user code cannot modify it)
; Set it correctly now.
xor eax, eax
mov dr3, eax

; Set the stack to the one we defined
mov esp, stack_top

; Jump to the kernel main function
call KernelMain

; We should never get here, but halt just in case
cli
hlt
jmp $
```

File: ./arch/x86/cpu/interrupt.c

```
1
2  #include <machine/regs.h>
3  #include <machine/interrupt.h>
4  #include <machine/pic.h>
5  #include <log.h>
6  #include <irq.h>
7  #include <irql.h>
8  #include <virtual.h>
9  #include <syscall.h>
10 #include <panic.h>
11 #include <console.h>
12
13 #define ISR_SYSTEM_CALL 96
14 #define ISR_PAGE_FAULT  14
15 #define ISR_NMI         2
16
17 static bool ready_for_irqs = false;
18
19 static int GetRequiredIrql(int irq_num) {
20     if (irq_num == PIC_IRQ_BASE + 0) {
21         return IRQL_TIMER;
22     } else {
23         return IRQL_DRIVER + irq_num - PIC_IRQ_BASE;
24     }
25 }
26
27 void x86HandleInterrupt(struct x86_regs* r) {
28     int num = r->int_no;
29
30     if (num >= PIC_IRQ_BASE && num < PIC_IRQ_BASE + 16) {
31         RespondToIrq(num, GetRequiredIrql(num), r);
32
33     } else if (num == ISR_PAGE_FAULT) {
34         extern size_t x86GetCr2();
35
36         int type = 0;
37         if (r->err_code & 1) {
38             type |= VM_READ;
39         }
40         if (r->err_code & 2) {
41             type |= VM_WRITE;
42         }
43         if (r->err_code & 4) {
44             type |= VM_USER;
45         }
46         if (r->err_code & 16) {
47             type |= VM_EXEC;
48         }
49
50         LogWriteSerial("\n\nPage fault: cr2 0x%X, eip 0x%X, nos-err 0x%X\n", x86GetCr2(), r->eip, type);
51
52         HandleVirtFault(x86GetCr2(), type);
53
54     } else if (num == ISR_NMI) {
55         Panic(PANIC_NON_MASKABLE_INTERRUPT);
56
57     } else if (num == ISR_SYSTEM_CALL) {
58         r->eax = HandleSystemCall(r->eax, r->ebx, r->ecx, r->edx, r->esi, r->edi);
59
60     } else {
61         LogWriteSerial("Got interrupt %d. (r->eip = 0x%X)\n", num, r->eip);
62         UnhandledFault();
63     }
64 }
65
66 void ArchSendEoi(int irq_num) {
67     SendPicEoi(irq_num);
68 }
69
70 void ArchSetIrql(int irql) {
71     if (irql == IRQL_HIGH || irql == IRQL_TIMER || !x86IsReadyForIrqs()) {
72         /*
73          * Interrupts stay off.
74          */
75         return;
76     }
77
78     if (irql >= IRQL_DRIVER) {
79         int irq_num = irql - IRQL_DRIVER;
80
81         /*
82          * We want to disable all higher IRQs (as the PIC puts the lowest priority interrupts at
83          * high numbers), as well as our self. Allow IRQ2 to stay enabled as it is used internally.
84          */
85         uint16_t mask = (0xFFFF ^ ((1 << irq_num) - 1)) & ~(1 << 2);
86         DisablePicLines(mask);
87
88     } else {
89         /*
90          * Allow everything to go through.
91          */
92          DisablePicLines(0x0000);
93     }
94
95     ArchEnableInterrupts();
96 }
97
98 bool x86IsReadyForIrqs(void) {
99     return ready_for_irqs;
100 }
101
102 void x86MakeReadyForIrqs(void) {
103     ready_for_irqs = true;
104     RaiseIrql(GetIrql());
105 }
```

File: ./arch/x86/cpu/cpu.c

```
1
2  #include <stdbool.h>
3  #include <virtual.h>
4  #include <machine/gdt.h>
5  #include <machine/idt.h>
6  #include <machine/tss.h>
7  #include <machine/pic.h>
8  #include <machine/pit.h>
9  #include <cpu.h>
10 #include <machine/portio.h>
11 #include <machine/interrupt.h>
12 #include <errno.h>
13 #include <driver.h>
14
15 static void x86EnableNMIs(void) {
16     outb(0x70, inb(0x70) & 0x7F);
17     inb(0x71);
18 }
19
20 void ArchInitBootstrapCpu(struct cpu*) {
21     x86InitGdt();
22     x86InitIdt();
23     x86InitTss();
24
25     InitPic();
26     InitPit(40);
27
28     ArchEnableInterrupts();
29     x86MakeReadyForIrqs();
30     x86EnableNMIs();
31 }
32
33 bool ArchInitNextCpu(struct cpu*) {
34     return false;
35 }
36
37 static void x86Reboot(void) {
38     uint8_t good = 0x02;
39     while (good & 0x02) {
40         good = inb(0x64);
41  }
42     outb(0x64, 0xFE);
43 }
44
45 static void x86Shutdown(void) {
46     size_t acpicaShutdown = GetSymbolAddress("AcpicaShutdown");
47     if (acpicaShutdown != 0) {
48         ((void (*)(void)) acpicaShutdown)();
49     }
50
51     /*
52      * Some emulators have ways of doing a shutdown if we don't have ACPI support yet.
53      */
54     outw(0xB004, 0x2000);       // Bochs and old QEMU
55     outw(0x0604, 0x2000);       // New QEMU
56     outw(0x4004, 0x3400);       // VirtualBox
57     outw(0x0600, 0x0034);       // Cloud Hypervisor
58 }
59
60 static void x86Sleep(void) {
61     size_t acpicaSleep = GetSymbolAddress("AcpicaSleep");
62     if (acpicaSleep != 0) {
63         ((void (*)(void)) acpicaSleep)();
64     }
65 }
66
67 int ArchSetPowerState(int power_state) {
68  switch (power_state) {
69     case ARCH_POWER_STATE_REBOOT:
70         x86Reboot();
71         break;
72     case ARCH_POWER_STATE_SHUTDOWN:
73         x86Shutdown();
74         break;
75     case ARCH_POWER_STATE_SLEEP: {
76         x86Sleep();
77         break;
78     }
79     default:
80         return EINVAL;
81     }
82
83     while (1) {
84   ArchStallProcessor();
85  }
86 }
```

File: ./arch/x86/include/idt.h

```
#pragma once

#include <common.h>

/* x86/lowlevel/idt.h - Interrupt Descriptor Table
*
*
*/

/*
* An entry in the IDT. The offset is the address the CPU will jump to,
* and the selector is what segment should be used (i.e. we need to have
* setup a GDT already). The layout of this structure is mandated by the CPU.
*/
struct idt_entry
{
uint16_t isr_offset_low;
uint16_t segment_selector;
uint8_t reserved;
uint8_t type;
uint16_t isr_offset_high;

} __attribute__((packed));

/*
* Used to tell the CPU where the IDT is and how long it is.
* The layout of this structure is mandated by the CPU.
*/
struct idt_ptr
{
uint16_t size;
```

```
size_t location;
} __attribute__((packed));


void x86InitIdt(void);
```

File: ./arch/x86/include/config.h

```
#pragma once

/*
* As this is for x86 (not x86-64), we set the limit to 4GB. On x86-64, we can set
* it larger. This will make the bitmap much larger, but this is no problem on an
* x86-64 system (only ancient x86 systems will have e.g. 4MB of RAM).
*/
#define ARCH_MAX_RAM_KBS (1024 * 1024 * 4)

#define ARCH_PAGE_SIZE 4096

/*
* Non-inclusive of ARCH_USER_AREA_LIMIT
*/
#define ARCH_USER_AREA_BASE 0x08000000
#define ARCH_USER_AREA_LIMIT 0xC0000000

#define ARCH_USER_STACK_BASE 0x08000000
#define ARCH_USER_STACK_LIMIT 0x10000000

/*
* Non-inclusive of ARCH_KRNL_SBRK_LIMIT. Note that we can't use the top 8MB,
* as we use that for recursive mapping.
*/
#define ARCH_KRNL_SBRK_BASE 0xC4000000
#define ARCH_KRNL_SBRK_LIMIT 0xFFC00000
#define ARCH_PROG_LOADER_BASE 0xBFC00000
#define ARCH_PROG_LOADER_ENTRY 0xBFC00000

#define ARCH_MAX_CPU_ALLOWED 16

#undef ARCH_BIG_ENDIAN
#define ARCH_LITTLE_ENDIAN


#include <machine/gdt.h>
#include <machine/idt.h>
#include <machine/tss.h>
#include <machine/regs.h>

typedef struct {
/* Plz keep tss at the top, thread switching assembly needs it */
struct tss* tss;

struct gdt_entry gdt[16];
struct idt_entry idt[256];

struct gdt_ptr gdtr;
struct idt_ptr idtr;

} platform_cpu_data_t;

typedef struct {
size_t p_page_directory; // cr3
size_t* v_page_directory; // what we use to access the tables

} platform_vas_data_t;

typedef struct x86_regs platform_irq_context_t;
```

File: ./arch/x86/include/virtual.h

```
#pragma once

#include <stddef.h>

__attribute__((fastcall)) size_t x86KernelMemoryToPhysical(size_t virtual);
```

File: ./arch/x86/include/pit.h

```
#pragma once

void InitPit(int hertz);
```

File: ./arch/x86/include/dev.h

```
#pragma once

void InitIde(void);
void InitFloppy(void);
```

File: ./arch/x86/include/elf.h

```
#pragma once
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

#define ELF_NIDENT 16

typedef uint16_t Elf32_Half; // Unsigned half int
typedef uint32_t Elf32_Off; // Unsigned offset
typedef uint32_t Elf32_Addr; // Unsigned address
```

```
typedef uint32_t Elf32_Word; // Unsigned int
typedef int32_t Elf32_Sword; // Signed int

struct Elf32_Ehdr
{
uint8_t e_ident[ELF_NIDENT];
Elf32_Half e_type;
Elf32_Half e_machine;
Elf32_Word e_version;
Elf32_Addr e_entry;
Elf32_Off e_phoff;
Elf32_Off e_shoff;
Elf32_Word e_flags;
Elf32_Half e_ehsize;
Elf32_Half e_phentsize;
Elf32_Half e_phnum;
Elf32_Half e_shentsize;
Elf32_Half e_shnum;
Elf32_Half e_shstrndx;

};

enum Elf_Ident
{
EI_MAG0 = 0, // 0x7F
EI_MAG1 = 1, // 'E'
EI_MAG2 = 2, // 'L'
EI_MAG3 = 3, // 'F'
EI_CLASS = 4, // Architecture (32/64)
EI_DATA = 5, // Byte Order
EI_VERSION = 6, // ELF Version
EI_OSABI = 7, // OS Specific
EI_ABIVERSION = 8, // OS Specific
EI_PAD = 9 // Padding
};

#define ELFMAG0 0x7F // e_ident[EI_MAG0]
#define ELFMAG1 'E' // e_ident[EI_MAG1]
#define ELFMAG2 'L' // e_ident[EI_MAG2]
#define ELFMAG3 'F' // e_ident[EI_MAG3]

#define ELFDATA2LSB (1) // Little Endian
#define ELFCLASS32 (1) // 32-bit Architecture

enum Elf_Type
{
ET_NONE = 0, // Unkown Type
ET_REL = 1, // Relocatable File
ET_EXEC = 2 // Executable File
};

#define EM_386 (3) // x86 Machine Type
#define EV_CURRENT (1) // ELF Current Version


struct Elf32_Shdr
{
Elf32_Word sh_name;
Elf32_Word sh_type;
Elf32_Word sh_flags;
Elf32_Addr sh_addr;
Elf32_Off sh_offset;
Elf32_Word sh_size;
Elf32_Word sh_link;
Elf32_Word sh_info;
Elf32_Word sh_addralign;
Elf32_Word sh_entsize;
};

#define SHN_UNDEF 0x0000 // Undefined/Not present
#define SHN_ABS 0xFFF1 // Absolute value

enum ShT_Types
{
SHT_NULL = 0, // Null section
SHT_PROGBITS = 1, // Program information
SHT_SYMTAB = 2, // Symbol table
SHT_STRTAB = 3, // String table
SHT_RELA = 4, // Relocation (w/ addend)
SHT_NOBITS = 8, // Not present in file
SHT_REL = 9, // Relocation (no addend)
};

enum ShT_Attributes
{
SHF_WRITE = 0x01, // Writable section
SHF_ALLOC = 0x02 // Exists in memory
};

struct Elf32_Sym
{
Elf32_Word st_name;
Elf32_Addr st_value;
Elf32_Word st_size;
uint8_t st_info;
uint8_t st_other;
Elf32_Half st_shndx;
};

#define ELF32_ST_BIND(INFO) ((INFO) >> 4)
#define ELF32_ST_TYPE(INFO) ((INFO) & 0x0F)

enum StT_Bindings
{
STB_LOCAL = 0, // Local scope
STB_GLOBAL = 1, // Global scope
STB_WEAK = 2 // Weak, (ie. __attribute__((weak)))
};
```

```
enum StT_Types
{
STT_NOTYPE = 0, // No type
STT_OBJECT = 1, // Variables, arrays, etc.
STT_FUNC = 2 // Methods or functions
};

struct Elf32_Rel
{
Elf32_Addr r_offset;
Elf32_Word r_info;
};

struct Elf32_Rela {
Elf32_Addr r_offset;
Elf32_Word r_info;
Elf32_Sword r_addend;
};

#define ELF32_R_SYM(INFO) ((INFO) >> 8)
#define ELF32_R_TYPE(INFO) ((uint8_t)(INFO))

enum RtT_Types
{
R_386_NONE = 0, // No relocation
R_386_32 = 1, // Symbol + Offset
R_386_PC32 = 2, // Symbol + Offset - Section Offset
R_386_RELATIVE = 8,
};

struct Elf32_Phdr
{
Elf32_Word p_type;
Elf32_Off p_offset;
Elf32_Addr p_vaddr;
Elf32_Addr p_paddr;
Elf32_Word p_filesz;
Elf32_Word p_memsz;
Elf32_Word p_flags;
Elf32_Word p_align;
};

enum PH_Types
{
PHT_NULL = 0,
PHT_LOAD = 1,
PHT_DYNAMIC = 2,
PHT_INTERP = 3,
PHT_NOTE = 4,
PHT_SHLIB = 5,
PHT_PHDIR = 6,
PHT_LOOS = 0x60000000,
PHT_HIOS = 0x6FFFFFFF,
PHT_LOPROC = 0x70000000,
PHT_HIPROC = 0x7FFFFFFF
};

#define ELF32_R_SYM(INFO) ((INFO) >> 8)
#define ELF32_R_TYPE(INFO) ((uint8_t)(INFO))

#define DO_386_32(S, A) ((S) + (A))
#define DO_386_RELATIVE(B, A) ((B) + (A))
#define DO_386_PC32(S, A, P) ((S) + (A) - (P))

#define PF_X 1
#define PF_W 2
#define PF_R 4
```

File: ./arch/x86/include/pic.h

```
#pragma once

#include <stdbool.h>
#include <stdint.h>

#define PIC_IRQ_BASE 32

void InitPic(void);
void SendPicEoi(int irq_num);
bool IsPicIrqSpurious(int irq_num);
void DisablePicLines(uint16_t irq_bitfield);
```

File: ./arch/x86/include/interrupt.h

```
#pragma once

#include <stdbool.h>

bool x86IsReadyForIrqs(void);
void x86MakeReadyForIrqs(void);
```

File: ./arch/x86/include/regs.h

```
#pragma once
#include <common.h>

struct x86_regs
{
/*
* The registers that are pushed to us in x86/lowlevel/trap.s
*
* SS is the first thing pushed, and thus the last to be popped
* GS is the last thing pushed, and thus the first to be popped
```

```
*/
size_t gs, fs, es, ds;
size_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
size_t int_no, err_code;
size_t eip, cs, eflags, useresp, ss;
};
```

File: ./arch/x86/include/tss.h

```
#pragma once

/* x86/lowlevel/tss.h - Task State Segment
*
*
*/

#include <common.h>

/*
* The task state segment was designed to store information about a task so
* that task switching could be done in hardware. We do not use it for this purpose,
* instead only using it to set the stack correctly after a user -> kernel switch.
*
* The layout of this structure is mandated by the CPU.
*/
struct tss
{
uint16_t link;
uint16_t reserved0;
uint32_t esp0;
uint16_t ss0;
uint16_t reserved1;
uint32_t esp1;
uint16_t ss1;
uint16_t reserved2;
uint32_t esp2;
uint16_t ss2;
uint16_t reserved3;
uint32_t cr3;
uint32_t eip;
uint32_t eflags;
uint32_t eax;
uint32_t ecx;
uint32_t edx;
uint32_t ebx;
uint32_t esp;
uint32_t ebp;
uint32_t esi;
uint32_t edi;
uint16_t es;
uint16_t reserved4;
uint16_t cs;
uint16_t reserved5;
uint16_t ss;
uint16_t reserved6;
uint16_t ds;
uint16_t reserved7;
uint16_t fs;
uint16_t reserved8;
uint16_t gs;
uint16_t reserved9;
uint16_t ldtr;
uint16_t reserved10;
uint16_t reserved11;
uint16_t iopb;

} __attribute__((packed));

void x86InitTss(void);
```

File: ./arch/x86/include/gdt.h

```
#pragma once

#include <common.h>

/*
* An entry in the GDT table. The layout of this structure is mandated by the CPU.
*/
struct gdt_entry
{
uint16_t limit_low;
uint16_t base_low;
uint8_t base_middle;
uint8_t access;
uint8_t flags_and_limit_high;
uint8_t base_high;

} __attribute__((packed));

/*
* Describes the GDT address and size. We use the address of this structure
* to tell the CPU where the GDT is. The layout of this structure is mandated by the CPU.
*/
struct gdt_ptr
{
uint16_t size;
size_t location;
} __attribute__((packed));

struct tss;

void x86InitGdt(void);
uint16_t x86AddTssToGdt(struct tss* tss);
```

File: ./arch/x86/include/portio.h

```
#pragma once

/*
* x86/portio.h - Port Input / Output
*
*
* On the x86, a lot of older hardware is accessed using IO ports. A port has an
* address from 0x0000 to 0xFFFF, and can be read from and written to using special
* instructions.
*
* We are going to inline these functions, as they are all single instructions.
*
*/

#include <common.h>
#include <log.h>

/*
* Writing to ports
*/
always_inline void outb(uint16_t port, uint8_t value)
{
asm volatile ("outb %0, %1" : : "a"(value), "Nd"(port));
}

always_inline void outw(uint16_t port, uint16_t value)
{
asm volatile ("outw %0, %1" : : "a"(value), "Nd"(port));
}

always_inline void outl(uint16_t port, uint32_t value)
{
asm volatile ("outl %0, %1" : : "a"(value), "Nd"(port));
}

/*
* Reading from ports
*/
always_inline uint8_t inb(uint16_t port)
{
uint8_t value;
asm volatile ("inb %1, %0"
: "=a"(value)
: "Nd"(port));
return value;
}

always_inline uint16_t inw(uint16_t port)
{
uint16_t value;
asm volatile ("inw %1, %0"
: "=a"(value)
: "Nd"(port));
return value;
}

always_inline uint32_t inl(uint16_t port)
{
uint32_t value;
asm volatile ("inl %1, %0"
: "=a"(value)
: "Nd"(port));
return value;
}
```

File: ./arch/x86/driver.ld

```
OUTPUT_FORMAT("elf32-i386")

SECTIONS
{
. = 0xD0000000;

.text BLOCK(4096) : ALIGN(4096)
{
*(.text)
*(.rodata)
}

.data BLOCK(4096) : ALIGN(4096)
{
*(.data)
}

.lockedtext BLOCK(4096) : ALIGN(4096)
{
*(.lockedtext)
*(.lockedrodata)
}

.lockeddata BLOCK(4096) : ALIGN(4096)
{
*(.lockeddata)
}

.bss BLOCK(4096) : ALIGN(4096)
{
*(COMMON)
*(.bss)
*(.bootstrap_stack)
}
```

/DISCARD/ :
{
*(.comment)
}
}

File: ./arch/x86/dev/pic.c

```
#include <machine/pic.h>
#include <machine/portio.h>
#include <arch.h>

/*
* x86/dev/pic.c - Programmable Interrupt Controller
*
* The PIC controlls the hardare raised interrupts (IRQs), i.e. those from
* 'external' devices such as the keyboard, system timer, disk, etc. Hence the
* PIC must be configured before any external interrupts can be seen by the CPU.
*
* There are two quirks to certain IRQs. The first comes about by the fact that
* each system actually has two PICs - the primary PIC handling IRQs 0-7, and the
* secondary PIC handling IRQs 8-15. They are connected (cascaded) to each other,
* using IRQ 2 for communication. Thus, an IRQ 2 will get to the CPU.
*
* The other quirk is the 'spurious interrupt', which can occur on IRQ 7 or 15.
* See pic_is_spurious for more details.
*/

#define PIC1_COMMAND    0x20
#define PIC1_DATA       0x21
#define PIC2_COMMAND    0xA0
#define PIC2_DATA       0xA1

#define PIC_EOI         0x20
#define PIC_REG_ISR     0x0B

#define ICW1_ICW4       0x01
#define ICW1_INIT       0x10
#define ICW4_8086       0x01


/*
* Delay for a short period of time, for use in betwwen IO calls to the PIC.
* This is required as some PICs have a hard time keeping up with the speed
* of modern CPUs (the original PIC was introduced in 1976!).
*/
static void IoWait(void) {

}

/*
* Read an internal PIC register.
*/
static uint16_t ReadPicReg(int ocw3) {
    outb(PIC1_COMMAND, ocw3);
    outb(PIC2_COMMAND, ocw3);
    return ((uint16_t) inb(PIC2_COMMAND) << 8) | inb(PIC1_COMMAND);
}

/*
* Due to a race condition between the PIC and the CPU, we sometimes get a
* 'spurious' interrupt sent to the CPU on IRQ 7 or 15. If an IRQ 7 or 15
* arrives, we need to check if it an actual interrupt or a spurious interrupt.
* Distinguishing them is important - spurious IRQs may cause drivers to misbehave,
* and we don't need to send an EOI after a spurious interrupt.
*/
bool IsPicIrqSpurious(int irq_num) {
    if (irq_num == PIC_IRQ_BASE + 7) {
        uint16_t isr = ReadPicReg(PIC_REG_ISR);
        if (!(isr & (1 << 7))) {
            return true;
        }

    } else if (irq_num == PIC_IRQ_BASE + 15) {
        uint16_t isr = ReadPicReg(PIC_REG_ISR);
        if (!(isr & (1 << 15))) {
            /*
            * It is spurious, but the primary PIC doesn't know that, as it came
            * from the secondary PIC. So only send an EOI to the primary PIC.
            */
            outb(PIC1_COMMAND, PIC_EOI);
            return true;
        }
    }

    return false;
}

/*
* Acknowledge the previous interrupt. We will not receive any interrupts of
* the same type until we have acknowledged it.
*/
void SendPicEoi(int irq_num) {
    if (irq_num >= PIC_IRQ_BASE + 8) {
        outb(PIC2_COMMAND, PIC_EOI);
    }

    outb(PIC1_COMMAND, PIC_EOI);
}

/*
* Change which interrupt numbers are used by the IRQs. They will initially
* use interrupts 0 through 15, which isn't very good as it conflicts with the
* interrupt numbers for the CPU exceptions.
*/
static void RemapPic(int offset) {
    uint8_t mask1 = inb(PIC1_DATA);
    uint8_t mask2 = inb(PIC2_DATA);

    outb(PIC1_COMMAND, ICW1_INIT | ICW1_ICW4);
    IoWait();
    outb(PIC2_COMMAND, ICW1_INIT | ICW1_ICW4);
    IoWait();
    outb(PIC1_DATA, offset);
    IoWait();
    outb(PIC2_DATA, offset + 8);
    IoWait();
    outb(PIC1_DATA, 4);
    IoWait();
    outb(PIC2_DATA, 2);
    IoWait();

    outb(PIC1_DATA, ICW4_8086);
    IoWait();
    outb(PIC2_DATA, ICW4_8086);
    IoWait();
```

```
119
120     outb(PIC1_DATA, mask1);
121     outb(PIC2_DATA, mask2);
122 }
123
124 /**
125  * Set which IRQ numbers are disabled. Overwrites the previous call to this function completely
126  * (i.e. this is an 'equals' operation, not an 'and' or 'or'.)
127  *
128  * @param irq_bitfield A bitfield of IRQs, where the lowest bit corresponds to IRQ0. For each bit,
129  *                     a zero will enable the interrupt, and a one will disable the interrupt.
130  *                     To disable all lines, specify 0xFFFF. To enable all lines, specify 0x0000.
131  */
132 void DisablePicLines(uint16_t irq_bitfield) {
133     static uint16_t prev = 0xFFFF;      // we initially set it to 0, so this will be different
134
135     if (prev == irq_bitfield) {
136         return;
137     }
138
139     outb(PIC1_DATA, irq_bitfield & 0xFF);
140     outb(PIC2_DATA, irq_bitfield >> 8);
141     prev = irq_bitfield;
142 }
143
144 /*
145 * Initialise the PIC.
146 */
147 void InitPic(void) {
148     /*
149     * Remap the PIC so that interrupts 0-15 are mapped to 32-47.
150     * This way we don't have conflicts with the CPU exceptions which are
151     * hard-wired to these interrupt numbers.
152     */
153     RemapPic(PIC_IRQ_BASE);
154
155     /*
156     * Now we can disable all masks, allowing interrupts to reach the CPU.
157     */
158     DisablePicLines(0x0000);
159 }
```

File: ./arch/x86/dev/init.c

```
1  #include <machine/dev.h>
2  #include <machine/portio.h>
3  #include <driver.h>
4  #include <thread.h>
5  #include <panic.h>
6  #include <log.h>
7  #include <virtual.h>
8
9  static size_t Loadx86Driver(const char* filename, const char* init) {
10     int res = RequireDriver(filename);
11     if (res != 0) {
12         PanicEx(PANIC_REQUIRED_DRIVER_NOT_FOUND, filename);
13     }
14
15     size_t addr = GetSymbolAddress(init);
16     if (addr == 0) {
17         PanicEx(PANIC_REQUIRED_DRIVER_MISSING_SYMBOL, filename);
18     }
19
20     return addr;
21 }
22
23 static void LoadSlowDriversInBackground(void*) {
24     /*
25      * To make the OS boot faster, we'll load the less critical, and slower
26      * drivers in a new thread. This means we can continue initialising the rest
27      * of the OS while drivers load.
28      */
29     ((void(*)()) (Loadx86Driver("sys:/acpi.sys", "InitAcpica")))();
30 }
31
32 void ArchInitDev(bool fs) {
33     if (!fs) {
34         InitIde();
35         //InitFloppy();
36
37     } else {
38         ((void(*)()) (Loadx86Driver("sys:/vesa.sys", "InitVesa")))();
39         ((void(*)()) (Loadx86Driver("sys:/ps2.sys", "InitPs2")))();
40         CreateThread(LoadSlowDriversInBackground, NULL, GetVas(), "drvloader");
41     }
42 }
```

File: ./arch/x86/dev/floppy.c

```
1   #include <common.h>
2   #include <semaphore.h>
3   #include <log.h>
4   #include <thread.h>
5   #include <vfs.h>
6   #include <string.h>
7   #include <transfer.h>
8   #include <assert.h>
9   #include <irq.h>
10  #include <errno.h>
11  #include <machine/pic.h>
12  #include <machine/portio.h>
13  #include <heap.h>
14  #include <virtual.h>
15  #include <stdlib.h>
16  #include <sys/stat.h>
17  #include <dirent.h>
18  #include <irql.h>
19  #include <diskutil.h>
20
21  #define CYLINDER_SIZE (512 * 18 * 2)
22
23  #define FLOPPY_DOR      2
24  #define FLOPPY_MSR      4
25  #define FLOPPY_FIFO     5
26  #define FLOPPY_CCR      7
27
28  #define CMD_SPECIFY     0x03
29  #define CMD_READ        0x06
30  #define CMD_RECALIBRATE 0x07
31  #define CMD_SENSE_INT   0x08
32  #define CMD_SEEK        0x0F
33  #define CMD_CONFIGURE   0x13
34
35  struct semaphore* floppy_lock = NULL;
```

```
36
37 struct floppy_data {
38     int disk_num;
39     uint8_t* cylinder_buffer;
40     uint8_t* cylinder_zero;
41     size_t base;
42     struct disk_partition_helper partitions;
43     int stored_cylinder;
44     bool got_cylinder_zero;
45 };
46
47 static void FloppyWriteCommand(struct floppy_data* flp, int cmd) {
48     int base = flp->base;
49
50     for (int i = 0; i < 60; ++i) {
51         SleepMilli(10);
52         if (inb(base + FLOPPY_MSR) & 0x80) {
53             outb(base + FLOPPY_FIFO, cmd);
54             return;
55         }
56     }
57
58     LogWriteSerial("floppy_write_cmd: timeout\n");
59 }
60
61 static uint8_t FloppyReadData(struct floppy_data* flp) {
62     int base = flp->base;
63     for (int i = 0; i < 60; ++i) {
64         SleepMilli(10);
65         if (inb(base + FLOPPY_MSR) & 0x80) {
66             return inb(base + FLOPPY_FIFO);
67         }
68     }
69
70     LogWriteSerial("floppy_read_data: timeout\n");
71     return 0;
72 }
73
74 static void FloppyCheckInterrupt(struct floppy_data* flp, int* st0, int* cyl) {
75     LogWriteSerial("FloppyCheckInterrupt\n");
76
77     FloppyWriteCommand(flp, CMD_SENSE_INT);
78     *st0 = FloppyReadData(flp);
79     *cyl = FloppyReadData(flp);
80 }
81
82 /*
83  * The state can be 0 (off), 1 (on) or 2 (currently on, but will shortly be turned off).
84  */
85
86 static volatile int floppy_motor_state = 0;
87 static volatile int floppy_motor_ticks = 0;
88
89 static void FloppyMotor(struct floppy_data* flp, bool state) {
90     LogWriteSerial("FloppyMotor\n");
91
92     int base = flp->base;
93
94     if (state) {
95         if (!floppy_motor_state) {
96             outb(base + FLOPPY_DOR, 0x1C);
97             SleepMilli(150);
98         }
99         floppy_motor_state = 1;
100
101     } else {
102         floppy_motor_state = 2;
103         floppy_motor_ticks = 1000;
104     }
105 }
106
107 static volatile bool floppy_got_irq = false;
108
109 static void FloppyIrqWait() {
110     LogWriteSerial("FloppyIrqWait\n");
111
112     /*
113     * Wait for the interrupt to come. If it doesn't the system will probably
114     * lockup.
115     */
116     while (!floppy_got_irq) {
117         SleepMilli(10);
118     }
119
120     /*
121     * Clear it for next time.
122     */
123     floppy_got_irq = false;
124 }
125
126 static int FloppyIrqHandler(struct x86_regs*) {
127     LogWriteSerial("FloppyIrqHandler\n");
128
129     floppy_got_irq = true;
130     return 0;
131 }
132
133 static void FloppyMotorControlThread(void*) {
134     LogWriteSerial("FloppyMotorControlThread\n");
135
136     while (1) {
137         SleepMilli(50);
138         if (floppy_motor_state == 2) {
139             floppy_motor_ticks -= 50;
140             if (floppy_motor_ticks <= 0) {
141                 /*
142                 * Actually turn off the motor.
143                 */
144                 //outb(0x3F0 + FLOPPY_DOR, 0x0C);
145                 //loppy_motor_state = 0;
146             }
147         }
148     }
149 }
150
151 /*
152 * Move to cylinder 0.
153 */
154 static int FloppyCalibrate(struct floppy_data* flp) {
155     LogWriteSerial("FloppyCalibrate\n");
156
157     int st0 = -1;
158     int cyl = -1;
159
160     FloppyMotor(flp, true);
161
162     for (int i = 0; i < 10; ++i) {
163         FloppyWriteCommand(flp, CMD_RECALIBRATE);
164         FloppyWriteCommand(flp, 0);
165
```

```
        FloppyIrqWait();
        FloppyCheckInterrupt(flp, &st0, &cyl);

        if (st0 & 0xC0) {
            continue;
        }

        if (cyl == 0) {
            FloppyMotor(flp, false);
            return 0;
        }
    }

    LogDeveloperWarning("couldn't calibrate floppy\n");
    FloppyMotor(flp, false);
    return EIO;
}

static void FloppyConfigure(struct floppy_data* flp) {
    LogWriteSerial("FloppyConfigure\n");

    FloppyWriteCommand(flp, CMD_CONFIGURE);
    FloppyWriteCommand(flp, 0x00);
    FloppyWriteCommand(flp, 0x08);
    FloppyWriteCommand(flp, 0x00);
}

static int FloppyReset(struct floppy_data* flp) {
    LogWriteSerial("FloppyReset\n");

    int base = flp->base;
    outb(base + FLOPPY_DOR, 0x00);
    outb(base + FLOPPY_DOR, 0x0C);

    FloppyIrqWait();

    for (int i = 0; i < 4; ++i) {
        int st0, cyl;
        FloppyCheckInterrupt(flp, &st0, &cyl);
    }

    outb(base + FLOPPY_CCR, 0x00);

    FloppyMotor(flp, true);

    FloppyWriteCommand(flp, CMD_SPECIFY);
    FloppyWriteCommand(flp, 0xDF);
    FloppyWriteCommand(flp, 0x02);

    SleepMilli(300);
    FloppyConfigure(flp);
    SleepMilli(300);
    FloppyMotor(flp, false);

    int res = FloppyCalibrate(flp);
    LogWriteSerial("FloppyReset: res = %d\n", res);
    return res;
}

static int FloppySeek(struct floppy_data* flp, int cylinder, int head) {
    LogWriteSerial("FloppySeek\n");

    int st0, cyl;
    FloppyMotor(flp, true);

    for (int i = 0; i < 10; ++i) {
        FloppyWriteCommand(flp, CMD_SEEK);
        FloppyWriteCommand(flp, head << 2);
        FloppyWriteCommand(flp, cylinder);

        FloppyIrqWait();
        FloppyCheckInterrupt(flp, &st0, &cyl);

        if (st0 & 0xC0) {
            continue;
        }

        if (cyl == cylinder) {
            FloppyMotor(flp, false);
            return 0;
        }
    }

    LogWriteSerial("couldn't seek floppy\n");
    FloppyMotor(flp, false);
    return EIO;
}

static void FloppyDmaInit(void) {
    LogWriteSerial("FloppyDmaInit\n");

    /*
    * Put the data at *physical address* 0x10000. The address can be anywhere
    * under 24MB that doesn't cross a 64KB boundary. We choose this location as
    * it should be unused as this is where the temporary copy of the kernel was
    * stored during boot.
    */
    uint32_t addr = (uint32_t) 0x10000;

    /*
    * We must give the DMA the actual count minus 1.
    */
    int count = 0x4800 - 1;

    /*
    * Send some magical stuff to the DMA controller.
    */
    outb(0x0A, 0x06);
    outb(0x0C, 0xFF);
    outb(0x04, (addr >> 0) & 0xFF);
    outb(0x04, (addr >> 8) & 0xFF);
    outb(0x81, (addr >> 16) & 0xFF);
    outb(0x0C, 0xFF);
    outb(0x05, (count >> 0) & 0xFF);
    outb(0x05, (count >> 8) & 0xFF);
    outb(0x0B, 0x46);
    outb(0x0A, 0x02);
}

static int FloppyDoCylinder(struct floppy_data* flp, int cylinder) {
    LogWriteSerial("FloppyDoCylinder\n");

    /*
    * Move both heads to the correct cylinder.
    */
    if (FloppySeek(flp, cylinder, 0) != 0) return EIO;
    if (FloppySeek(flp, cylinder, 1) != 0) return EIO;

    /*
    * This time, we'll try up to 20 times.
```

```
    */
    for (int i = 0; i < 20; ++i) {
        LogWriteSerial("READ ATTEMPT %d\n", i + 1);
        FloppyMotor(flp, true);

        if (i % 5 == 3) {
            if (i % 10 == 8) {
                LogWriteSerial("resetting floppy!\n");
                FloppyReset(flp);
                FloppyMotor(flp, true);
            }

            FloppyCalibrate(flp);

            if (FloppySeek(flp, cylinder, 0) != 0) return EIO;
            if (FloppySeek(flp, cylinder, 1) != 0) return EIO;
        }

        FloppyDmaInit();

        SleepMilli(100);

        /*
        * Send the read command.
        */
        FloppyWriteCommand(flp, CMD_READ | 0xC0);
        FloppyWriteCommand(flp, 0);
        FloppyWriteCommand(flp, cylinder);
        FloppyWriteCommand(flp, 0);
        FloppyWriteCommand(flp, 1);
        FloppyWriteCommand(flp, 2);
        FloppyWriteCommand(flp, 18);
        FloppyWriteCommand(flp, 0x1B);
        FloppyWriteCommand(flp, 0xFF);

        FloppyIrqWait();

        /*
        * Read back some status information, some of which is very mysterious.
        */
        uint8_t st0, st1, st2, rcy, rhe, rse, bps;
        st0 = FloppyReadData(flp);
        st1 = FloppyReadData(flp);
        st2 = FloppyReadData(flp);
        rcy = FloppyReadData(flp);
        rhe = FloppyReadData(flp);
        rse = FloppyReadData(flp);
        bps = FloppyReadData(flp);

        /*
        * Check for errors. More tests can be done, but it would make the code
        * even longer.
        */
        if (st0 & 0xC0) {
            static const char * status[] = { 0, "error", "invalid command", "drive not ready" };
            LogWriteSerial("floppy_do_sector: status = %s\n", status[st0 >> 6]);
            LogWriteSerial("st0 = 0x%X, st1 = 0x%X, st2 = 0x%X, bps = %d\n", st0, st1, st2, bps);
            continue;
        }
        if (st1 & 0x80) {
            continue;
        }
        if (st0 & 0x8) {
            continue;
        }
        if (st1 & 0x20) {
            continue;
        }
        if (st1 & 0x10) {
            continue;
        }
        if (bps != 2) {
            continue;
        }

        (void) st2;
        (void) rcy;
        (void) rhe;
        (void) rse;

        FloppyMotor(flp, false);
        return 0;
    }

    LogWriteSerial("couldn't read floppy\n");
    FloppyMotor(flp, false);
    return EIO;
}



static int FloppyIo(struct floppy_data* flp, struct transfer* io) {
    LogWriteSerial("FloppyIo\n");
    EXACT_IRQL(IRQL_STANDARD);

    if (io->direction == TRANSFER_WRITE) {
        return EROFS;
    }

    int lba = io->offset / 512;
    int count = io->length_remaining / 512;

    if (io->offset % 512 != 0) {
        return EINVAL;
    }
    if (io->length_remaining % 512 != 0) {
        return EINVAL;
    }
    if (count <= 0 || count > 0xFF || lba < 0 || lba >= 2880) {
        return EINVAL;
    }

    AcquireMutex(floppy_lock, -1);

next_sector:;
    /*
    * Floppies use CHS (cylinder, head, sector) for addressing sectors instead of
    * LBA (linear block addressing). Hence we need to convert to CHS.
    *
    * Head: which side of the disk it is on (i.e. either the top or bottom)
    * Cylinder: which 'slice' (cylinder) of the disk we should look at
    * Sector: which 'ring' (sector) of that cylinder we should look at
    *
    * Note that sector is 1-based, whereas cylinder and head are 0-based.
    * Don't ask why.
    */
    int head = (lba % (18 * 2)) / 18;
    int cylinder = (lba / (18 * 2));
    int sector = (lba % 18) + 1;

```

```
    /*
    * Cylinder 0 has some commonly used data, so cache it seperately for improved
    * speed.
    */
    if (cylinder == 0) {
        if (!flp->got_cylinder_zero) {
            flp->got_cylinder_zero = true;

            int error = FloppyDoCylinder(flp, cylinder);

            if (error != 0) {
                ReleaseMutex(floppy_lock);
                return error;
            }

            memcpy(flp->cylinder_zero, flp->cylinder_buffer, 0x4800);

            /*
            * Must do this as we trashed the cached cylinder.
            * Luckily we only ever need to do this once.
            */
            flp->stored_cylinder = cylinder;
        }

        PerformTransfer(flp->cylinder_zero + (512 * (sector - 1 + head * 18)), io, 512);

    } else {
        if (cylinder != flp->stored_cylinder) {
            int error = FloppyDoCylinder(flp, cylinder);

            if (error != 0) {
                ReleaseMutex(floppy_lock);
                return error;
            }
        }

        PerformTransfer(flp->cylinder_buffer + (512 * (sector - 1 + head * 18)), io, 512);
        flp->stored_cylinder = cylinder;
    }

    /*
    * Read only read one sector, so we need to repeat the process if multiple sectors
    * were requested. We could just do a larger copy above, but this is a bit simpler,
    * as we don't need to worry about whether the entire request is on cylinder or not.
    */
    if (count > 0) {
        ++lba;
        --count;
        goto next_sector;
    }

    ReleaseMutex(floppy_lock);
    return 0;
}

static bool IsSeekable(struct vnode*) {
    return true;
}

static int ReadWrite(struct vnode* node, struct transfer* io) {
    return FloppyIo(node->data, io);
}

static int Create(struct vnode* node, struct vnode** partition, const char* name, int, mode_t) {
    AcquireMutex(floppy_lock, -1);
    struct floppy_data* flp = node->data;
    int res = DiskCreateHelper(&flp->partitions, partition, name);
    ReleaseMutex(floppy_lock);
    return res;
}

static int Follow(struct vnode* node, struct vnode** output, const char* name) {
    AcquireMutex(floppy_lock, -1);
    struct floppy_data* flp = node->data;
    int res = DiskFollowHelper(&flp->partitions, output, name);
    ReleaseMutex(floppy_lock);
    return res;
}

static uint8_t DirentType(struct vnode*) {
    return DT_BLK;
}

static int Stat(struct vnode*, struct stat* st) {
    st->st_mode = S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO;
    st->st_atime = 0;
    st->st_blksize = 512;
    st->st_blocks = 2880;
    st->st_ctime = 0;
    st->st_dev = 0xBABECAFE;
    st->st_gid = 0;
    st->st_ino = 0xCAFEBABE;
    st->st_mtime = 0;
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = 1024 * 1440;
    st->st_uid = 0;
    return 0;
}

static int Close(struct vnode*) {
    return 0;
}

static const struct vnode_operations dev_ops = {
    .is_seekable    = IsSeekable,
    .read           = ReadWrite,
    .write          = ReadWrite,
    .close          = Close,
    .create         = Create,
    .follow         = Follow,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

void InitFloppy(void) {
    floppy_lock = CreateMutex("floppy");

    CreateThread(FloppyMotorControlThread, NULL, GetVas(), "flpmotor");

    for (int i = 0; i < 1; ++i) {
        struct vnode* node = CreateVnode(dev_ops);
        struct floppy_data* flp = AllocHeap(sizeof(struct floppy_data));

        RegisterIrqHandler(PIC_IRQ_BASE + 6, FloppyIrqHandler);

        flp->disk_num          = 0;
        flp->base              = 0x3F0;
        flp->cylinder_buffer   = (uint8_t*) MapVirt(0, 0, CYLINDER_SIZE, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
        flp->cylinder_zero     = (uint8_t*) MapVirt(0, 0, CYLINDER_SIZE, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
```

```
        flp->got_cylinder_zero = false;
        flp->stored_cylinder   = -1;
        FloppyReset(flp);

        InitDiskPartitionHelper(&flp->partitions);

        node->data = flp;
        AddVfsMount(node, GenerateNewRawDiskName(DISKUTIL_TYPE_FLOPPY));
        CreateDiskPartitions(CreateOpenFile(node, 0, 0, true, true));
    }
}
```

File: ./arch/x86/dev/ide.c

```
#include <common.h>
#include <semaphore.h>
#include <log.h>
#include <thread.h>
#include <vfs.h>
#include <transfer.h>
#include <assert.h>
#include <errno.h>
#include <machine/portio.h>
#include <heap.h>
#include <virtual.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <irql.h>
#include <diskutil.h>

#define MAX_TRANSFER_SIZE (1024 * 16)

struct semaphore* ide_lock = NULL;

struct ide_data {
    int disk_num;
    unsigned int sector_size;
    uint64_t total_num_sectors;
    uint16_t* transfer_buffer;
    size_t primary_base;
    size_t primary_alternative;
    size_t secondary_base;
    size_t secondary_alternative;
    size_t busmaster_base;

    struct disk_partition_helper partitions;
};

int IdeCheckError(struct ide_data* ide) {
    uint16_t base = ide->disk_num >= 2 ? ide->secondary_base : ide->primary_base;

    uint8_t status = inb(base + 0x7);
    if (status & 0x01) {
        return EIO;

    } else if (status & 0x20) {
        return EIO;

    } else if (!(status & 0x08)) {
        return EIO;
    }

    return 0;
}

int IdePoll(struct ide_data* ide) {
    uint16_t base = ide->disk_num >= 2 ? ide->secondary_base : ide->primary_base;
    uint16_t alt_status_reg = ide->disk_num >= 2 ? ide->secondary_alternative : ide->primary_alternative;

    /*
    * Delay for a moment by reading the alternate status register.
    */
    for (int i = 0; i < 4; ++i) {
        inb(alt_status_reg);
    }

    /*
    * Wait for the device to not be busy. We have a timeout in case the
    * device is faulty, we don't want to be in an endless loop and freeze
    * the kernel.
    */
    int timeout = 0;
    while (inb(base + 0x7) & 0x80) {
        if (timeout > 975) {
            SleepMilli(10);
        }
        if (timeout++ > 1000) {
            return EIO;
        }
    }

    return 0;
}

/*
* Read or write the primary ATA drive on the first controller. We use LBA28,
* so we are limited to a 28 bit sector number (i.e. disks up to 128GB in size)
*/
static int IdeIo(struct ide_data* ide, struct transfer* io) {
    EXACT_IRQL(IRQL_STANDARD);
    int disk_num = ide->disk_num;

    /*
    * IDE devices do not contain an (accessible) disk buffer in PIO mode, as
    * they transfer data through the IO ports. Hence we must read/write into
    * this buffer first, and then move it safely to the destination.
    * (we could use DMA instead, but PIO is simpler)
    *
    * Allow up to 4KB sector sizes. Make sure there is enough room on the
    * stack to handle this.
    */
    uint16_t* buffer = ide->transfer_buffer;

    int sector = io->offset / ide->sector_size;
    int count = io->length_remaining / ide->sector_size;

    if (io->offset % ide->sector_size != 0) {
        return EINVAL;
    }
    if (io->length_remaining % ide->sector_size != 0) {
        return EINVAL;
    }
    if (count <= 0 || sector < 0 || sector > 0xFFFFFFF || (uint64_t) sector + count >= ide->total_num_sectors) {
        return EINVAL;
    }
```

```
113
114     AcquireSemaphore(ide_lock, -1);
115
116     uint16_t base = disk_num >= 2 ? ide->secondary_base : ide->primary_base;
117     uint16_t dev_ctrl_reg = disk_num >= 2 ? ide->secondary_alternative : ide->primary_alternative;
118
119     int max_sectors_at_once = MAX_TRANSFER_SIZE / ide->sector_size;
120     if (max_sectors_at_once > 255) {
121         // hardware limitation
122         max_sectors_at_once = 255;
123     }
124
125     while (count > 0) {
126         int sectors_in_this_transfer = count > max_sectors_at_once ? max_sectors_at_once : count;
127
128         if (io->direction == TRANSFER_WRITE) {
129             PerformTransfer(buffer, io, ide->sector_size);
130         }
131
132         /*
133         * Send a whole heap of flags and the high 4 bits of the LBA to the controller.
134         */
135         outb(base + 0x6, 0xE0 | ((disk_num & 1) << 4) | ((sector >> 24) & 0xF));
136
137         /*
138         * Disable interrupts, we are going to use polling.
139         */
140         outb(dev_ctrl_reg, 2);
141
142         /*
143         * May not be needed, but it doesn't hurt to do it.
144         */
145         outb(base + 0x1, 0x00);
146
147         /*
148         * Send the number of sectors, and the sector's LBA.
149         */
150         outb(base + 0x2, sectors_in_this_transfer);
151         outb(base + 0x3, (sector >> 0) & 0xFF);
152         outb(base + 0x4, (sector >> 8) & 0xFF);
153         outb(base + 0x5, (sector >> 16) & 0xFF);
154
155         /*
156         * Send either the read or write command.
157         */
158         outb(base + 0x7, io->direction == TRANSFER_WRITE ? 0x30 : 0x20);
159
160         /*
161         * Wait for the data to be ready.
162         */
163         IdePoll(ide);
164
165         /*
166         * Read/write the data from/to the disk using ports.
167         */
168         if (io->direction == TRANSFER_WRITE) {
169             for (int c = 0; c < sectors_in_this_transfer; ++c) {
170                 if (c != 0) {
171                     PerformTransfer(buffer, io, ide->sector_size);
172                     IdePoll(ide);
173                 }
174
175                 for (uint64_t i = 0; i < ide->sector_size / 2; ++i) {
176                     outw(base + 0x00, buffer[i]);
177                 }
178             }
179             IdePoll(ide);
180
181             /*
182             * We need to flush the disk's cache if we are writing.
183             */
184             outb(base + 0x7, 0xE7);
185             IdePoll(ide);
186
187         } else {
188             int err = IdeCheckError(ide);
189             if (err) {
190                 ReleaseSemaphore(ide_lock);
191                 return err;
192             }
193
194             for (int c = 0; c < sectors_in_this_transfer; ++c) {
195                 if (c != 0) {
196                     IdePoll(ide);
197                 }
198
199                 for (uint64_t i = 0; i < ide->sector_size / 2; ++i) {
200                     buffer[i] = inw(base + 0x00);
201                 }
202                 PerformTransfer(buffer, io, ide->sector_size);
203             }
204         }
205
206         /*
207         * Get ready for the next part of the transfer.
208         */
209         count -= sectors_in_this_transfer;
210         sector += sectors_in_this_transfer;
211     }
212
213     ReleaseSemaphore(ide_lock);
214
215     return 0;
216 }
217
218 static int IdeGetNumSectors(struct ide_data* ide) {
219     AcquireSemaphore(ide_lock, -1);
220
221     uint16_t base = ide->disk_num >= 2 ? ide->secondary_base : ide->primary_base;
222
223     /*
224     * Select the correct drive.
225     */
226     outb(base + 0x6, 0xE0 | ((ide->disk_num & 1) << 4));
227
228     /*
229     * Send the READ NATIVE MAX ADDRESS command, which will return the size
230     * of disk in sectors.
231     */
232     outb(base + 0x7, 0xF8);
233
234     IdePoll(ide);
235
236     /*
237     * The outputs are in the same registers we use to put the LBA
238     * when we read/write from the disk.
239     */
240     int sectors = 0;
241     sectors |= (int) inb(base + 0x3);
242     sectors |= ((int) inb(base + 0x4)) << 8;
```

```
    sectors |= ((int) inb(base + 0x5)) << 16;
    sectors |= ((int) inb(base + 0x6) & 0xF) << 24;

    ReleaseSemaphore(ide_lock);

    return sectors;
}

static bool IsSeekable(struct vnode*) {
    return true;
}

static int ReadWrite(struct vnode* node, struct transfer* io) {
    return IdeIo(node->data, io);
}

static int Create(struct vnode* node, struct vnode** partition, const char* name, int, mode_t) {
    AcquireSemaphore(ide_lock, -1);
    struct ide_data* ide = node->data;
    int res = DiskCreateHelper(&ide->partitions, partition, name);
    ReleaseSemaphore(ide_lock);
    return res;
}

static int Follow(struct vnode* node, struct vnode** output, const char* name) {
    AcquireSemaphore(ide_lock, -1);
    struct ide_data* ide = node->data;
    int res = DiskFollowHelper(&ide->partitions, output, name);
    ReleaseSemaphore(ide_lock);
    return res;
}

static uint8_t DirentType(struct vnode*) {
    return DT_BLK;
}

static int Stat(struct vnode* node, struct stat* st) {
    struct ide_data* ide = node->data;

    st->st_mode = S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO;
    st->st_atime = 0;
    st->st_blksize = ide->sector_size;
    st->st_blocks = ide->total_num_sectors;
    st->st_ctime = 0;
    st->st_dev = 0xBABECAFE;
    st->st_gid = 0;
    st->st_ino = 0xCAFEBABE;
    st->st_mtime = 0;
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = ide->sector_size * ide->total_num_sectors;
    st->st_uid = 0;

    return 0;
}

static int Close(struct vnode*) {
    return 0;
}

static const struct vnode_operations dev_ops = {
    .is_seekable    = IsSeekable,
    .read           = ReadWrite,
    .write          = ReadWrite,
    .close          = Close,
    .create         = Create,
    .follow         = Follow,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

void InitIde(void) {
    ide_lock = CreateMutex("ide");

    for (int i = 0; i < 1; ++i) {
        struct vnode* node = CreateVnode(dev_ops);
        struct ide_data* ide = AllocHeap(sizeof(struct ide_data));

        ide->disk_num               = 0;
        ide->primary_base           = 0x1F0;
        ide->secondary_base         = 0x170;
        ide->primary_alternative    = 0x3F6;
        ide->secondary_alternative  = 0x376;
        ide->busmaster_base         = 0x0;
        ide->sector_size            = 512;
        ide->total_num_sectors      = IdeGetNumSectors(ide);
        ide->transfer_buffer        = (uint16_t*) MapVirt(0, 0, MAX_TRANSFER_SIZE, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
        InitDiskPartitionHelper(&ide->partitions);

        node->data = ide;
        AddVfsMount(node, GenerateNewRawDiskName(DISKUTIL_TYPE_FIXED));
        CreateDiskPartitions(CreateOpenFile(node, 0, 0, true, true));
    }
}
```

File: ./arch/x86/dev/pit.c

```

#include <machine/pic.h>
#include <machine/pit.h>
#include <machine/portio.h>
#include <arch.h>
#include <assert.h>
#include <common.h>
#include <timer.h>
#include <irq.h>
#include <log.h>

/*
* x86/dev/pit.c - Programmable Interval Timer
*
* A system timer which can generate an interrupt (on IRQ 0) at regular intervals.
*/

static uint64_t pit_hertz = 0;
static uint64_t pit_nanos = 0;

static int HandlePit(struct x86_regs* r) {
    (void) r;
    assert(pit_nanos != 0);

    ReceivedTimer(pit_nanos);

    return 0;
}

void InitPit(int hertz) {
 pit_hertz = (uint64_t) hertz;

 int divisor = 1193180 / hertz;
 outb(0x43, 0x36);
 outb(0x40, divisor & 0xFF);
 outb(0x40, divisor >> 8);

    pit_nanos = 1000000000ULL / pit_hertz;

    RegisterIrqHandler(PIC_IRQ_BASE + 0, HandlePit);
}
```

File: ./arch/x86/lowlevel/trap.s

```

  global isr0
  global isr1
  global isr2
  global isr3
  global isr4
  global isr5
  global isr6
  global isr7
  global isr8
  global isr9
  global isr10
  global isr11
  global isr12
  global isr13
  global isr14
  global isr15
  global isr16
  global isr17
  global isr18
  global isr19
  global isr20
  global isr21
  global isr96
  global irq0
  global irq1
  global irq2
  global irq3
  global irq4
  global irq5
  global irq6
  global irq7
  global irq8
  global irq9
  global irq10
  global irq11
  global irq12
  global irq13
  global irq14
  global irq15


 ; We don't need to disable interrupts - they are automatically disabled when
 ; the interrupt comes in

 isr0:
     push 0
     push 0
     jmp int_common_handler

 isr1:
     push byte 0
     push byte 1
     jmp int_common_handler

 isr2:
     push byte 0
     push byte 2
     jmp int_common_handler

 isr3:
     push byte 0
     push byte 3
     jmp int_common_handler

 isr4:
     push byte 0
     push byte 4
     jmp int_common_handler

 isr5:
     push byte 0
     push byte 5
     jmp int_common_handler

 isr6:
     push byte 0
     push byte 6
     jmp int_common_handler

 isr7:
     push byte 0
     push byte 7
```

```
84      jmp int_common_handler
85
86  isr8:
87      push byte 8
88      jmp int_common_handler
89
90  isr9:
91      push byte 0
92      push byte 9
93      jmp int_common_handler
94
95  isr10:
96      push byte 10
97      jmp int_common_handler
98
99  isr11:
100     push byte 11
101     jmp int_common_handler
102
103 isr12:
104     push byte 12
105     jmp int_common_handler
106
107 isr13:
108     push byte 13
109     jmp int_common_handler
110
111 isr14:
112     push byte 14
113     jmp int_common_handler
114
115 isr15:
116     push byte 0
117     push byte 15
118     jmp int_common_handler
119
120 isr16:
121     push byte 0
122     push byte 16
123     jmp int_common_handler
124
125 isr17:
126     push byte 17
127     jmp int_common_handler
128
129 isr18:
130     push byte 0
131     push byte 18
132     jmp int_common_handler
133
134 isr19:
135     push byte 0
136     push byte 19
137     jmp int_common_handler
138
139 isr20:
140     push byte 0
141     push byte 20
142     jmp int_common_handler
143
144 isr21:
145     push byte 0
146     push byte 21
147     jmp int_common_handler
148
149
150 ; This is our system call handler
151 isr96:
152     push byte 0
153     push 96
154     jmp int_common_handler
155
156
157
158 ; Note that in the PIC setup, we remap our IRQs so they start at 32
159 ; That is, IRQ0 is actually ISR32, etc. up to IRQ15 which is ISR47
160 ; This is so they don't clash with the exceptions above, which are
161 ; not re-mappable.
162 irq0:
163     push byte 0
164     push byte 32
165     jmp int_common_handler
166
167 irq1:
168     push byte 0
169     push byte 33
170     jmp int_common_handler
171
172 irq2:
173     push byte 0
174     push byte 34
175     jmp int_common_handler
176
177 irq3:
178     push byte 0
179     push byte 35
180     jmp int_common_handler
181
182 irq4:
183     push byte 0
184     push byte 36
185     jmp int_common_handler
186
187 irq5:
188     push byte 0
189     push byte 37
190     jmp int_common_handler
191
192 irq6:
193     push byte 0
194     push byte 38
195     jmp int_common_handler
196
197 irq7:
198     push byte 0
199     push byte 39
200     jmp int_common_handler
201
202 irq8:
203     push byte 0
204     push byte 40
205     jmp int_common_handler
206
207 irq9:
208     push byte 0
209     push byte 41
210     jmp int_common_handler
211
212 irq10:
213     push byte 0
```

```
    push byte 42
    jmp int_common_handler

irq11:
    push byte 0
    push byte 43
    jmp int_common_handler

irq12:
    push byte 0
    push byte 44
    jmp int_common_handler

irq13:
    push byte 0
    push byte 45
    jmp int_common_handler

irq14:
    push byte 0
    push byte 46
    jmp int_common_handler

irq15:
    push byte 0
    push byte 47
    jmp int_common_handler


; Our common interrupt handler
extern x86HandleInterrupt
int_common_handler:
    ; Save the registers and segments
    pushad
 push ds
    push es
    push fs
    push gs

    ; Ensure we have kernel segments and not user segments
    mov ax, 0x10
 mov ds, ax
    mov es, ax
    ; We are going to use FS to store the CPU number, so don't overwrite it
    ; Kernel has no use for GS, so don't bother setting it

    ; Allow nested interrupts
    sti

    ; Push a pointer to the registers to the kernel handler
    push esp
 cld
    call x86HandleInterrupt

    ; Restore registers
    add esp, 4
    pop gs
    pop fs
    pop es
    pop ds
    popad

    ; Skip over the error code and interrupt number (we can't pop them
    ; anywhere, as the registers have already been restored)
    add esp, 8

    ; Return from the interrupt - also restores the stack and the flags
    iretd
```

File: ./arch/x86/lowlevel/idt.s

```

global x86LoadIdt
x86LoadIdt:
 ; The address of the IDTR is passed in as an argument
 mov eax, [esp + 4]
 lidt [eax]

 ret
```

File: ./arch/x86/lowlevel/gdt.c

```

#include <common.h>
#include <cpu.h>
#include <machine/gdt.h>

extern void x86LoadGdt(size_t addr);

static struct gdt_entry x86CreateGdtEntry(size_t base, size_t limit, uint8_t access, uint8_t granularity)
{
struct gdt_entry entry;

entry.base_low = base & 0xFFFF;
entry.base_middle = (base >> 16) & 0xFF;
entry.base_high = (base >> 24) & 0xFF;
entry.limit_low = limit & 0xFFFF;
entry.flags_and_limit_high = (limit >> 16) & 0xF;
entry.flags_and_limit_high |= (granularity & 0xF) << 4;
entry.access = access;

return entry;
}

void x86InitGdt(void)
{
platform_cpu_data_t* cpu_data = GetCpu()->platform_specific;

cpu_data->gdt[0] = x86CreateGdtEntry(0, 0, 0, 0);     // null segment
cpu_data->gdt[1] = x86CreateGdtEntry(0, 0xFFFFFFFF, 0x9A, 0xC);  // kernel code
cpu_data->gdt[2] = x86CreateGdtEntry(0, 0xFFFFFFFF, 0x92, 0xC);  // kernel data
cpu_data->gdt[3] = x86CreateGdtEntry(0, 0xFFFFFFFF, 0xFA, 0xC);  // user code
cpu_data->gdt[4] = x86CreateGdtEntry(0, 0xFFFFFFFF, 0xF2, 0xC);  // user data

cpu_data->gdtr.size = sizeof(cpu_data->gdt) - 1;
cpu_data->gdtr.location = (size_t) &cpu_data->gdt;

x86LoadGdt((size_t) &cpu_data->gdtr);
}

/*
* Adds a Task State Segment (TSS) entry into the GDT. This allows the TSS to be
* used to switch from user mode to kernel mode.
*
* Returns the selector used in the GDT.
*/

uint16_t x86AddTssToGdt(struct tss* tss)
{
platform_cpu_data_t* cpu_data = GetCpu()->platform_specific;
cpu_data->gdt[5] = x86CreateGdtEntry((size_t) tss, sizeof(struct tss), 0x89, 0x0);
return 5 * 0x8;
}
```

File: ./arch/x86/lowlevel/tss.c

```

#include <heap.h>
#include <machine/tss.h>
#include <arch.h>
#include <machine/gdt.h>
#include <cpu.h>

extern void x86LoadTss(size_t selector);

void x86InitTss(void) {
    platform_cpu_data_t* cpu_data = GetCpu()->platform_specific;
    cpu_data->tss = (struct tss*) AllocHeap(sizeof(struct tss));

    cpu_data->tss->link = 0x10;
    cpu_data->tss->esp0 = 0;
    cpu_data->tss->ss0 = 0x10;
    cpu_data->tss->iopb = sizeof(struct tss);

    uint16_t selector = x86AddTssToGdt(cpu_data->tss);
    x86LoadTss(selector);
}
```

File: ./arch/x86/lowlevel/misc.s

```

global ArchReadTimestamp
global ArchEnableInterrupts
global ArchDisableInterrupts
global ArchStallProcessor
global ArchFlushTlb
global x86GetCr2
global x86AreCpusOn
global ArchGetCurrentCpuIndex

ArchGetCurrentCpuIndex:
        ; Needs to be something that can't be modified by user code (e.g. a debug register).
mov eax, dr3
ret

ArchReadTimestamp:
rdtsc
ret

ArchEnableInterrupts:
sti
ret

ArchDisableInterrupts:
cli
ret

ArchStallProcessor:
hlt
ret

x86AreCpusOn:
    pushf
    pop eax
    and eax, 0x200
    shr eax, 9
    ret
```

File: ./arch/x86/lowlevel/idt.c

```
1
2  #include <common.h>
3  #include <cpu.h>
4
5  /*
6  * x86/lowlevel/idt.c - Interrupt Descriptor Table
7  *
8  * The interrupt decriptor table (IDT) is essentially a lookup table for where
9  * the CPU should jump to when an interrupt is receieved.
10 */
11
12 extern void x86LoadIdt(size_t addr);
13
14 /*
15 * Our trap handlers, defined in lowlevel/trap.s, which will be called
16 * when an interrupt occurs.
17 */
18 extern void isr0();
19 extern void isr1();
20 extern void isr2();
21 extern void isr3();
22 extern void isr4();
23 extern void isr5();
24 extern void isr6();
25 extern void isr7();
26 extern void isr8();
27 extern void isr9();
28 extern void isr10();
29 extern void isr11();
30 extern void isr12();
31 extern void isr13();
32 extern void isr14();
33 extern void isr15();
34 extern void isr16();
35 extern void isr17();
36 extern void isr18();
37 extern void isr19();
38 extern void isr20();
39 extern void isr21();
40 extern void isr96();
41 extern void irq0();
42 extern void irq1();
43 extern void irq2();
44 extern void irq3();
45 extern void irq4();
46 extern void irq5();
47 extern void irq6();
48 extern void irq7();
49 extern void irq8();
50 extern void irq9();
51 extern void irq10();
52 extern void irq11();
53 extern void irq12();
54 extern void irq13();
55 extern void irq14();
56 extern void irq15();
57
58 /*
59 * Fill in an entry in the IDT. There are a number of 'types' of interrupt, determining
60 * whether interrupts are disabled automatically before calling the handler, whether
61 * it is a 32-bit or 16-bit entry, and whether user mode can invoke the interrupt manually.
62 */
63 static void x86SetIdtEntry(int num, size_t isr_addr, uint8_t type)
64 {
65  platform_cpu_data_t* cpu_data = GetCpu()->platform_specific;
66
67  cpu_data->idt[num].isr_offset_low = (isr_addr & 0xFFFF);
68  cpu_data->idt[num].isr_offset_high = (isr_addr >> 16) & 0xFFFF;
69  cpu_data->idt[num].segment_selector = 0x08;
70  cpu_data->idt[num].reserved = 0;
71  cpu_data->idt[num].type = type;
72 }
73
74 /*
75 * Initialise the IDT. After this has occured, interrupts may be enabled.
76 */
77 void x86InitIdt(void)
78 {
79  platform_cpu_data_t* cpu_data = GetCpu()->platform_specific;
80
81  void (*const isrs[])() = {
82   isr0 , isr1 , isr2 , isr3 , isr4 , isr5 , isr6 , isr7 ,
83   isr8 , isr9 , isr10, isr11, isr12, isr13, isr14, isr15,
84   isr16, isr17, isr18, isr19, isr20, isr21,
85  };
86
87  void (*const irqs[])() = {
88   irq0 , irq1 , irq2 , irq3 , irq4 , irq5 , irq6 , irq7 ,
89   irq8 , irq9 , irq10, irq11, irq12, irq13, irq14, irq15
90  };
91
92  /*
93  * Install handlers for CPU exceptions (e.g. for page faults, divide-by-zero, etc.).
94  */
95  for (int i = 0; i < 21; ++i) {
96   x86SetIdtEntry(i, (size_t) isrs[i], 0x8E);
97  }
98
99  /*
100 * Install handlers for IRQs (hardware interrupts, e.g. keyboard, system timer).
101 */
102 for (int i = 0; i < 16; ++i) {
103  x86SetIdtEntry(i + 32, (size_t) irqs[i], 0x8E);
104 }
105
106 /*
107 * Install our system call handler. Note that the flag byte is 0xEE instead of 0x8E,
108 * this allows user code to directly invoke this interrupt.
109 */
110 x86SetIdtEntry(96, (size_t) isr96, 0xEE);
111
112 cpu_data->idtr.location = (size_t) &cpu_data->idt;
113 cpu_data->idtr.size = sizeof(cpu_data->idt) - 1;
114
115 x86LoadIdt((size_t) &cpu_data->idtr);
116 }
```

File: ./arch/x86/lowlevel/tss.s

```
1
2 ;
3 ;
4 ; x86/lowlevel/tss.s - Task State Segment
5 ;
6 ; Like with the GDT and IDT, we need assembly to load the TSS using the
7 ; special instruction 'ltr'.
8 ;
9
10 extern x86LoadTss
11
12 x86LoadTss:
13     mov eax, [esp + 4]
14     ltr ax
15     ret
```

File: ./arch/x86/lowlevel/gdt.s

```
1
2
3  global x86LoadGdt
4  x86LoadGdt:
5   ; The address of the GDTR is passed in as an argument
6   mov eax, [esp + 4]
7   lgdt [eax]
8
9   ; We now need to reload CS using a far jump...
10  jmp 0x08:.reloadSegments
11
12 .reloadSegments:
13  ; And all of the other segments by loading them
14  mov ax, 0x10
15  mov ds, ax
16  mov es, ax
17  ; Kernel doesn't use gs/fs
18  mov ss, ax
19
20  ret
```

File: ./arch/x86/application.ld

```
ENTRY(_start)
OUTPUT_FORMAT("elf32-i386")

SECTIONS
{
. = 0x10000000;

.text ALIGN(4096) : AT (ADDR (.text))
{
*(.text)
*(.rodata)
*(.symtab)
*(.strtab)
}

.data ALIGN(4096) : AT (ADDR (.data))
{
*(.data)
}

.bss ALIGN(4096) : AT (ADDR (.bss))
{
*(COMMON)
*(.bss)
*(.bootstrap_stack)
}

/DISCARD/ :
{
*(.comment)
}
}
```

File: ./arch/x86/thread/spinlock.s

```
;
;
; x86/thread/spinlock.s - Spinlocks
;
; Implement spinlocks in assembly so we can guarentee that they are
; atmoic.
;
;


global ArchSpinlockAcquire
global ArchSpinlockRelease

ArchSpinlockAcquire:
 ; The address of the lock is passed in as an argument
 mov eax, [esp + 4]

.try_acquire:
 ; Try to acquire the lock
 lock bts dword [eax], 0
 jc .spin_wait

 ret

.spin_wait:
 ; Lock was not acquired, so do the 'spin' part of spinlock

 ; Hint to the CPU that we are spinning
 pause

 ; No point trying to acquire it until it is free
 test dword [eax], 1
 jnz .spin_wait

 ; Now that it is free, we can attempt to atomically acquire it again
 jmp .try_acquire


ArchSpinlockRelease:
 ; The address of the lock is passed in as an argument
 mov eax, [esp + 4]
 lock btr dword [eax], 0
 ret
```

File: ./arch/x86/thread/usermode.s

```
 global ArchSwitchToUsermode

 ; This is only called the first time we want to switch a given
 ; thread into usermode. In all other cases the switch will occur
 ; back through an interrupt handler (e.g. after a system call completes)

ArchSwitchToUsermode:
    ; Takes in an address to a usermode address to start execution
    mov ebx, [esp + 4]

    ; And a user stack pointer
    mov ecx, [esp + 8]

    ; And initial argument. For x86, we'll just store this in EDX, and the
    ; program can just read that value.
    mov edx, [esp + 12]

    ; Usermode data segment
    mov ax, 0x23
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; We need to push the current stack as the usermode part will use it too
    push 0x23             ; Usermode stack segment
    push ecx              ; Usermode stack pointer
    push 0x202            ; Flags
    push 0x1B             ; Usermode code segment
    push ebx              ; Usermode entry point

    ; Pop all of that off the stack and go to usermode.
    iretd
```

File: ./arch/x86/thread/switch.s

```
1  global ArchPrepareStack
2  global ArchSwitchThread
3
4  extern ThreadInitialisationHandler
5  extern GetCpu
6
7  ArchPrepareStack:
8    ; We need to put 5 things on the stack - dummy values for EBX, ESI,
9    ; EDI and EBP, as well as the address of thread_startup_handler
10   ;
11   ; This is because these get popped off in arch_switch_thread
12
13   ; Grab the address of the new thread's stack from our stack (it was
14   ; passed in as an argument)
15   mov eax, [esp + 4]
16
17   ; We need to get to the bottom position, and we also need to return that
18   ; address in EAX so it can be put into the struct, so it makes sense to modify it.
19   sub eax, 20
20
21   ; This is where the address of arch_switch_thread needs to go.
22   ; +0 is where EBP is, +4 is EDI, +8 for ESI, +12 for EBX,
23   ; and so +16 for the return value.
24   ; (see the start of arch_switch_thread for where these get pushed)
25   mov [eax + 16], dword ThreadInitialisationHandler
26
27   ret
28
29 ArchSwitchThread:
30   ; The old and new threads are passed in on the stack as arguments, in that order.
31
32   ; The calling convention we use already saves EAX, ECX and EDX whenever a
33   ; function (e.g. ArchSwitchThread) is called. Therefore, we only need to save the other four.
34   push ebx
35   push esi
36   push edi
37   push ebp
38
39   ; We are now free to trash the general purpose registers (except ESP),
40   ; so we can now load the current task using the argument.
41
42      ; First we have to save the old stack pointer. The old thread was the first
43      ; argument, and we just pushed 4 things to the stack. The first argument gets
44      ; pushed last, so read back 5 places. Also load the new thread's address in.
45
46      mov edi, [esp + (4 + 1) * 4]        ; edi = old_thread
47      mov esi, [esp + (4 + 2) * 4]        ; esi = new_thread
48
49   ; The second entry in a thread structure is guaranteed to be the stack pointer.
50   ; Save our stack there.
51   mov [edi + 4], esp                  ; old_thread->stack_pointer = esp
52
53      ; Now we can load the new thread's stack pointer.
54      mov esp, [esi + 4]                  ; esp = new_thread->stack_pointer
55
56      ; ESI is callee-saved, so no need to do anything here. We only need ESI and ESP
57      ; at this point, so it's all good.
58
59      call GetCpu                         ; eax = GetCpu()
60
61   ; The top of the kernel stack (which needs to go in the TSS for
62   ; user to kernel switches), is the first entry in new_thread.
63   mov ebx, [esi]                      ; ebx = new_thread->kernel_stack_top
64
65   ; The third entry in current_cpu is a pointer to CPU specific data.
66   mov ecx, [eax + 8]                  ; ecx = GetCpu()->platform_specific
67
68   ; The first entry in the CPU specific data is the TSS pointer
69   mov edx, [ecx + 0]                  ; edx = GetCpu()->platform_specific->tss
70
71   ; Load the TSS's ESP0 with the new thread's stack
72   mov [edx + 4], ebx
73
74   ; Now we have the new thread's stack, we can just pop off the state
75   ; that would have been pushed when it was switched out.
76   pop ebp
77   pop edi
78   pop esi
79   pop ebx
80
81   ret
```

File: ./arch/x86/progload.ld

```
ENTRY(_start)
OUTPUT_FORMAT("binary")

SECTIONS
{
. = 0xBFC00000;

.text ALIGN(4096) : AT (ADDR (.text))
{
*(.text)
*(.rodata)
*(.symtab)
*(.strtab)
}

.data ALIGN(4096) : AT (ADDR (.data))
{
*(.data)
}

.bss ALIGN(4096) : AT (ADDR (.bss))
{
*(COMMON)
*(.bss)
*(.bootstrap_stack)
}

.fake : { . = . + SIZEOF(.bss); }

/DISCARD/ :
{
*(.comment)
}
}
```

File: ./arch/x86/linker.ld

```
ENTRY(_start)
OUTPUT_FORMAT("elf32-i386")

SECTIONS
{
. = 1M;

_kernel_start = .;
.multiboot.data : {
*(.multiboot.data)
}

.multiboot.text : {
*(.multiboot.text)
}

. += 0xC0000000;

.text ALIGN(4096) : AT (ADDR (.text) - 0xC0000000)
{
*(.text .text.*)
*(.ctors)
*(.dtors)
}

.rodata ALIGN(4096) : AT (ADDR (.rodata) - 0xC0000000)
{
*(.rodata)
}


. = ALIGN(4096);
_start_pageablek_section = .;

.pageablek ALIGN(4096) : AT (ADDR (.data) + SIZEOF(.data) - 0xC0000000)
{
*(.pageablektext)
*(.pageablekdata)
}
_end_pageablek_section = .;

.data ALIGN(4096) : AT (ADDR (.data) - 0xC0000000)
{
*(.data)
}

.bss ALIGN(4096) : AT (ADDR (.bss) - 0xC0000000)
{
*(COMMON)
*(.bss)
*(.bootstrap_stack)
}

_kernel_end = .;

/DISCARD/ :
{
*(.comment)
}
}
```

File: ./arch/x86/elf/elf.c

```
#include <common.h>
#include <errno.h>
#include <log.h>
#include <vfs.h>
#include <virtual.h>
#include <voidptr.h>
#include <string.h>
#include <driver.h>
#include <sys/stat.h>
#include <assert.h>
#include <heap.h>
#include <irql.h>
#include <stdlib.h>
#include <machine/elf.h>
#include <panic.h>

static bool IsElfValid(struct Elf32_Ehdr* header) {
    if (header->e_ident[EI_MAG0] != ELFMAG0) return false;
    if (header->e_ident[EI_MAG1] != ELFMAG1) return false;
    if (header->e_ident[EI_MAG2] != ELFMAG2) return false;
    if (header->e_ident[EI_MAG3] != ELFMAG3) return false;

    /* TODO: check for other things, such as the platform, etc. */

    return true;
}

static size_t ElfGetSizeOfImageIncludingBss(void* data) {
    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;
 struct Elf32_Phdr* prog_headers = (struct Elf32_Phdr*) AddVoidPtr(data, elf_header->e_phoff);

    size_t base_point = 0xD0000000U;
    size_t total_size = 0;

    for (int i = 0; i < elf_header->e_phnum; ++i) {
        struct Elf32_Phdr* prog_header = prog_headers + i;

        size_t address = prog_header->p_vaddr;
  size_t size = prog_header->p_filesz;
  size_t num_zero_bytes = prog_header->p_memsz - size;
  size_t type = prog_header->p_type;

  if (type == PHT_LOAD) {
            if (address - base_point + size + num_zero_bytes >= total_size) {
                total_size = address - base_point + size + num_zero_bytes;
            }
  }
    }

    return (total_size + ARCH_PAGE_SIZE - 1) & (~(ARCH_PAGE_SIZE - 1));
```

```
}

static int ElfLoadProgramHeaders(void* data, size_t relocation_point, struct open_file* file) {
    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;
 struct Elf32_Phdr* prog_headers = (struct Elf32_Phdr*) AddVoidPtr(data, elf_header->e_phoff);

    size_t base_point = 0xD0000000U;

    for (int i = 0; i < elf_header->e_phnum; ++i) {
        struct Elf32_Phdr* prog_header = prog_headers + i;

        size_t address = prog_header->p_vaddr;
  size_t offset = prog_header->p_offset;
  size_t size = prog_header->p_filesz;
  size_t type = prog_header->p_type;
  uint32_t flags = prog_header->p_flags;
  size_t num_zero_bytes = prog_header->p_memsz - size;

  if (type == PHT_LOAD) {
   size_t addr = address + relocation_point - base_point;
   size_t remainder = size & (ARCH_PAGE_SIZE - 1);

   int page_flags = 0;
   if (flags & PF_X) page_flags |= VM_EXEC;
   if (flags & PF_W) page_flags |= VM_WRITE;
   if (flags & PF_R) page_flags |= VM_READ;

   /*
    * We don't actually want to write to the executable file, so we must just copy to the page as normal
    * instead of using a file-backed page.
    */
   if (flags & PF_W) {
    size_t pages = (size + num_zero_bytes + (ARCH_PAGE_SIZE - 1)) / ARCH_PAGE_SIZE;

    for (size_t i = 0; i < pages; ++i) {
     SetVirtPermissions(addr + i * ARCH_PAGE_SIZE, page_flags, (VM_READ | VM_WRITE | VM_EXEC) & ~page_flags);
    }

    memcpy((void*) addr, (const void*) AddVoidPtr(data, offset), size);

   } else {
    size_t pages = (size - remainder) / ARCH_PAGE_SIZE;

    if (addr & (ARCH_PAGE_SIZE - 1)) {
     return EINVAL;
    }
    if (pages > 0) {
     LogWriteSerial("doing the little fiddly thing...\n");
     UnmapVirt(addr, pages * ARCH_PAGE_SIZE);
     size_t v = MapVirt(relocation_point, addr, pages * ARCH_PAGE_SIZE, VM_RELOCATABLE | VM_FILE | page_flags, file, offset);
     if (v != addr) {
      return ENOMEM;
     }
    }

    if (remainder > 0) {
     SetVirtPermissions(addr + pages * ARCH_PAGE_SIZE, page_flags | VM_WRITE, (VM_READ | VM_EXEC) & ~page_flags);
     memcpy((void*) AddVoidPtr(addr, pages * ARCH_PAGE_SIZE), (const void*) AddVoidPtr(data, offset + pages * ARCH_PAGE_SIZE), remainder);
     SetVirtPermissions(addr + pages * ARCH_PAGE_SIZE, 0, VM_WRITE);
    }
   }
  }
    }

 return 0;
}

static char* ElfLookupString(void* data, int offset) {
    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;

 if (elf_header->e_shstrndx == SHN_UNDEF) {
  return NULL;
 }

 struct Elf32_Shdr* sect_headers = (struct Elf32_Shdr*) AddVoidPtr(data, elf_header->e_shoff);

 char* string_table = (char*) AddVoidPtr(data, sect_headers[elf_header->e_shstrndx].sh_offset);
 if (string_table == NULL) {
  return NULL;
 }

 return string_table + offset;
}

static size_t ElfGetSymbolValue(void* data, int table, size_t index, bool* error, size_t relocation_point, size_t base_address) {
    *error = false;

 if (table == SHN_UNDEF || index == SHN_UNDEF) {
  *error = true;
  return 0;
 }

    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;
    struct Elf32_Shdr* sect_headers = (struct Elf32_Shdr*) AddVoidPtr(data, elf_header->e_shoff);
 struct Elf32_Shdr* symbol_table = sect_headers + table;
 struct Elf32_Shdr* string_table = sect_headers + symbol_table->sh_link;

 size_t num_symbol_table_entries = symbol_table->sh_size / symbol_table->sh_entsize;
 if (index >= num_symbol_table_entries) {
  *error = true;
  return 0;
 }

 struct Elf32_Sym* symbol = ((struct Elf32_Sym*) AddVoidPtr(data, symbol_table->sh_offset)) + index;

 if (symbol->st_shndx == SHN_UNDEF) {
  const char* name = (const char*) AddVoidPtr(data, string_table->sh_offset + symbol->st_name);

  size_t target = GetSymbolAddress(name);
  if (target == 0) {
   if (!(ELF32_ST_BIND(symbol->st_info) & STB_WEAK)) {
    *error = true;
   }
   return 0;

  } else {
   return target;
  }

 } else if (symbol->st_shndx == SHN_ABS) {
  return symbol->st_value;

 } else {
  return symbol->st_value + (relocation_point - base_address);
 }
}

static bool ElfPerformRelocation(void* data, size_t relocation_point, struct Elf32_Shdr* section, struct Elf32_Rel* relocation_table, struct quick_relocation_tab
{
 size_t base_address = 0xD0000000U;
```

```
size_t addr = (size_t) relocation_point - base_address + relocation_table->r_offset;
size_t* ref = (size_t*) addr;

int symbolValue = 0;
if (ELF32_R_SYM(relocation_table->r_info) != SHN_UNDEF) {
 bool error = false;
 symbolValue = ElfGetSymbolValue(data, section->sh_link, ELF32_R_SYM(relocation_table->r_info), &error, relocation_point, base_address);
 if (error) {
  return false;
 }
}

bool needs_write_low = (GetVirtPermissions(addr) & VM_WRITE) == 0;
bool needs_write_high = (GetVirtPermissions(addr + sizeof(size_t) - 1) & VM_WRITE) == 0;

if (needs_write_low) {
 SetVirtPermissions(addr, VM_WRITE, 0);
}
if (needs_write_high) {
 SetVirtPermissions(addr + sizeof(size_t) - 1, VM_WRITE, 0);
}

int type = ELF32_R_TYPE(relocation_table->r_info);
bool success = true;
size_t val = 0;

if (type == R_386_32) {
 val = DO_386_32(symbolValue, *ref);

} else if (type == R_386_PC32) {
 val = DO_386_PC32(symbolValue, *ref, addr);

} else if (type == R_386_RELATIVE) {
 val = DO_386_RELATIVE((relocation_point - base_address), *ref);

} else {
 LogWriteSerial("some whacko type...\n");
 success = false;
}

*ref = val;
LogWriteSerial("relocating 0x%X -> 0x%X\n", addr, val);
AddToQuickRelocationTable(table, addr, val);

if (needs_write_low) {
 SetVirtPermissions(addr, 0, VM_WRITE);
}
if (needs_write_high) {
 SetVirtPermissions(addr + sizeof(size_t) - 1, 0, VM_WRITE);
}
return success;
}

static bool ElfPerformRelocations(void* data, size_t relocation_point, struct quick_relocation_table** table) {
    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;
struct Elf32_Shdr* sect_headers = (struct Elf32_Shdr*) AddVoidPtr(data, elf_header->e_shoff);

for (int i = 0; i < elf_header->e_shnum; ++i) {
 struct Elf32_Shdr* section = sect_headers + i;

 if (section->sh_type == SHT_REL) {
  struct Elf32_Rel* relocation_tables = (struct Elf32_Rel*) AddVoidPtr(data, section->sh_offset);
  int count = section->sh_size / section->sh_entsize;

  if (strcmp(ElfLookupString(data, section->sh_name), ".rel.dyn")) {
   continue;
  }

  *table = CreateQuickRelocationTable(count);

  for (int index = 0; index < count; ++index) {
   bool success = ElfPerformRelocation(data, relocation_point, section, relocation_tables + index, *table);
   if (!success) {
    LogWriteSerial("failed to do a relocation!! (%d)\n", index);
    return false;
   }
  }

  SortQuickRelocationTable(*table);


  } else if (section->sh_type == SHT_RELA) {
   LogDeveloperWarning("[ElfPerformRelocations]: unsupported section type: SHT_RELA\n");
            return false;
  }
 }

 return true;
}

static int ElfLoad(void* data, size_t* relocation_point, struct open_file* file, struct quick_relocation_table** table) {
    MAX_IRQL(IRQL_PAGE_FAULT);

    struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) data;

    if (!IsElfValid(elf_header)) {
        return EINVAL;
    }

    /*
    * To load a driver, we need the section headers.
    */
    if (elf_header->e_shnum == 0) {
        return EINVAL;
    }

    /*
    * We always need the program headers.
    */
    if (elf_header->e_phnum == 0) {
        return EINVAL;
    }

    /*
    * Load into memory.
    */
    size_t size = ElfGetSizeOfImageIncludingBss(data);

*relocation_point = MapVirt(0, 0, size, VM_READ, NULL, 0);
LogWriteSerial("RELOCATION POINT AT 0x%X\n", *relocation_point);
ElfLoadProgramHeaders(data, *relocation_point, file);

bool success = ElfPerformRelocations(data, *relocation_point, table);
if (success) {
 return 0;

} else {
 return EINVAL;
}
}
```

```
311
312 int ArchLoadDriver(size_t* relocation_point, struct open_file* file, struct quick_relocation_table** table) {
313     MAX_IRQL(IRQL_PAGE_FAULT);
314
315     off_t file_size;
316  int res = GetFileSize(file, &file_size);
317  if (res != 0) {
318   return res;
319  }
320
321     size_t file_rgn = MapVirt(0, 0, file_size, VM_READ | VM_FILE, file, 0);
322     res = ElfLoad((void*) file_rgn, relocation_point, file, table);
323
324  struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) file_rgn;
325     struct Elf32_Shdr* sect_headers = (struct Elf32_Shdr*) (file_rgn + elf_header->e_shoff);
326
327  for (int i = 0; i < elf_header->e_shnum; ++i) {
328   const char* sh_name = ElfLookupString((void*) file_rgn, sect_headers[i].sh_name);
329   if (!strcmp(sh_name, ".lockedtext") || !strcmp(sh_name, ".lockeddata")) {
330    // it's okay to lock extra memory - it wouldn't be ok if we did it the other way around though
331    // (assumed everything was locked, and only unlocked parts of it)
332    size_t start_addr = (sect_headers[i].sh_addr - 0xD0000000U + *relocation_point) & ~(ARCH_PAGE_SIZE - 1);
333    size_t num_pages = (sect_headers[i].sh_size + ARCH_PAGE_SIZE - 1) / ARCH_PAGE_SIZE;
334    while (num_pages--) {
335     LockVirt(start_addr);
336     start_addr += ARCH_PAGE_SIZE;
337    }
338   }
339  }
340
341     UnmapVirt(file_rgn, file_size);
342
343     return res;
344 }
345
346 void ArchLoadSymbols(struct open_file* file, size_t adjust) {
347  off_t size;
348  int res = GetFileSize(file, &size);
349  if (res != 0) {
350   Panic(PANIC_BAD_KERNEL);
351  }
352
353  size_t mem = MapVirt(0, 0, size, VM_READ | VM_FILE, file, 0);
354
355     struct Elf32_Ehdr* elf_header = (struct Elf32_Ehdr*) mem;
356
357     /*
358     * These should never happen - otherwise our kernel shouldn't be running!
359     */
360     if (!IsElfValid(elf_header) || elf_header->e_shoff == 0) {
361   Panic(PANIC_BAD_KERNEL);
362     }
363
364     struct Elf32_Shdr* section_headers = (struct Elf32_Shdr*) (size_t) (mem + elf_header->e_shoff);
365     size_t symbol_table_offset = 0;
366     size_t symbol_table_length = 0;
367     size_t string_table_offset = 0;
368     size_t string_table_length = 0;
369
370     /*
371     * Find the address and size of the symbol and string tables.
372     */
373     for (int i = 0; i < elf_header->e_shnum; ++i) {
374         size_t file_offset = (section_headers + i)->sh_offset;
375         size_t address = (section_headers + elf_header->e_shstrndx)->sh_offset + (section_headers + i)->sh_name;
376
377         char* name_buffer = (char*) (mem + address);
378
379         if (!strcmp(name_buffer, ".symtab")) {
380             symbol_table_offset = file_offset;
381             symbol_table_length = (section_headers + i)->sh_size;
382
383         } else if (!strcmp(name_buffer, ".strtab")) {
384             string_table_offset = file_offset;
385             string_table_length = (section_headers + i)->sh_size;
386         }
387     }
388
389     if (symbol_table_offset == 0 || string_table_offset == 0 || symbol_table_length == 0 || string_table_length == 0) {
390         Panic(PANIC_BAD_KERNEL);
391     }
392
393     struct Elf32_Sym* symbol_table = (struct Elf32_Sym*) (mem + symbol_table_offset);
394     const char* string_table = (const char*) (mem + string_table_offset);
395
396     /*
397     * Register all of the visible symbols we find.
398     */
399     for (size_t i = 0; i < symbol_table_length / sizeof(struct Elf32_Sym); ++i) {
400         struct Elf32_Sym symbol = symbol_table[i];
401
402         if (symbol.st_value == 0) {
403             continue;
404         }
405
406    /*
407     * Skip "hidden" and "internal" symbols
408     */
409    if ((symbol.st_other & 3) != 0) {
410     continue;
411    }
412
413    /*
414     * No need for strdup, as the symbol table will call strdup anyway.
415     */
416         AddSymbol(string_table + symbol.st_name, symbol.st_value + adjust);
417     }
418
419  UnmapVirt(mem, size);
420 }
```

File: ./arch/x86/mem/virtual.s

```
1  global x86GetCr2
2  global x86SetCr3
3
4  x86GetCr2:
5      mov eax, cr2
6      ret
7
8  x86SetCr3:
9      mov eax, [esp + 4]
10     mov cr3, eax
11     ret
```

File: ./arch/x86/mem/physical.c

```

#include <common.h>
#include <arch.h>
#include <assert.h>
#include <panic.h>
#include <log.h>
#include <machine/virtual.h>

/*
* x86/mem/physical.c - Physical Memory Detection
*
* We need to detect what physical memory exists on the system so it can
* actually be allocated. When the bootloader runs, it puts a pointer to
* a table in EBX. This table then contains a pointer to a memory table
* containing the ranges of memory, and whether or not they are available for use.
*/

/*
* An entry in the memory table that GRUB loads.
*/
struct memory_table_entry
{
 uint32_t size;
 uint32_t addr_low;
 uint32_t addr_high;
 uint32_t len_low;
 uint32_t len_high;
 uint32_t type;

} __attribute__((packed));

/*
* We can only use memory if the type (see the struct above) is this.
*/
#define MULTIBOOT_MEMORY_AVAILABLE  1

/*
* A pointer to the main GRUB table. Defined and set correctly in
* x86/lowlevel/kernel_entry.s
*/
extern uint32_t* x86_grub_table;

/*
* A pointer to the memory table found in the main GRUB table.
*/
static struct memory_table_entry* memory_table = NULL;

struct arch_memory_range* ArchGetMemory(void)
{
 static struct arch_memory_range range;
 static int bytes_used = 0;
 static int table_length = 0;

retry:
 /*
 * If this is the first time we are called, we need to find the address of
 * the memory table in the main table.
 */
 if (memory_table == NULL) {
  x86_grub_table = (uint32_t*) x86KernelMemoryToPhysical((size_t) x86_grub_table);

  /*
  * A quick check to ensure that the table is somewhat valid.
  */
  uint32_t flags = x86_grub_table[0];
  if (!((flags >> 6) & 1)) {
   Panic(PANIC_NO_MEMORY_MAP);
  }

  table_length = x86_grub_table[11];
  memory_table = (struct memory_table_entry*) x86KernelMemoryToPhysical(x86_grub_table[12]);
 }

 /*
 * No more memory, we have reached the end of the table
 */
 if (bytes_used >= table_length) {
  return NULL;
 }

 /*
 * Start reading the memory table into the range.
 *
 * If the high half of the length is non-zero, we have at least 4GB of memory in
 * this range. We can't handle any more than 4GB, so just make it a 4GB range.
 */
 size_t type = memory_table->type;
 LogWriteSerial("At 0x%X, we have type %d\n", memory_table->addr_low, type);
 range.start = memory_table->addr_low;
 range.length = memory_table->len_high ? 0xFFFFFFFFU : memory_table->len_low;
 ++memory_table;
 bytes_used += sizeof(struct memory_table_entry);

 extern size_t _kernel_end;
 size_t max_kernel_addr = (((size_t) &_kernel_end) - 0xC0000000 + 0xFFF) & ~0xFFF;

 /*
 * Don't allow the use of non-RAM, or addresses completely below the kernel.
 */
 if (type != MULTIBOOT_MEMORY_AVAILABLE) {
  goto retry;
 }

 if (range.start < 0x80000) {
  if (range.start == 0x0) {
   range.start += 4096;
   range.length -= 4096;
  }

  /*
   * Try to salvage some low memory too.
   */
  if (range.length + range.start >= 0x80000) {
   range.length = 0x80000 - range.start;
  }
  if (range.length + range.start >= max_kernel_addr) {
   LogDeveloperWarning("LOST SOME MEMORY WITH RANGE 0x%X -> 0x%X\n", range.start, range.start + range.length);
  }

 } else if (range.start < 0x100000) {
  goto retry;

 } else if (range.start < max_kernel_addr) {
  /*
  * If it starts below the kernel, but ends above it, cut it off so only the
  * part above the kernel is used.
  */
  range.length = range.start + range.length - max_kernel_addr;
```

```
 range.start = max_kernel_addr;
 }
 if (range.length <= 0) {
  goto retry;
 }

 LogWriteSerial("Allowing range: 0x%X -> 0x%X to be used\n", range.start, range.start + range.length - 1);
 return &range;
}
```

File: ./arch/x86/mem/virtual.c

```
#include <machine/virtual.h>
#include <assert.h>
#include <string.h>
#include <arch.h>
#include <physical.h>
#include <arch.h>
#include <log.h>
#include <avl.h>
#include <heap.h>
#include <virtual.h>

__attribute__((fastcall)) size_t x86KernelMemoryToPhysical(size_t virtual)
{
 assert(virtual < 0x400000);
 return 0xC0000000 + virtual;
}

static struct vas vas_table[ARCH_MAX_CPU_ALLOWED];
static platform_vas_data_t vas_data_table[ARCH_MAX_CPU_ALLOWED];

static size_t kernel_page_directory[1024] __attribute__((aligned(ARCH_PAGE_SIZE)));
static size_t first_page_table[1024] __attribute__((aligned(ARCH_PAGE_SIZE)));

#define x86_PAGE_PRESENT  1
#define x86_PAGE_WRITE   2
#define x86_PAGE_USER   4
#define x86_PAGE_ACCESSED  (1 << 5)
#define x86_PAGE_DIRTY   (1 << 6)

static void x86AllocatePageTable(struct vas* vas, size_t table_num) {
 size_t* page_dir = vas->arch_data->v_page_directory;
 size_t page_dir_phys = AllocPhys();
 page_dir[table_num] = page_dir_phys | x86_PAGE_PRESENT | x86_PAGE_WRITE | x86_PAGE_USER;
 ArchFlushTlb(vas);
 inline_memset((void*) (0xFFC00000 + table_num * ARCH_PAGE_SIZE), 0, ARCH_PAGE_SIZE);
}

static size_t* x86GetPageEntry(struct vas* vas, size_t virtual) {
 if (vas != GetVas()) {
  LogDeveloperWarning("NON-LOCAL VAS x86GetPageEntry!!! THIS ISN'T GOING TO WORK AS-IS!\n");
 }
 size_t table_num = virtual / 0x400000;
 size_t page_num = (virtual % 0x400000) / ARCH_PAGE_SIZE;
 size_t* page_dir = vas->arch_data->v_page_directory;

 if (!(page_dir[table_num] & x86_PAGE_PRESENT)) {
  x86AllocatePageTable(vas, table_num);
 }

 return ((size_t*) (0xFFC00000 + table_num * ARCH_PAGE_SIZE)) + page_num;
}

static void x86MapPage(struct vas* vas, size_t physical, size_t virtual, int flags) {
 if (vas != GetVas()) {
  LogDeveloperWarning("NON-LOCAL VAS x86MapPage!!! THIS ISN'T GOING TO WORK AS-IS!\n");
 }
 LogWriteSerial("x86MapPage v 0x%X -> p 0x%X [0x%X]. vas 0x%X\n", virtual, physical, flags, vas);
 *x86GetPageEntry(vas, virtual) = physical | flags;
}

size_t ArchVirtualToPhysical(size_t virtual) {
 struct vas* vas = GetVas();

 size_t table_num = virtual / 0x400000;
 size_t page_num = (virtual % 0x400000) / ARCH_PAGE_SIZE;
 size_t* page_dir = vas->arch_data->v_page_directory;

 if (!(page_dir[table_num] & x86_PAGE_PRESENT)) {
  return 0;
 }

 size_t entry = ((size_t*) (0xFFC00000 + table_num * ARCH_PAGE_SIZE))[page_num];
 if (entry & x86_PAGE_PRESENT) {
  return entry & (~0xFFF);
 } else {
  return 0;
 }
}

void ArchUpdateMapping(struct vas* vas, struct vas_entry* entry) {
 int flags = 0;

 /*
  * Non-in-RAM pages need to be writable, as we need to be able to bring them into RAM
  * by writing to them!
  */
 if ((!entry->cow && entry->write) || entry->allow_temp_write) flags |= x86_PAGE_WRITE;
 if (entry->in_ram) flags |= x86_PAGE_PRESENT;
 if (entry->user) flags |= x86_PAGE_USER;

 if (entry->num_pages > 1) {
  for (int i = 0; i < entry->num_pages; ++i) {
   x86MapPage(vas, entry->physical == 0 ? 0 : (entry->physical + i * ARCH_PAGE_SIZE), entry->virtual + i * ARCH_PAGE_SIZE, flags);
  }
 } else {
  x86MapPage(vas, entry->physical, entry->virtual, flags);
 }
}

void ArchGetPageUsageBits(struct vas* vas, struct vas_entry* vas_entry, bool* accessed, bool* dirty) {
 size_t entry = *x86GetPageEntry(vas, vas_entry->virtual);
 *accessed = entry & x86_PAGE_ACCESSED;
 *dirty = entry & x86_PAGE_DIRTY;
}

void ArchSetPageUsageBits(struct vas* vas, struct vas_entry* vas_entry, bool accessed, bool dirty) {
 size_t* entry = x86GetPageEntry(vas, vas_entry->virtual);

 if (accessed) *entry |= x86_PAGE_ACCESSED;
 else *entry &= ~x86_PAGE_ACCESSED;

 if (dirty) *entry |= x86_PAGE_DIRTY;
 else *entry &= ~x86_PAGE_DIRTY;
```

```
}

void ArchAddMapping(struct vas* vas, struct vas_entry* entry) {
 ArchUpdateMapping(vas, entry);
}

void ArchUnmap(struct vas* vas, struct vas_entry* entry) {
 x86MapPage(vas, 0, entry->virtual, 0);
}

void ArchSetVas(struct vas* vas) {
 extern size_t x86SetCr3(size_t);
 x86SetCr3(vas->arch_data->p_page_directory);
}

void ArchFlushTlb(struct vas* vas) {
 ArchSetVas(vas);
}

void ArchInitVas(struct vas* vas) {
 vas->arch_data = AllocHeap(sizeof(platform_vas_data_t));
 vas->arch_data->v_page_directory = (size_t*) MapVirt(0, 0, ARCH_PAGE_SIZE, VM_READ | VM_WRITE | VM_USER | VM_LOCK, NULL, 0);
 vas->arch_data->p_page_directory = GetPhysFromVirt((size_t) vas->arch_data->v_page_directory);

 for (int i = 768; i < 1023; ++i) {
  vas->arch_data->v_page_directory[i] = kernel_page_directory[i];
 }
 vas->arch_data->v_page_directory[1023] = ((size_t) vas->arch_data->p_page_directory) | x86_PAGE_PRESENT | x86_PAGE_WRITE;
}

void ArchInitVirt(void) {
 struct vas* vas = &vas_table[0];
 vas->arch_data = &vas_data_table[0];
 CreateVasEx(vas, VAS_NO_ARCH_INIT);

 inline_memset(kernel_page_directory, 0, ARCH_PAGE_SIZE);
 inline_memset(first_page_table, 0, ARCH_PAGE_SIZE);

 extern size_t _kernel_end;
 size_t max_kernel_addr = (((size_t) &_kernel_end) + 0xFFF) & ~0xFFF;

 /*
 * Map the kernel by mapping the first 1MB + kernel size up to 0xC0000000 (assumes the kernel is
    * less than 4MB). This needs to match what kernel_entry.s exactly.
 */

 kernel_page_directory[768] = ((size_t) first_page_table - 0xC0000000) | x86_PAGE_PRESENT | x86_PAGE_WRITE | x86_PAGE_USER;

 /* <= is required to make it match kernel_entry.s */
 size_t num_pages = (max_kernel_addr - 0xC0000000) / ARCH_PAGE_SIZE;
    for (size_t i = 0; i <= num_pages; ++i) {
  first_page_table[i] = (i * ARCH_PAGE_SIZE) | x86_PAGE_PRESENT | x86_PAGE_WRITE;
 }

 /*
 * Set up recursive mapping by mapping the 1024th page table to
 * the page directory. See arch_vas_set_entry for an explaination of why we do this.
    * "Locking" this page directory entry is the only we can lock the final page of virtual
    * memory, due to the recursive nature of this entry.
 */
 kernel_page_directory[1023] = ((size_t) kernel_page_directory - 0xC0000000) | x86_PAGE_PRESENT | x86_PAGE_WRITE;

 vas->arch_data->p_page_directory = ((size_t) kernel_page_directory) - 0xC0000000;
 vas->arch_data->v_page_directory = kernel_page_directory;

 SetVas(vas);

 /*
 * The virtual memory manager is now initialised, so we can fill in
 * the rest of the kernel state. This is important as we need all of
 * the kernel address spaces to share page tables, so we must allocate
 * them all now so new address spaces can copy from us.
 */
 /*for (int i = 769; i < 1023; ++i) {
  x86AllocatePageTable(vas, i);
 }*/

 /*
  * The maximum amount of virtual kernel memory we can access will depend on
  * the amount of RAM - full access requires 1MB, which is an issue if we've got, e.g.
  * only 1.5MB of RAM. We ensure we always get at least 128MB of kernel virtual memory -
  * systems with 1.5MB of RAM will certainly not need that much virtual memory.
  *
  * A quick reference table:
  *
  *  Physical RAM  Max Kernel Virtual RAM  Physical RAM Usage
  *  1280 KB    132 MB       248 /  544 (54% free)
  *  1536 KB    194 MB       312 /  800 (61% free)
  *  2048 KB    324 MB       440 / 1312 (66% free)
  *  3072 KB    580 MB       696 / 2336 (70% free)
  *  4096 KB    764 MB       880 / 3360 (73% free)
  *  8192 KB    764 MB       884 / 7456 (88% free)
  */

 size_t tables_allocated = 0;
 int start = ARCH_KRNL_SBRK_BASE / 0x400000;
 for (int i = start; i < 1023; ++i) {
  if (i >= start + 32 && tables_allocated * 16 > GetTotalPhysKilobytes()) {
   break;
  }
  ++tables_allocated;
  x86AllocatePageTable(vas, i);
 }

 LogWriteSerial("can access %d MB of kernel virtual memory\n", tables_allocated * 4);

 /*
  * The boot assembly code set up two page tables for us, that we no longer need.
  * We can release that physical memory.
  */
 extern size_t boot_page_directory;
 extern size_t boot_page_table1;
 DeallocPhys(ArchVirtualToPhysical((size_t) &boot_page_directory));
 DeallocPhys(ArchVirtualToPhysical((size_t) &boot_page_table1));
}
```

File: ./dev/diskcache.c

```
1
2  #include <heap.h>
3  #include <stdlib.h>
4  #include <vfs.h>
5  #include <log.h>
6  #include <assert.h>
7  #include <virtual.h>
8  #include <errno.h>
```

```
#include <transfer.h>
#include <sys/stat.h>
#include <dirent.h>
#include <linkedlist.h>
#include <avl.h>
#include <semaphore.h>
#include <diskcache.h>

static int current_mode = DISKCACHE_NORMAL;
static struct linked_list* cache_list;
static struct semaphore* cache_list_lock = NULL;

struct cache_entry {
    size_t addr;
    size_t size;
    size_t lba;
};

struct cache_data {
    struct open_file* underlying_disk;
    int block_size;
    struct avl_tree* cache;
    struct semaphore* lock;
};

/*static*/ bool IsCacheCreationAllowed(void) {
    AcquireMutex(cache_list_lock, -1);
    bool retv = current_mode == DISKCACHE_NORMAL;
    ReleaseMutex(cache_list_lock);
    return retv;
}

static void SynchroniseEntry(struct cache_entry* entry) {
    // TODO: write to disk...
    (void) entry;
}

static int Read(struct vnode*, struct transfer*) {
    return 0;
}

static int Write(struct vnode*, struct transfer*) {
    return 0;
}

static uint8_t DirentType(struct vnode*) {
    return DT_BLK;
}

static int Stat(struct vnode* node, struct stat* st) {
    struct cache_data* data = node->data;
    return VnodeOpStat(data->underlying_disk->node, st);
}

static void TossCache(struct cache_data* data) {
    AcquireMutex(data->lock, -1);
    AvlTreeDestroy(data->cache);
    data->cache = AvlTreeCreate();
    ReleaseMutex(data->lock);
}

static void ReduceCache(struct cache_data* data) {
    AcquireMutex(data->lock, -1);
    // .. TODO: do something here...
    ReleaseMutex(data->lock);
}

static int Close(struct vnode* node) {
    TossCache(node->data);
    return 0;
}

static const struct vnode_operations dev_ops = {
    .read           = Read,
    .write          = Write,
    .dirent_type    = DirentType,
    .stat           = Stat,
    .close          = Close
};

void RemoveCacheEntryHandler(void* entry_) {
    struct cache_entry* entry = entry_;
    SynchroniseEntry(entry);
    UnmapVirt(entry->addr, entry->size);
}

struct open_file* CreateDiskCache(struct open_file* underlying_disk)
{
    struct vnode* node = CreateVnode(dev_ops);
    struct cache_data* data = AllocHeap(sizeof(struct cache_data));
    data->underlying_disk = underlying_disk;
    data->cache = AvlTreeCreate();
    data->lock = CreateMutex("vcache");
    data->block_size = 4096;        // TODO: max of block size and ARCH_PAGE_SIZE
    AvlTreeSetDeletionHandler(data->cache, RemoveCacheEntryHandler);
    node->data = data;

    struct open_file* cache = CreateOpenFile(node, underlying_disk->initial_mode, underlying_disk->flags, underlying_disk->can_read, underlying_disk->can_write);

    AcquireMutex(cache_list_lock, -1);
    LinkedListInsertEnd(cache_list, cache);
    ReleaseMutex(cache_list_lock);

    return cache;
}

static void ReduceCacheAmounts(bool toss) {
    struct linked_list_node* node = LinkedListGetFirstNode(cache_list);
    while (node != NULL) {
        struct cache_data* data = ((struct open_file*) LinkedListGetDataFromNode(node))->node->data;
        (toss ? TossCache : ReduceCache)(data);
        node = LinkedListGetNextNode(node);
    }
}

void SetDiskCaches(int mode) {
    /*
     * The PMM calls on allocation / free this before InitDiskCaches is called,
     * so need to guard here.
     */
    if (cache_list_lock == NULL) {
        return;
    }

    AcquireMutex(cache_list_lock, -1);

    if (mode == DISKCACHE_REDUCE && current_mode == DISKCACHE_NORMAL) {
        ReduceCacheAmounts(false);

    } else if (mode == DISKCACHE_TOSS && current_mode != DISKCACHE_TOSS) {
```

```
        ReduceCacheAmounts(true);
    }

    current_mode = mode;
    ReleaseMutex(cache_list_lock);
}

void InitDiskCaches(void) {
    cache_list = LinkedListCreate();
    cache_list_lock = CreateMutex("vclist");
}
```

File: ./dev/partition.c

```

#include <heap.h>
#include <stdlib.h>
#include <vfs.h>
#include <log.h>
#include <assert.h>
#include <errno.h>
#include <string.h>
#include <transfer.h>
#include <sys/stat.h>
#include <dirent.h>
#include <virtual.h>
#include <filesystem.h>

struct partition_data {
    struct open_file* fs;
    struct open_file* disk;
    int id;
    uint64_t start_byte;
    uint64_t length_bytes;
    int disk_bytes_per_sector;
    int media_type;
    bool boot;
};

static int Access(struct vnode* node, struct transfer* tr, bool write) {
    struct partition_data* partition = node->data;

    uint64_t start_addr = tr->offset + partition->start_byte;
    int64_t length = tr->length_remaining;
    if (tr->offset + tr->length_remaining > partition->length_bytes) {
        length = ((int64_t) partition->length_bytes) - ((int64_t) tr->offset);
    }

    struct transfer real_transfer = *tr;
    real_transfer.length_remaining = length;
    real_transfer.offset = start_addr;

    int res = (write ? WriteFile : ReadFile)(partition->disk, &real_transfer);

    uint64_t bytes_transferred = length - real_transfer.length_remaining;

    tr->offset += bytes_transferred;
    tr->length_remaining -= bytes_transferred;
    tr->address = ((uint8_t*) tr->address) + bytes_transferred;

    return res;
}

static int Read(struct vnode* node, struct transfer* tr) {
    return Access(node, tr, false);
}

static int Write(struct vnode* node, struct transfer* tr) {
    return Access(node, tr, true);
}

static bool IsSeekable(struct vnode*) {
    return true;
}

static int Create(struct vnode* node, struct vnode** fs, const char*, int flags, mode_t mode) {
    struct partition_data* partition = node->data;
    if (partition->fs != NULL) {
        return EALREADY;
    }

    partition->fs = CreateOpenFile(*fs, flags, mode, true, true);
    return 0;
}

static uint8_t DirentType(struct vnode*) {
    return DT_BLK;
}

static int Stat(struct vnode* node, struct stat* st) {
    struct partition_data* partition = node->data;

    LogWriteSerial("calling stat on a partition... bps = %d, len = %d\n", partition->disk_bytes_per_sector, partition->length_bytes);

    st->st_mode = S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO;
    st->st_atime = 0;
    st->st_blksize = partition->disk_bytes_per_sector;
    st->st_blocks = partition->length_bytes / partition->disk_bytes_per_sector;
    st->st_ctime = 0;
    st->st_dev = 0xBABECAFE;
    st->st_gid = 0;
    st->st_ino = 0xCAFEBABE;
    st->st_mtime = 0;
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = partition->length_bytes;
    st->st_uid = 0;
    return 0;
}

static int Follow(struct vnode* node, struct vnode** out, const char* name) {
    struct partition_data* partition = node->data;

    if (!strcmp(name, "fs")) {
        if (partition->fs == NULL) {
            return EINVAL;
        }

        *out = partition->fs->node;
        return 0;
    }

    return EINVAL;
}

static const struct vnode_operations dev_ops = {
    .is_seekable    = IsSeekable,
```

```
    .read           = Read,
    .write          = Write,
    .create         = Create,
    .follow         = Follow,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

struct open_file* CreatePartition(struct open_file* disk, uint64_t start, uint64_t length, int id, int sector_size, int media_type, bool boot) {
    struct partition_data* data = AllocHeap(sizeof(struct partition_data));
    data->disk = disk;
    data->disk_bytes_per_sector = sector_size;
    data->id = id;
    data->length_bytes = length;
    data->start_byte = start;
    data->media_type = media_type;
    data->boot = boot;
    data->fs = NULL;

    struct vnode* node = CreateVnode(dev_ops);
    node->data = data;

    struct open_file* partition = CreateOpenFile(node, 0, 0, true, true);
    LogWriteSerial("created the partition...\n");
    MountFilesystemForDisk(partition);
    return partition;
}

struct open_file* CreateMbrPartitionIfExists(struct open_file* disk, uint8_t* mem, int index, int sector_size) {
    int offset = 0x1BE + index * 16;

    uint8_t active = mem[offset + 0];
    if (active & 0x7F) {
        return NULL;
    }

    int media_type = mem[offset + 4];

    uint32_t start_sector = mem[offset + 11];
    start_sector <<= 8;
    start_sector |= mem[offset + 10];
    start_sector <<= 8;
    start_sector |= mem[offset + 9];
    start_sector <<= 8;
    start_sector |= mem[offset + 8];

    uint32_t total_sectors = mem[offset + 15];
    total_sectors <<= 8;
    total_sectors |= mem[offset + 14];
    total_sectors <<= 8;
    total_sectors |= mem[offset + 13];
    total_sectors <<= 8;
    total_sectors |= mem[offset + 12];

    if (start_sector == 0 && total_sectors == 0) {
        return NULL;
    }

    return CreatePartition(disk, ((uint64_t) start_sector) * sector_size, ((uint64_t) total_sectors) * sector_size, index, sector_size, media_type, active & 0x80
}

/*
 * caller to free return value.
 */
struct open_file** GetMbrPartitions(struct open_file* disk) {
    struct stat st;
    int res = VnodeOpStat(disk->node, &st);
    if (res != 0) {
        return NULL;
    }

    uint8_t* mem = (uint8_t*) MapVirt(0, 0, st.st_blksize, VM_READ | VM_FILE, disk, 0);
    if (mem == NULL) {
        return NULL;
    }

    if (mem[0x1FE] != 0x55) {
        return NULL;
    }
    if (mem[0x1FF] != 0xAA) {
        return NULL;
    }

    struct open_file** partitions = AllocHeap(sizeof(struct open_file) * 5);
    inline_memset(partitions, 0, sizeof(struct open_file) * 5);

    int partitions_found = 0;
    for (int i = 0; i < 4; ++i) {
        struct open_file* partition = CreateMbrPartitionIfExists(disk, mem, i, st.st_blksize);
        if (partition != NULL) {
            partitions[partitions_found++] = partition;
        }
    }

    UnmapVirt((size_t) mem, st.st_blksize);

    return partitions;
}

/*
 * null terminated array of struct vnode*
 * e.g. {vnode_ptr_1, vnode_ptr_2, vnode_ptr_3, NULL}
 */
struct open_file** GetPartitionsForDisk(struct open_file* disk) {
    struct open_file** partitions = GetMbrPartitions(disk);

    if (partitions == NULL) {
        // check for GPT
    }

    return partitions;
}
```

File: ./dev/random.c

```
#include <heap.h>
#include <stdlib.h>
#include <vfs.h>
#include <log.h>
#include <assert.h>
#include <errno.h>
#include <transfer.h>
#include <sys/stat.h>
#include <dirent.h>

static int Read(struct vnode*, struct transfer* io) {
    while (io->length_remaining > 0) {
        uint8_t random_byte = rand() & 0xFF;
        int err = PerformTransfer(&random_byte, io, 1);
        if (err) {
            return err;
        }
    }

    return 0;
}

static uint8_t DirentType(struct vnode*) {
    return DT_CHR;
}

static int Stat(struct vnode*, struct stat* st) {
    st->st_mode = S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO;
    st->st_atime = 0;
    st->st_blksize = 0;
    st->st_blocks = 0;
    st->st_ctime = 0;
    st->st_dev = 0xBABECAFE;
    st->st_gid = 0;
    st->st_ino = 0xCAFEBABE;
    st->st_mtime = 0;
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = 0;
    st->st_uid = 0;
    return 0;
}

static const struct vnode_operations dev_ops = {
    .read           = Read,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

void InitRandomDevice(void)
{
    AddVfsMount(CreateVnode(dev_ops), "rand");
}
```

File: ./dev/pty.c

```
 #include <heap.h>
 #include <stdlib.h>
 #include <vfs.h>
 #include <log.h>
 #include <assert.h>
 #include <irql.h>
 #include <errno.h>
 #include <string.h>
#include <transfer.h>
#include <sys/stat.h>
#include <dirent.h>
#include <panic.h>
#include <thread.h>
#include <termios.h>
#include <blockingbuffer.h>
#include <virtual.h>

#define INTERNAL_BUFFER_SIZE 256        // used to communicate with master and sub. can have any length - but lower means both input AND **PRINTING** will incur m
#define LINE_BUFFER_SIZE 300            // maximum length of a typed line
#define FLUSHED_BUFFER_SIZE 500         // used to store any leftover after pressing '\n' that the program has yet to read

struct pty_master_internal_data {
    struct vnode* subordinate;
    struct blocking_buffer* display_buffer;
    struct blocking_buffer* keybrd_buffer;
    struct blocking_buffer* flushed_buffer;
    struct thread* line_processing_thread;
};

struct pty_subordinate_internal_data {
    struct vnode* master;
    struct termios termios;
    char line_buffer[LINE_BUFFER_SIZE];
    uint8_t line_buffer_char_width[LINE_BUFFER_SIZE];
    int line_buffer_pos;
};

// "THE SCREEN"
static int MasterRead(struct vnode* node, struct transfer* tr) {
    struct pty_master_internal_data* internal = node->data;
    while (tr->length_remaining > 0) {
        char c = BlockingBufferGet(internal->display_buffer);
        PerformTransfer(&c, tr, 1);
    }

    return 0;
}

// "THE KEYBOARD"
static int MasterWrite(struct vnode* node, struct transfer* tr) {
    struct pty_master_internal_data* internal = node->data;

    while (tr->length_remaining > 0) {
        char c;
        PerformTransfer(&c, tr, 1);
        BlockingBufferAdd(internal->keybrd_buffer, c, true);
    }

    return 0;
}

static int MasterWait(struct vnode*, int, uint64_t) {
    return ENOSYS;
}

static int SubordinateWait(struct vnode*, int, uint64_t) {
    return ENOSYS;
}

```

```
71  static void FlushSubordinateLineBuffer(struct vnode* node) {
72      struct pty_subordinate_internal_data* internal = node->data;
73      struct pty_master_internal_data* master_internal = internal->master->data;
74
75      // could add a 'BlockingBufferAddMany' call?
76      for (int i = 0; i < internal->line_buffer_pos; ++i) {
77          BlockingBufferAdd(master_internal->flushed_buffer, internal->line_buffer[i], true);
78      }
79
80      internal->line_buffer_pos = 0;
81  }
82
83  static void RemoveFromSubordinateLineBuffer(struct vnode* node) {
84      struct pty_subordinate_internal_data* internal = node->data;
85
86      if (internal->line_buffer_pos == 0) {
87          return;
88      }
89
90      internal->line_buffer[--internal->line_buffer_pos] = 0;
91  }
92
93  static void AddToSubordinateLineBuffer(struct vnode* node, char c, int width) {
94      struct pty_subordinate_internal_data* internal = node->data;
95
96      if (internal->line_buffer_pos == LINE_BUFFER_SIZE) {
97          Panic(PANIC_NOT_IMPLEMENTED);
98          return;
99      }
100
101     internal->line_buffer[internal->line_buffer_pos] = c;
102     internal->line_buffer_char_width[internal->line_buffer_pos] = width;
103     internal->line_buffer_pos++;
104 }
105
106 static void LineProcessor(void* sub_) {
107     SetThreadPriority(GetThread(), SCHEDULE_POLICY_FIXED, FIXED_PRIORITY_KERNEL_HIGH);
108
109     struct vnode* node = (struct vnode*) sub_;
110     struct pty_subordinate_internal_data* internal = node->data;
111     struct pty_master_internal_data* master_internal = internal->master->data;
112
113     while (true) {
114         bool echo = internal->termios.c_lflag & ECHO;
115         bool canon = internal->termios.c_lflag & ICANON;
116
117         char c = BlockingBufferGet(master_internal->keybrd_buffer);
118
119         /*
120          * This must happen before we modify the line buffer (i.e. to add or backspace a character), as
121          * the backspace code here needs to check for a non-empty line (and so this must be done before we make
122          * the line empty).
123          */
124         if (echo) {
125             if (c == '\b' && canon) {
126                 if (internal->line_buffer_pos > 0) {
127                     BlockingBufferAdd(master_internal->display_buffer, '\b', true);
128                     BlockingBufferAdd(master_internal->display_buffer, ' ', true);
129                     BlockingBufferAdd(master_internal->display_buffer, '\b', true);
130                 }
131             } else {
132                 BlockingBufferAdd(master_internal->display_buffer, c, true);
133             }
134         }
135
136         if (c == '\b' && canon) {
137             RemoveFromSubordinateLineBuffer(node);
138
139         } else {
140             AddToSubordinateLineBuffer(node, c, 1);
141         }
142
143         if (c == '\n' || c == 3 || !canon) {
144             FlushSubordinateLineBuffer(node);
145         }
146     }
147 }
148
149 // "THE STDIN LINE BUFFER"
150 static int SubordinateRead(struct vnode* node, struct transfer* tr) {
151     struct pty_subordinate_internal_data* internal = (struct pty_subordinate_internal_data*) node->data;
152     struct pty_master_internal_data* master_internal = (struct pty_master_internal_data*) internal->master->data;
153
154     if (tr->length_remaining == 0) {
155         return 0;
156     }
157
158     char c = BlockingBufferGet(master_internal->flushed_buffer);
159     PerformTransfer(&c, tr, 1);
160
161     int res = 0;
162     while (tr->length_remaining > 0 && !(res = BlockingBufferTryGet(master_internal->flushed_buffer, (uint8_t*) &c))) {
163         PerformTransfer(&c, tr, 1);
164     }
165
166     return 0;
167 }
168
169 // "WRITING TO STDOUT"
170 static int SubordinateWrite(struct vnode* node, struct transfer* tr) {
171     struct pty_subordinate_internal_data* internal = (struct pty_subordinate_internal_data*) node->data;
172     struct pty_master_internal_data* master_internal = (struct pty_master_internal_data*) internal->master->data;
173
174     while (tr->length_remaining > 0) {
175         char c;
176         int err = PerformTransfer(&c, tr, 1);
177         if (err) {
178             return err;
179         }
180
181         BlockingBufferAdd(master_internal->display_buffer, c, true);
182     }
183
184     return 0;
185 }
186
187 static uint8_t GenericDirentType(struct vnode*) {
188     return DT_CHR;
189 }
190
191 static int GenericStat(struct vnode*, struct stat* st) {
192     st->st_mode = S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO;
193     st->st_atime = 0;
194     st->st_blksize = 0;
195     st->st_blocks = 0;
196     st->st_ctime = 0;
197     st->st_dev = 0xBABECAFE;
198     st->st_gid = 0;
199     st->st_ino = 0xCAFEBABE;
200     st->st_mtime = 0;
```

```
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = 0;
    st->st_uid = 0;
    return 0;
}

static int SubordinateCheckTty(struct vnode*) {
    return 0;
}

static const struct vnode_operations master_operations = {
    .read           = MasterRead,
    .write          = MasterWrite,
    .dirent_type    = GenericDirentType,
    .stat           = GenericStat,
    .wait           = MasterWait,
};

static const struct vnode_operations subordinate_operations = {
    .check_tty      = SubordinateCheckTty,
    .read           = SubordinateRead,
    .write          = SubordinateWrite,
    .dirent_type    = GenericDirentType,
    .stat           = GenericStat,
    .wait           = SubordinateWait,
};

void CreatePseudoTerminal(struct vnode** master, struct vnode** subordinate) {
    struct vnode* m = CreateVnode(master_operations);
    struct vnode* s = CreateVnode(subordinate_operations);

    struct pty_master_internal_data* m_data = AllocHeap(sizeof(struct pty_master_internal_data));
    struct pty_subordinate_internal_data* s_data = AllocHeap(sizeof(struct pty_subordinate_internal_data));

    m_data->subordinate = s;
    m_data->display_buffer = BlockingBufferCreate(INTERNAL_BUFFER_SIZE);
    m_data->keybrd_buffer = BlockingBufferCreate(INTERNAL_BUFFER_SIZE);
    m_data->flushed_buffer = BlockingBufferCreate(FLUSHED_BUFFER_SIZE);
    m_data->line_processing_thread = CreateThread(LineProcessor, (void*) s, GetVas(), "line processor");

    s_data->master = m;
    s_data->termios.c_lflag = ICANON | ECHO;

    m->data = m_data;
    s->data = s_data;
    *master = m;
    *subordinate = s;
}
```

File: ./dev/null.c

```

#include <heap.h>
#include <stdlib.h>
#include <vfs.h>
#include <log.h>
#include <assert.h>
#include <errno.h>
#include <transfer.h>
#include <sys/stat.h>
#include <dirent.h>

static int Read(struct vnode*, struct transfer*) {
    return 0;
}

static int Write(struct vnode*, struct transfer*) {
    // TODO: do we need to set io->length_remaining to zero?
    return 0;
}

static uint8_t DirentType(struct vnode*) {
    return DT_CHR;
}

static int Stat(struct vnode*, struct stat* st) {
    st->st_mode = S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO;
    st->st_atime = 0;
    st->st_blksize = 0;
    st->st_blocks = 0;
    st->st_ctime = 0;
    st->st_dev = 0xBABECAFE;
    st->st_gid = 0;
    st->st_ino = 0xCAFEBABE;
    st->st_mtime = 0;
    st->st_nlink = 1;
    st->st_rdev = 0xCAFEDEAD;
    st->st_size = 0;
    st->st_uid = 0;
    return 0;
}

static const struct vnode_operations dev_ops = {
    .read           = Read,
    .write          = Write,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

void InitNullDevice(void)
{
    AddVfsMount(CreateVnode(dev_ops), "null");
}
```

File: ./sync/spinlock.c

```c
#include <spinlock.h>
#include <string.h>
#include <arch.h>
#include <irql.h>
#include <panic.h>
#include <log.h>
#include <thread.h>
#include <assert.h>

void InitSpinlock(struct spinlock* lock, const char* name, int irql) {
    assert(strlen(name) <= 15);

    if (irql < IRQL_SCHEDULER) {
        Panic(PANIC_SPINLOCK_WRONG_IRQL);
    }

    lock->lock = 0;
    lock->owner = NULL;
    lock->irql = irql;
    strcpy(lock->name, name);
}

void AcquireSpinlockDirect(struct spinlock* lock) {
    if (lock->lock != 0) {
        LogWriteSerial("OOPS! %s\n", lock->name);
        Panic(PANIC_SPINLOCK_DOUBLE_ACQUISITION);
    }
    assert(lock->lock == 0);

    ArchSpinlockAcquire(&lock->lock);
    //lock->owner = GetThread();
}

void ReleaseSpinlockDirect(struct spinlock* lock) {
    if (lock->lock == 0) {
        Panic(PANIC_SPINLOCK_RELEASED_BEFORE_ACQUIRED);
    }
    assert(lock->lock != 0);
    //assert(lock->owner == GetThread() || lock->irql == IRQL_HIGH);
    //lock->owner = NULL;
    ArchSpinlockRelease(&lock->lock);
}

/**
 * This function has no atomic guarantees. It should only be used for debugging and writing
 * assertion statements.
 */
bool IsSpinlockHeld(struct spinlock* lock) {
    return lock->lock;
}

int AcquireSpinlockIrql(struct spinlock* lock) {
    assert(lock->lock == 0);

    int prior_irql = GetIrql();
    RaiseIrql(lock->irql);

    if (lock->irql != GetIrql()) {
        Panic(PANIC_SPINLOCK_WRONG_IRQL);
    }

    AcquireSpinlockDirect(lock);
    lock->prev_irql = prior_irql;
    return prior_irql;
}

void ReleaseSpinlockIrql(struct spinlock* lock) {
    int old_irql = lock->prev_irql;
    ReleaseSpinlockDirect(lock);
    LowerIrql(old_irql);
}
```

File: ./sync/semaphore.c

```c

 #include <thread.h>
 #include <semaphore.h>
 #include <threadlist.h>
 #include <heap.h>
 #include <errno.h>
 #include <string.h>
 #include <irql.h>
 #include <timer.h>
#include <assert.h>
#include <panic.h>
#include <log.h>

struct semaphore {
    const char* name;
    int max_count;
    int current_count;
    struct thread_list waiting_list;
};

/**
 * Creates a semaphore object with a specified limit on the number of concurrent holders.
 *
 * @param max_count         The maximum number of concurrent holders of the semaphore
 * @param initial_count     The initial number of holders of the semaphore. Should usually either be 0,
 *                              which is a 'acquire until full' state, or equal to `max_count`, which is
 *                              a `it's full until released` state.
 * @returns The initialised semaphore.
 *
 * @maxirql IRQL_SCHEDULER
 */
struct semaphore* CreateSemaphore(const char* name, int max_count, int initial_count) {
    MAX_IRQL(IRQL_SCHEDULER);

    struct semaphore* sem = AllocHeap(sizeof(struct semaphore));
    sem->name = name;
    sem->max_count = max_count;
    sem->current_count = initial_count;
    ThreadListInit(&sem->waiting_list, NEXT_INDEX_SEMAPHORE);
    return sem;
}

/**
 * Acquires (i.e. does the waits or P operation on) a semaphore. This operation my block depending on the timeout value.
 *
 * @param sem           The semaphore to acquire.
 * @param timeout_ms    One of either:
 *                           0: Attempt to acquire the semaphore, but will not block if it cannot be acquired.
 *                          -1: Will acquire semaphore, even if it needs to block to do so. Will not timeout.
 *                          +N: Same as -1, except that the operation will timeout after the specified number of
 *                              milliseconds.
 *
```

```
 * @return 0 if the semaphore was acquired
 *         ETIMEDOUT if the semaphore was not acquired, and the operation timed out
 *         EAGAIN if the semaphore was not acquired, and the timeout_ms value was 0
 *
 * @maxirql IRQL_PAGE_FAULT
 */
int AcquireSemaphore(struct semaphore* sem, int timeout_ms) {
    MAX_IRQL(IRQL_PAGE_FAULT);
    assert(sem != NULL);

    LockScheduler();

    struct thread* thr = GetThread();
    if (thr == NULL) {
        if (sem->current_count < sem->max_count) {
            sem->current_count++;
        } else {
            Panic(PANIC_SEM_BLOCK_WITHOUT_THREAD);
        }
        UnlockScheduler();
        return 0;
    }

    /*
     * This gets set to true by the sleep wakeup routine if we get timed-out.
     */
    thr->timed_out = false;

    if (sem->current_count < sem->max_count) {
        /*
         * Uncontested, so acquire straight away.
         */
        sem->current_count++;
    } else {
        /*
         * Need to block for the semaphore (or return if the timeout is zero).
         */
        thr->waiting_on_semaphore = sem;

        if (timeout_ms == 0) {
            thr->timed_out = true;

        } else if (timeout_ms == -1) {
            ThreadListInsert(&sem->waiting_list, thr);
            BlockThread(THREAD_STATE_WAITING_FOR_SEMAPHORE);

        } else {
            ThreadListInsert(&sem->waiting_list, thr);
            thr->sleep_expiry = GetSystemTimer() + ((uint64_t) timeout_ms) * 1000ULL * 1000ULL;
            QueueForSleep(thr);
            BlockThread(THREAD_STATE_WAITING_FOR_SEMAPHORE_WITH_TIMEOUT);
        }
    }

    UnlockScheduler();

    return thr->timed_out ? (timeout_ms == 0 ? EAGAIN : ETIMEDOUT) : 0;
}

/**
 * Releases (i.e., does the signal, or V operation on) a semaphore. If there are threads waiting on this semaphore,
 * it will cause the first one to wake up.
 *
 * @param sem The semaphore to release/signal
 */
void ReleaseSemaphore(struct semaphore* sem) {
    MAX_IRQL(IRQL_PAGE_FAULT);

    LockScheduler();
    assert(sem->current_count > 0);

    if (sem->waiting_list.head == NULL) {
        if (sem->current_count == 0) {
            Panic(PANIC_NEGATIVE_SEMAPHORE);
        }
        sem->current_count--;

    } else {
        struct thread* top = ThreadListDeleteTop(&sem->waiting_list);

        /*
         * If it's in the THREAD_STATE_WAITING_FOR_SEMAPHORE_WITH_TIMEOUT state, it could mean one of two things:
         *      - it's still on the sleep queue, in which case we need to get it off that queue, and put it on the ready queue
         *      - it's been taken off the sleep queue and onto the ready already, but it hasn't yet been run yet (and is therefore
         *        still in this state)
         */
        if (top->state == THREAD_STATE_WAITING_FOR_SEMAPHORE_WITH_TIMEOUT) {
            bool on_sleep_queue = TryDequeueForSleep(top);

            if (on_sleep_queue) {
                /*
                 * Change the state to prevent UnblockThread from seeing it's in the timeout state and calling CancelSemaphoreOfThread.
                 * If CancelSemaphoreOfThread were called, then it would attempt to delete it from the queue - but it's already been
                 * deleted by this point and so would crash.
                 */
                top->state = THREAD_STATE_READY;
                UnblockThread(top);
            }
            /*
             * Do not unblock the thread if it's not on the sleep queue, as not being on the sleep queue means it's
             * already on the ready queue.
             */

        } else {
            UnblockThread(top);
        }
    }
    UnlockScheduler();
}

/**
 * Deallocates a semaphore. Loses its shit if someone is still holding onto it, as it is probably a bug if you're trying
 * to destroy a semaphore when there's a possiblity that someone might even be thinking about trying to acquire it (which
 * would then try to acquire a deleted memory region, which is very bad).
 *
 * @param sem The semaphore to destroy.
 * @param flags One of SEM_DONT_CARE, SEM_REQUIRE_ZERO or SEM_REQUIRE_FULL.
 *
 * @maxirql IRQL_SCHEDULER
 */
int DestroySemaphore(struct semaphore* sem, int flags) {
    MAX_IRQL(IRQL_SCHEDULER);

    LockScheduler();
    if (flags == SEM_REQUIRE_ZERO && sem->current_count != 0) {
        UnlockScheduler();
        return EBUSY;
    }
    if (flags == SEM_REQUIRE_FULL && sem->current_count != sem->max_count) {
        UnlockScheduler();
```

```
        return EBUSY;
    }

    FreeHeap(sem);
    UnlockScheduler();
    return 0;
}

/**
 * Internal function. Used by the sleep wakeup routine to cancel a semaphore that has been timed-out.
 * Removes the thread from the semaphore wait list. If this does not occur, then the sleep wakeup routine will allow
 * the thread to continue running, which will lead to a crash if that thread then attempts to acquire the same semaphore.
 * Without this, stress tests will crash.
 */
void CancelSemaphoreOfThread(struct thread* thr) {
    AssertSchedulerLockHeld();
    assert(ThreadListContains(&thr->waiting_on_semaphore->waiting_list, thr));
    ThreadListDelete(&thr->waiting_on_semaphore->waiting_list, thr);
}

int GetSemaphoreCount(struct semaphore* sem) {
    return sem->current_count;
}
```

File: ./merged.pdf

[binary]

File: ./fs/.DS_Store

[binary]

File: ./fs/filesystem.c

```
1
2  #include <common.h>
3  #include <vfs.h>
4  #include <errno.h>
5  #include <string.h>
6  #include <spinlock.h>
7  #include <irql.h>
8  #include <diskutil.h>
9  #include <semaphore.h>
10 #include <log.h>
11 #include <filesystem.h>
12 #include <heap.h>
13
14 #include <fs/demofs/demofs.h>
15
16 #define MAX_REGISTERED_FILESYSTEMS 8
17
18 struct filesystem {
19     char* name;
20     fs_mount_creator mount_creator;
21 };
22
23 static struct filesystem registered_filesystems[MAX_REGISTERED_FILESYSTEMS];
24 static int num_filesystems = 0;
25 static struct semaphore* fs_table_lock;
26
27 void InitFilesystemTable(void) {
28     num_filesystems = 0;
29     fs_table_lock = CreateMutex("fs table");
30     RegisterFilesystem("demofs", DemofsMountCreator);
31 }
32
33 int RegisterFilesystem(char* fs_name, fs_mount_creator mount) {
34     if (fs_name == NULL || mount == NULL) {
35         return EINVAL;
36     }
37
38     struct filesystem fs;
39     fs.name = strdup_pageable(fs_name);
40     fs.mount_creator = mount;
41
42     int ret = 0;
43
44     AcquireMutex(fs_table_lock, -1);
45     if (num_filesystems < MAX_REGISTERED_FILESYSTEMS) {
46         registered_filesystems[num_filesystems++] = fs;
47     } else {
48         ret = EALREADY;
49     }
50     registered_filesystems[num_filesystems++] = fs;
51     ReleaseMutex(fs_table_lock);
52
53     if (ret != 0) {
54         FreeHeap(fs.name);
55     }
56
57     return ret;
58 }
59
60 int MountFilesystemForDisk(struct open_file* partition) {
61     LogWriteSerial("mouting filesystem for disk...\n");
62
63     AcquireMutex(fs_table_lock, -1);
64
65     struct open_file* fs = NULL;
66
67     for (int i = 0; i < num_filesystems; ++i) {
68         fs = NULL;
69         int res = registered_filesystems[i].mount_creator(partition, &fs);
70         if (res == 0) {
71             break;
72         }
73     }
74
75     ReleaseMutex(fs_table_lock);
76
77     if (fs == NULL) {
78         return ENODEV;
79     }
80
81     int res = VnodeOpCreate(partition->node, &fs->node, "fs", 0, 0);
82     if (res != 0) {
83         return res;
84     }
85
86     AddVfsMount(fs->node, GenerateNewMountedDiskName());
87     return 0;
88 }
```

File: ./fs/demofs/demofs_private.h

```
#pragma once

#include <sys/types.h>
#include <common.h>
#include <vfs.h>
#include <transfer.h>

struct demofs {
struct open_file* disk;
ino_t root_inode;
};

#define MAX_NAME_LENGTH 24

#define INODE_TO_SECTOR(inode) (inode & 0xFFFFFF)
#define INODE_IS_DIR(inode) (inode >> 31)
#define INODE_TO_DIR(inode) (inode | (1U << 31U))

int demofs_read_file(struct demofs* fs, ino_t file, uint32_t file_size, struct transfer* io);
int demofs_read_directory_entry(struct demofs* fs, ino_t directory, struct transfer* io);
int demofs_follow(struct demofs* fs, ino_t parent, ino_t* child, const char* name, uint32_t* file_length_out);
```

File: ./fs/demofs/demofs_inodes.c

```
1
2  #include <common.h>
3  #include <errno.h>
4  #include <vfs.h>
5  #include <string.h>
```

```
#include <assert.h>
#include <panic.h>
#include <transfer.h>
#include <log.h>
#include <sys/types.h>
#include <dirent.h>
#include <fs/demofs/demofs_private.h>

#define SECTOR_SIZE 512

/*
* We are going to use the high bit of an inode ID to indicate whether or not
* we are talking about a directory or not (high bit set = directory).
*
* This allows us to, for example, easily catch ENOTDIR in demofs_follow.
*
* Remember to use INODE_TO_SECTOR. Note that inodes are only stored using 24 bits
* anyway.
*/
int demofs_read_inode(struct demofs* fs, ino_t inode, uint8_t* buffer) {
    struct transfer io = CreateKernelTransfer(buffer, SECTOR_SIZE, SECTOR_SIZE * INODE_TO_SECTOR(inode), TRANSFER_READ);
    return ReadFile(fs->disk, &io);
}

int demofs_read_file(struct demofs* fs, ino_t file, uint32_t file_size_left, struct transfer* io) {
    if (io->offset >= file_size_left) {
        return 0;
    }

    file_size_left -= io->offset;

    while (io->length_remaining != 0 && file_size_left != 0) {
        int sector = file + io->offset / SECTOR_SIZE;
        int sector_offset = io->offset % SECTOR_SIZE;

        if (sector_offset == 0 && io->length_remaining >= SECTOR_SIZE && file_size_left >= SECTOR_SIZE) {
            /*
            * We have an aligned sector amount, so transfer it all directly,
            * execpt for possible a few bytes at the end.
            *
            * ReadFile only allows lengths that are a multiple of the sector
            * size, so round down to the nearest sector. The remainder must be
            * kept track of so it can be added back on after the read.
            */

            int remainder = io->length_remaining % SECTOR_SIZE;
            io->length_remaining -= remainder;

            /*
            * We need the disk offset, not the file offset.
            * Ensure we move it back though afterwards.
            */
            int delta = sector * SECTOR_SIZE - io->offset;
            io->offset += delta;

            int status = ReadFile(fs->disk, io);
            if (status != 0) {
                return status;
            }

            io->offset -= delta;
            io->length_remaining = remainder;
            file_size_left -= SECTOR_SIZE;      // ???!

        } else {
            /*
            * A partial sector transfer.
            *
            * We must read the sector into an internal buffer, and then copy a
            * subsection of that to the return buffer.
            */
            uint8_t sector_buffer[SECTOR_SIZE];

            /* Read the sector */
            struct transfer temp_io = CreateKernelTransfer(sector_buffer, SECTOR_SIZE, sector * SECTOR_SIZE, TRANSFER_READ);

            int status = ReadFile(fs->disk, &temp_io);
            if (status != 0) {
                return status;
            }

            /* Transfer to the correct buffer */
            size_t transfer_size = MIN(MIN(SECTOR_SIZE - (io->offset % SECTOR_SIZE), io->length_remaining), file_size_left);
            PerformTransfer(sector_buffer + (io->offset % SECTOR_SIZE), io, transfer_size);
            file_size_left -= transfer_size;
        }
    }

    return 0;
}

int demofs_follow(struct demofs* fs, ino_t parent, ino_t* child, const char* name, uint32_t* file_length_out) {
    assert(fs);
    assert(fs->disk);
    assert(child);
    assert(name);
    assert(file_length_out);
    assert(SECTOR_SIZE % 32 == 0);

    uint8_t buffer[SECTOR_SIZE];

    if (strlen(name) > MAX_NAME_LENGTH) {
        return ENAMETOOLONG;
    }

    if (!INODE_IS_DIR(parent)) {
        return ENOTDIR;
    }

    /*
    * The directory may contain many entries, so we need to iterate through them.
    */
    while (true) {
        /*
        * Grab the current entry.
        */
        int status = demofs_read_inode(fs, parent, buffer);
        if (status != 0) {
            return status;
        }

        /*
        * Something went very wrong if the directory header is not present!
        */
        if (buffer[0] != 0xFF && buffer[0] != 0xFE) {
            return EIO;
        }

        for (int i = 1; i < SECTOR_SIZE / 32; ++i) {
            /*
```

```
            * Check if there are no more names in the directory.
            */
            if (buffer[i * 32] == 0) {
                return ENOENT;
            }

            /*
            * Check if we've found the name.
            */
            if (!strncmp(name, (char*) buffer + i * 32, MAX_NAME_LENGTH)) {
                /*
                * If so, read the inode number and return it.
                * Remember to add the directory flag if necessary.
                */
                ino_t inode = buffer[i * 32 + MAX_NAME_LENGTH + 4];
                inode |= (ino_t) buffer[i * 32 + MAX_NAME_LENGTH + 5] << 8;
                inode |= (ino_t) buffer[i * 32 + MAX_NAME_LENGTH + 6] << 16;

                if (buffer[i * 32 + MAX_NAME_LENGTH + 7] & 1) {
                    /*
                    * This is a directory.
                    */
                    inode = INODE_TO_DIR(inode);
                    *file_length_out = 0;

                } else {
                    /*
                    * This is a file.
                    */
                    uint32_t length = buffer[i * 32 + MAX_NAME_LENGTH];
                    length |= (uint32_t) buffer[i * 32 + MAX_NAME_LENGTH + 1] << 8;
                    length |= (uint32_t) buffer[i * 32 + MAX_NAME_LENGTH + 2] << 16;
                    length |= (uint32_t) buffer[i * 32 + MAX_NAME_LENGTH + 3] << 24;

                    *file_length_out = length;
                }

                *child = inode;
                return 0;
            }
        }

        /*
        * Now we need to move on to the next entry if there is one.
        */
        if (buffer[0] == 0xFF) {
            /* No more entries. */
            return ENOENT;

        } else if (buffer[0] == 0xFE) {
            /*
            * There is another entry, so read its inode and keep the loop going
            */
            parent = buffer[1];
            parent |= (ino_t) buffer[2] << 8;
            parent |= (ino_t) buffer[3] << 16;

            /*
            * Add the directory bit to the inode number as it should be a directory.
            */
            parent = INODE_TO_DIR(parent);

        } else {
            /*
            * Something went very wrong if the directory header is not present!
            */
            return EIO;
        }
    }
}

int demofs_read_directory_entry(struct demofs* fs, ino_t directory, struct transfer* io) {
    if (!INODE_IS_DIR(directory)) {
        return ENOTDIR;
    }

    assert(SECTOR_SIZE % 32 == 0);
    uint8_t buffer[SECTOR_SIZE];

    struct dirent dir;

    if (io->offset % sizeof(struct dirent) != 0) {
        return EINVAL;
    }

    int entry_number = io->offset / sizeof(struct dirent);

    /*
    * Each directory inode contains 31 files, and a pointer to the next directory entry.
    * Add 1 to the offset to skip past the header.
    */
    int indirections = entry_number / 31;
    int offset = entry_number % 31 + 1;

    /*
    * Get the correct inode
    */
    int status = 0;
    ino_t current_inode = directory;
    for (int i = 0; i < indirections; ++i) {
        status = demofs_read_inode(fs, current_inode, buffer);
        if (status != 0) {
            return status;
        }

        /*
        * Check for end of directory.
        */
        if (buffer[0] == 0xFF) {
            return 0;
        }

        if (buffer[0] != 0xFE) {
            return EIO;
        }

        /*
        * Get the next in the chain.
        */
        current_inode = buffer[1];
        current_inode |= (ino_t) buffer[2] << 8;
        current_inode |= (ino_t) buffer[3] << 16;
    }

    status = demofs_read_inode(fs, current_inode, buffer);
    if (status != 0) {
        return status;
    }

    /*
```

```
    * Check if we've gone past the end of the directory.
    */
    if (buffer[offset * 32] == 0) {
        return 0;
    }

    /*
    * strncpy is a bit iffy, so let's just play it safe.
    */
    char name[MAX_NAME_LENGTH + 2];
    memset(name, 0, MAX_NAME_LENGTH + 2);
    strncpy(name, (char*) buffer + offset * 32, MAX_NAME_LENGTH);
    strcpy(dir.d_name, name);

    dir.d_namlen = strlen(name);

    ino_t inode = buffer[offset * 32 + MAX_NAME_LENGTH + 4];
    inode |= (ino_t) buffer[offset * 32 + MAX_NAME_LENGTH + 5] << 8;
    inode |= (ino_t) buffer[offset * 32 + MAX_NAME_LENGTH + 6] << 16;

    dir.d_ino = inode;
    dir.d_type = INODE_IS_DIR(inode) ? DT_DIR : DT_REG;

    /* Perform the transfer to the correct location */
    return PerformTransfer(&dir, io, sizeof(struct dirent));
}
```

File: ./fs/demofs/demofs.h

```
#pragma once

#include <vfs.h>
#include <common.h>
#include <filesystem.h>

int DemofsMountCreator(struct open_file* raw_device, struct open_file** out);
```

File: ./fs/demofs/demofs_vnode.c

```
1
2  #include <heap.h>
3  #include <log.h>
4  #include <assert.h>
5  #include <errno.h>
6  #include <fcntl.h>
7  #include <vfs.h>
8  #include <transfer.h>
9  #include <string.h>
10 #include <dirent.h>
11 #include <sys/stat.h>
12 #include <sys/types.h>
13 #include <fs/demofs/demofs_private.h>
14
15 struct vnode_data {
16     ino_t inode;
17     struct demofs fs;
18     uint32_t file_length;
19     bool directory;
20 };
21
22 static int CheckOpen(struct vnode*, const char* name, int flags) {
23     if (strlen(name) >= MAX_NAME_LENGTH) {
24         return ENAMETOOLONG;
25     }
26
27     if ((flags & O_ACCMODE) == O_WRONLY || (flags & O_ACCMODE) == O_RDWR) {
28         return EROFS;
29     }
30
31     return 0;
32 }
33
34 static int Ioctl(struct vnode*, int, void*) {
35     return EINVAL;
36 }
37
38 static bool IsSeekable(struct vnode*) {
39     return true;
40 }
41
42 static int CheckTty(struct vnode*) {
43     return ENOTTY;
44 }
45
46 static int Read(struct vnode* node, struct transfer* io) {
47     struct vnode_data* data = node->data;
48     if (data->directory) {
49         return demofs_read_directory_entry(&data->fs, data->inode, io);
50     } else {
51         return demofs_read_file(&data->fs, data->inode, data->file_length, io);
52     }
53 }
54
55 static int Write(struct vnode*, struct transfer*) {
56     return EROFS;
57 }
58
59 static int Create(struct vnode*, struct vnode**, const char*, int, mode_t) {
60     return EROFS;
61 }
62
63 static uint8_t DirentType(struct vnode* node) {
64     struct vnode_data* data = node->data;
65     return data->directory ? DT_DIR : DT_REG;
66 }
67
68 static int Stat(struct vnode* node, struct stat* stat) {
69     struct vnode_data* data = node->data;
70
71     stat->st_atime = 0;
72     stat->st_blksize = 512;
73     stat->st_blocks = 0;
74     stat->st_ctime = 0;
75     stat->st_dev = 0xDEADDEAD;
76     stat->st_gid = 0;
77     stat->st_ino = data->inode;
78     stat->st_mode = (INODE_IS_DIR(data->inode) ? S_IFDIR : S_IFREG) | S_IRWXU | S_IRWXG | S_IRWXO;
79     stat->st_mtime = 0;
80     stat->st_nlink = 1;
81     stat->st_rdev = 0;
82     stat->st_size = data->file_length;
83     stat->st_uid = 0;
84
85     return 0;
```

```
}

static int Truncate(struct vnode*, off_t) {
    return EROFS;
}

static int Close(struct vnode* node) {
    FreeHeap(node->data);
    return 0;
}

static struct vnode* CreateDemoFsVnode();

static int Follow(struct vnode* node, struct vnode** out, const char* name) {
    struct vnode_data* data = node->data;
    if (data->directory) {
        ino_t child_inode;
        uint32_t file_length;

        int status = demofs_follow(&data->fs, data->inode, &child_inode, name, &file_length);
        if (status != 0) {
            return status;
        }

        /*
        * TODO: return existing vnode if someone opens the same file twice...
        */

        struct vnode* child_node = CreateDemoFsVnode();
        struct vnode_data* child_data = AllocHeap(sizeof(struct vnode_data));
        child_data->inode = child_inode;
        child_data->fs = data->fs;
        child_data->file_length = file_length;
        child_data->directory = INODE_IS_DIR(child_inode);

        child_node->data = child_data;

        *out = child_node;

        return 0;

    } else {
        return ENOTDIR;
    }
}
static const struct vnode_operations dev_ops = {
    .check_open     = CheckOpen,
    .ioctl          = Ioctl,
    .is_seekable    = IsSeekable,
    .check_tty      = CheckTty,
    .read           = Read,
    .write          = Write,
    .close          = Close,
    .truncate       = Truncate,
    .create         = Create,
    .follow         = Follow,
    .dirent_type    = DirentType,
    .stat           = Stat,
};

static struct vnode* CreateDemoFsVnode() {
    return CreateVnode(dev_ops);
}

static int CheckForDemofsSignature(struct open_file* raw_device) {
    struct stat st;
VnodeOpStat(raw_device->node, &st);

    uint8_t* buffer = AllocHeapEx(st.st_blksize, HEAP_ALLOW_PAGING);
    struct transfer io = CreateKernelTransfer(buffer, st.st_blksize, 8 * st.st_blksize, TRANSFER_READ);
    int res = ReadFile(raw_device, &io);
    if (res != 0) {
        FreeHeap(buffer);
        return ENOTSUP;
    }

    /*
     * Check for the DemoFS signature.
     */
    if (buffer[0] != 'D' || buffer[1] != 'E' || buffer[2] != 'M' || buffer[3] != 'O') {
        FreeHeap(buffer);
        return ENOTSUP;
    }

    return 0;
}

int DemofsMountCreator(struct open_file* raw_device, struct open_file** out) {
 int sig_check = CheckForDemofsSignature(raw_device);
    if (sig_check != 0) {
        return sig_check;
    }

 struct vnode* node = CreateDemoFsVnode();
    struct vnode_data* data = AllocHeap(sizeof(struct vnode_data));

    data->fs.disk = raw_device;
    data->fs.root_inode = 9 | (1 << 31);
    data->inode = 9 | (1 << 31);        /* root directory inode */
    data->file_length = 0;              /* root directory has no length */
    data->directory = true;

    node->data = data;

 *out = CreateOpenFile(node, 0, 0, true, false);
    return 0;
}
```

File: ./thread/thread.c

```
1
2  #include <cpu.h>
3  #include <thread.h>
4  #include <spinlock.h>
5  #include <heap.h>
6  #include <assert.h>
7  #include <timer.h>
8  #include <string.h>
9  #include <irql.h>
10 #include <log.h>
11 #include <errno.h>
12 #include <virtual.h>
13 #include <panic.h>
14 #include <common.h>
15 #include <threadlist.h>
```

```
#include <avl.h>
#include <priorityqueue.h>
#include <progload.h>
#include <semaphore.h>
#include <process.h>

static struct thread_list ready_list;
static struct spinlock scheduler_lock;
static struct spinlock innermost_lock;

/*
* Local fixed sized arrays and variables need to fit on the kernel stack.
* Allocate at least 8KB (depending on the system page size).
*
* Please note that overflowing the kernel stack into non-paged memory will lead to
* an immediate and unrecoverable crash on most systems.
*/
#define DEFAULT_KERNEL_STACK_KB   BytesToPages(1024 * 16) * ARCH_PAGE_SIZE / 1024

/*
* The user stack is allocated as needed - this is the maximum size of the stack in
* user virtual memory. (However, a larger max stack means more page tables need to be
* allocated to store it - even if there are no actual stack pages in yet).
*
* On x86, allocating a 4MB region only requires one page table, hence we'll use that.
*/
#define USER_STACK_MAX_SIZE BytesToPages(1024 * 1024 * 4) * ARCH_PAGE_SIZE


#define TIMESLICE_LENGTH_MS 50

/*
* Kernel stack overflow normally results in a total system crash/reboot because
* fault handlers will not work (they push data to a non-existent stack!).
*
* We will fill pages at the end of the stack with a certain value (CANARY_VALUE),
* and then we can check if they have been modified. If they are, we will throw a
* somewhat nicer error than a system reboot.
*
* Note that we can still overflow 'badly' if someone makes an allocation on the
* stack lwhich is larger than the remaining space on the stack and the canary size
* combined.
*
* If the canary page is only partially used for the canary, the remainder of the
* page is able to be used normally.
*/
#ifdef NDEBUG
#define NUM_CANARY_PAGES 0
#else
#define NUM_CANARY_BYTES (1024 * 8)
#define NUM_CANARY_PAGES BytesToPages(NUM_CANARY_BYTES)
#define CANARY_VALUE     0x8BADF00D

static void CheckCanary(size_t canary_base) {
    uint32_t* canary_ptr = (uint32_t*) canary_base;

    for (size_t i = 0; i < NUM_CANARY_BYTES / sizeof(uint32_t); ++i) {
        if (*canary_ptr++ != CANARY_VALUE) {
            Panic(PANIC_CANARY_DIED);
        }
    }
}

static void CreateCanary(size_t canary_base) {
    uint32_t* canary_ptr = (uint32_t*) canary_base;

    for (size_t i = 0; i < NUM_CANARY_BYTES / sizeof(uint32_t); ++i) {
        *canary_ptr++ = CANARY_VALUE;
    }
}

#endif


/*
* Allocates a new page-aligned stack for a kernel thread, and returns
* the address of either the top of the stack (if it grows downward),
* or the bottom (if it grows upward).
*/
static void CreateKernelStacks(struct thread* thr, int kernel_stack_kb) {
    int total_bytes = (BytesToPages(kernel_stack_kb * 1024) + NUM_CANARY_PAGES) * ARCH_PAGE_SIZE;

    size_t stack_bottom = MapVirt(0, 0, total_bytes, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
    size_t stack_top = stack_bottom + total_bytes;

#ifndef NDEBUG
    thr->canary_position = stack_bottom;
    CreateCanary(stack_bottom);
#endif
    thr->kernel_stack_top = stack_top;
    thr->kernel_stack_size = total_bytes;

    thr->stack_pointer = ArchPrepareStack(thr->kernel_stack_top);
}

static size_t CreateUserStack(int size) {
    /*
    * All user stacks share the same area of virtual memory, but have different
    * mappings to physical memory.
    */

    int total_bytes = BytesToPages(size) * ARCH_PAGE_SIZE;
    size_t stack_base = ARCH_USER_STACK_LIMIT - total_bytes;
    size_t actual_base = MapVirt(0, stack_base, total_bytes, VM_READ | VM_WRITE | VM_USER | VM_LOCAL, NULL, 0);

    assert(stack_base == actual_base);
    (void) actual_base;     // the assert gets taken out on release mode, so make the compiler happy

    return ARCH_USER_STACK_LIMIT;
}

/**
 * Blocks the currently executing thread (no effect will happen until the IRQL goes below IRQL_SCHEDULER).
 * The scheduler lock must be held.
 */

void BlockThread(int reason) {
    AssertSchedulerLockHeld();
    assert(reason != THREAD_STATE_READY && reason != THREAD_STATE_RUNNING);
    assert(GetCpu() != NULL && GetCpu()->current_thread != NULL);
    assert(GetCpu()->current_thread->state == THREAD_STATE_RUNNING);
    GetCpu()->current_thread->state = reason;
    PostponeScheduleUntilStandardIrql();
}

void UnblockThread(struct thread* thr) {
    AssertSchedulerLockHeld();
    if (thr->state == THREAD_STATE_WAITING_FOR_SEMAPHORE_WITH_TIMEOUT) {
        CancelSemaphoreOfThread(thr);
    }
```

```
    ThreadListInsert(&ready_list, thr);
    if (thr->priority < GetThread()->priority) {
        PostponeScheduleUntilStandardIrql();
    }
}

void UpdateThreadTimeUsed(void) {
    static uint64_t prev_time = 0;

    uint64_t time = GetSystemTimer();
    uint64_t time_elapsed = GetSystemTimer() - prev_time;
    prev_time = time;

    GetThread()->time_used += time_elapsed;
}

static int GetNextThreadId(void) {
    static struct spinlock thread_id_lock;
    static int next_thread_id = 0;
    static bool initialised = false;

    if (!initialised) {
        initialised = true;
        InitSpinlock(&thread_id_lock, "thread id", IRQL_SCHEDULER);
    }

    AcquireSpinlockIrql(&thread_id_lock);
    int result = next_thread_id++;
    ReleaseSpinlockIrql(&thread_id_lock);
    return result;
}

struct thread* CreateThreadEx(void(*entry_point)(void*), void* argument, struct vas* vas, const char* name, struct process* prcss, int policy, int priority, int
    struct thread* thr = AllocHeap(sizeof(struct thread));
    thr->argument = argument;
    thr->initial_address = entry_point;
    thr->state = THREAD_STATE_READY;
    thr->time_used = 0;
    thr->name = strdup(name);
    thr->priority = priority;
    thr->needs_termination = false;
    thr->time_used = false;
    thr->waiting_on_semaphore = NULL;
    thr->schedule_policy = policy;
    thr->timeslice_expiry = GetSystemTimer() + TIMESLICE_LENGTH_MS;
    thr->vas = vas;
    thr->thread_id = GetNextThreadId();
    CreateKernelStacks(thr, kernel_stack_kb == 0 ? DEFAULT_KERNEL_STACK_KB : 0);

    if (prcss != NULL) {
        AddThreadToProcess(prcss, thr);
    } else {
        thr->process = NULL;
    }

    LockScheduler();
    ThreadListInsert(&ready_list, thr);
    UnlockScheduler();

    return thr;
}

struct thread* CreateThread(void(*entry_point)(void*), void* argument, struct vas* vas, const char* name) {
    return CreateThreadEx(
        entry_point, argument, vas, name, GetProcess(), SCHEDULE_POLICY_FIXED, FIXED_PRIORITY_KERNEL_NORMAL, 0
    );
}

struct thread* GetThread(void) {
    return GetCpu()->current_thread;
}

static void UpdateTimesliceExpiry(void) {
    struct thread* thr = GetThread();
    thr->timeslice_expiry = GetSystemTimer() + (thr->priority == 255 ? 0 : (20 + thr->priority / 4) * 1000000ULL);
}

void ThreadExecuteInUsermode(void* arg) {
    struct thread* thr = GetThread();

    int res = CopyProgramLoaderIntoAddressSpace();
    if (res != 0) {
        LogDeveloperWarning("COULDN'T LOAD PROGRAM LOADER!\n");
        TerminateThread(thr);
    }

    size_t user_stack = CreateUserStack(USER_STACK_MAX_SIZE);

    LockScheduler();
    thr->stack_pointer = user_stack;
    UnlockScheduler();

    ArchFlushTlb(GetVas());
    ArchSwitchToUsermode(ARCH_PROG_LOADER_ENTRY, user_stack, arg);
}

struct process* CreateUsermodeProcess(struct process* parent, const char* filename) {
    return CreateProcessWithEntryPoint(GetPid(parent), ThreadExecuteInUsermode, (void*) filename);
}

void ThreadInitialisationHandler(void) {
    /*
     * This normally happends in the schedule code, just after the call to ArchSwitchThread,
     * but we forced ourselves to jump here instead, so we'd better do it now.
     */
    ReleaseSpinlockIrql(&innermost_lock);

    UpdateTimesliceExpiry();

    /*
    * To get here, someone must have called thread_schedule(), and therefore
    * the lock must have been held.
    */
    UnlockScheduler();


    /* Anything else you might want to do should be done here... */

    /* Go to the address the thread actually wanted. */
    GetThread()->initial_address(GetThread()->argument);

    /* The thread has returned, so just terminate it. */
    TerminateThread(GetThread());

    Panic(PANIC_IMPOSSIBLE_RETURN);
}

static int GetMinPriorityValueForPolicy(int policy) {
    if (policy == SCHEDULE_POLICY_USER_HIGHER) return 50;
    if (policy == SCHEDULE_POLICY_USER_NORMAL) return 100;
```

```
276     if (policy == SCHEDULE_POLICY_USER_LOWER) return 150;
277     return 0;
278 }
279
280 static int GetMaxPriorityValueForPolicy(int policy) {
281     if (policy == SCHEDULE_POLICY_FIXED) return 255;
282     return GetMinPriorityValueForPolicy(policy) + 100;
283 }
284
285 static void UpdatePriority(bool yielded) {
286     struct thread* thr = GetThread();
287     int policy = thr->schedule_policy;
288
289     if (policy != SCHEDULE_POLICY_FIXED) {
290         int new_val = thr->priority + (yielded ? -1 : 1);
291         if (new_val >= GetMinPriorityValueForPolicy(policy) && new_val <= GetMaxPriorityValueForPolicy(policy)) {
292             thr->priority = new_val;
293         }
294     }
295 }
296
297 void LockSchedulerX(void) {
298     AcquireSpinlockIrql(&scheduler_lock);
299 }
300
301 void UnlockSchedulerX(void) {
302     ReleaseSpinlockIrql(&scheduler_lock);
303 }
304
305 void AssertSchedulerLockHeld(void) {
306     assert(IsSpinlockHeld(&scheduler_lock));
307 }
308
309 __attribute__((returns_twice)) static void SwitchToNewTask(struct thread* old_thread, struct thread* new_thread) {
310     new_thread->state = THREAD_STATE_RUNNING;
311     ThreadListDeleteTop(&ready_list);
312
313     /*
314      * No IRQs allowed while this happens, as we need to protect the CPU structure.
315      * Only our CPU has access to it (as it is per-CPU), but if we IRQ and then someone
316      * calls GetCpu(), we'll be in a bit of strife.
317      */
318     struct cpu* cpu = GetCpu();
319     AcquireSpinlockIrql(&innermost_lock);
320
321     if (new_thread->vas != old_thread->vas) {
322         SetVas(new_thread->vas);
323     }
324
325     cpu->current_thread = new_thread;
326     cpu->current_vas = new_thread->vas;
327     ArchSwitchThread(old_thread, new_thread);
328
329     /*
330      * This code doesn't get called on the first time a thread gets run!! It jumps straight from
331      * ArchSwitchThread to ThreadInitialisationHandler!
332      */
333     ReleaseSpinlockIrql(&innermost_lock);
334
335     UpdateTimesliceExpiry();
336 }
337
338 static void ScheduleWithLockHeld(void) {
339     EXACT_IRQL(IRQL_SCHEDULER);
340     AssertSchedulerLockHeld();
341
342     struct thread* old_thread = GetThread();
343     struct thread* new_thread = ready_list.head;
344
345     if (old_thread == NULL) {
346         /*
347          * Multitasking not set up yet. Now check if someone has added a task that we can switch to.
348          * (If not, we keep waiting until they have, then we can start multitasking).
349          */
350         if (ready_list.head != NULL) {
351             /*
352              * We need a place where it can write the "old" stack pointer to.
353              */
354             struct thread dummy;
355             SwitchToNewTask(&dummy, new_thread);
356         }
357         return;
358     }
359
360     if (new_thread == old_thread || new_thread == NULL) {
361         /*
362          * Don't switch if there isn't anything else to switch to!
363          */
364         return;
365     }
366
367 #ifndef NDEBUG
368     CheckCanary(old_thread->canary_position);
369 #endif
370
371     bool yielded = old_thread->timeslice_expiry > GetSystemTimer();
372     UpdatePriority(yielded);
373     UpdateThreadTimeUsed();
374
375     /*
376      * Put the old task back on the ready list, but only if it didn't block / get suspended.
377      */
378     if (old_thread->state == THREAD_STATE_RUNNING) {
379         ThreadListInsert(&ready_list, old_thread);
380     }
381
382     SwitchToNewTask(old_thread, new_thread);
383 }
384
385 void Schedule(void) {
386     if (GetIrql() > IRQL_PAGE_FAULT) {
387         PostponeScheduleUntilStandardIrql();
388         return;
389     }
390
391     LockScheduler();
392     ScheduleWithLockHeld();
393     UnlockScheduler();
394
395     /**
396      * Used to allow TerminateThread() to kill a foreign process. This is because we can't just yank
397      * a thread off another list if it's blocked, as we don't know what list it's on. This way, we just
398      * signal that it needs terminating next time we allow it to run.
399      */
400     if (GetThread()->needs_termination) {
401         LogWriteSerial("Terminating a thread that was scheduled to die... stack at 0x%X\n", GetThread()->kernel_stack_top - GetThread()->kernel_stack_size);
402         TerminateThread(GetThread());
403         Panic(PANIC_IMPOSSIBLE_RETURN);
404     }
405 }
```

```

void InitScheduler() {
    ThreadListInit(&ready_list, NEXT_INDEX_READY);
    InitSpinlock(&scheduler_lock, "scheduler", IRQL_SCHEDULER);
    InitSpinlock(&innermost_lock, "inner scheduler", IRQL_HIGH);
}

[[noreturn]] void StartMultitasking(void) {
    InitIdle();
    InitCleaner();

    /*
     * Once this is called, "the game is afoot!" and threads will start running.
     */
    Schedule();
    Panic(PANIC_IMPOSSIBLE_RETURN);
}


/**
 * Sets the priority and/or policy of a thread.
 *
 * @param policy    The scheduling policy to use. Should be one of SCHEDULE_POLICY_FIXED, SCHEDULE_POLICY_USER_LOWER,
 *                      SCHEDULE_POLICY_USER_NORMAL, SCHEDULE_POLICY_USER_HIGHER, or -1. Set to -1 to indicate that the
 *                      policy should not be changed.
 * @param priority  The priority level to give the thread. Should be a value between 0 and 255, where 0 indicates the highest
 *                      possible priority. A value of 255 indicates that the thread should only be run when the system is dile.
 *                      If the thread's policy is SCHEDULE_POLICY_FIXED, then this value will get used directly. For other
 *                      policies, the actual priority given to the thread may differ depending on the rules of the policy.
 *                      Set to -1 to indicate that the policy should not be changed.
 *
 * @return Returns 0 on success, EINVAL if invalid arguments are given. If EINVAL is returned, no change will be made to the thread's
 *         policy or priority.
 *
 * @user This function may be used as a system call, as long as 'thr' points to a valid thread structure (which it should do,
 *       as the user will probably supply thread number, which the kernel then converts to address - or kernel may just make it
 *       a 'current thread' syscall, in which case GetThread() will be valid.
 *
 * @maxirql IRQL_HIGH
 */
int SetThreadPriority(struct thread* thr, int policy, int priority) {
    if (priority < -1 || priority > 255) {
        return EINVAL;
    }
    if (policy != -1 && policy != SCHEDULE_POLICY_FIXED && policy != SCHEDULE_POLICY_USER_LOWER && policy != SCHEDULE_POLICY_USER_NORMAL && policy != SCHEDULE_PO
        return EINVAL;
    }

    if (priority < GetMinPriorityValueForPolicy(policy)) {
        priority = GetMinPriorityValueForPolicy(policy);
    }
    if (priority > GetMaxPriorityValueForPolicy(policy)) {
        priority = GetMaxPriorityValueForPolicy(policy);
    }

    if (policy != -1) {
        thr->schedule_policy = policy;
    }
    if (priority != -1) {
        thr->priority = priority;
    }

    return 0;
}

void AssignThreadToCpu(void) {
    // NO-OP UNTIL SMP IMPLEMENTED
}

void UnassignThreadToCpu(void) {
    // NO-OP UNTIL SMP IMPLEMENTED
}
```

File: ./thread/timer.c

```

#include <timer.h>
#include <spinlock.h>
#include <assert.h>
#include <cpu.h>
#include <panic.h>
#include <irql.h>
#include <log.h>
#include <arch.h>
#include <thread.h>
#include <priorityqueue.h>
#include <threadlist.h>

static struct spinlock timer_lock;
static struct priority_queue* sleep_queue;
static struct thread_list sleep_overflow_list;

#define SLEEP_QUEUE_LENGTH 32

static uint64_t system_time = 0;
static int sleep_wakeups_posted = 0;

void ReceivedTimer(uint64_t nanos) {
    EXACT_IRQL(IRQL_TIMER);

    if (ArchGetCurrentCpuIndex() == 0) {
        /*
         * As we're in the timer handler, we know we already have IRQL_TIMER, and so we don't
         * need to incur the additional overhead of raising and lowering.
         */
        AcquireSpinlockDirect(&timer_lock);
        system_time += nanos;
        ReleaseSpinlockDirect(&timer_lock);
    }

    /*
     * Preempt the current thread if it has used up its timeslice.
     */
    struct thread* thr = GetThread();
    if (thr != NULL && thr->timeslice_expiry != 0 && thr->timeslice_expiry <= system_time) {
        PostponeScheduleUntilStandardIrql();
    }

    if (GetNumberInDeferQueue() < 8) {
        DeferUntilIrql(IRQL_STANDARD, HandleSleepWakeups, (void*) &system_time);
    }
}

uint64_t GetSystemTimer(void) {
    MAX_IRQL(IRQL_TIMER);

```

```
52      AcquireSpinlockIrql(&timer_lock);
53      uint64_t value = system_time;
54      ReleaseSpinlockIrql(&timer_lock);
55
56      return value;
57  }
58
59  void InitTimer(void) {
60      InitSpinlock(&timer_lock, "timer", IRQL_TIMER);
61      ThreadListInit(&sleep_overflow_list, NEXT_INDEX_SLEEP);
62      sleep_queue = PriorityQueueCreate(SLEEP_QUEUE_LENGTH, false, sizeof(struct thread*));
63  }
64
65  void QueueForSleep(struct thread* thr) {
66      AssertSchedulerLockHeld();
67
68      thr->timed_out = false;
69
70      if (PriorityQueueGetUsedSize(sleep_queue) == SLEEP_QUEUE_LENGTH) {
71          ThreadListInsert(&sleep_overflow_list, thr);
72      } else {
73          PriorityQueueInsert(sleep_queue, (void*) &thr, thr->sleep_expiry);
74      }
75  }
76
77  bool TryDequeueForSleep(struct thread* thr) {
78      AssertSchedulerLockHeld();
79
80      while (PriorityQueueGetUsedSize(sleep_queue) > 0) {
81          struct priority_queue_result res = PriorityQueuePeek(sleep_queue);
82          struct thread* top_thread = *((struct thread**) res.data);
83          PriorityQueuePop(sleep_queue);
84          if (top_thread == thr) {
85              return true;
86          }
87          ThreadListInsert(&sleep_overflow_list, top_thread);
88      }
89
90      struct thread* iter = sleep_overflow_list.head;
91      while (iter) {
92          if (iter == thr) {
93              ThreadListDelete(&sleep_overflow_list, iter);
94              return true;
95
96          } else {
97              iter = iter->next[NEXT_INDEX_SLEEP];
98          }
99      }
100
101     return false;
102 }
103
104 void HandleSleepWakeups(void* sys_time_ptr) {
105     MAX_IRQL(IRQL_PAGE_FAULT);
106
107     if (GetThread() == NULL) {
108         return;
109     }
110
111     LockScheduler();
112     if (sleep_wakeups_posted > 0) {
113         --sleep_wakeups_posted;
114     }
115
116     uint64_t system_time = *((uint64_t*) sys_time_ptr);
117
118     /*
119     * Wake up any sleeping tasks that need it.
120     */
121     while (PriorityQueueGetUsedSize(sleep_queue) > 0) {
122         struct priority_queue_result res = PriorityQueuePeek(sleep_queue);
123
124         /*
125         * Check if it needs waking.
126         */
127         if (res.priority <= system_time) {
128             struct thread* thr = *((struct thread**) res.data);
129             thr->timed_out = true;
130             PriorityQueuePop(sleep_queue);
131             UnblockThread(thr);
132
133         } else {
134             /*
135             * If this one doesn't need waking, none of the others will either.
136             */
137             break;
138         }
139     }
140
141     /*
142     * Check for any tasks that are asleep but on the overflow list. This is slow, but will
143     * only happen if we have more than 32 sleeping tasks, so it should normally not take any time.
144     */
145     struct thread* iter = sleep_overflow_list.head;
146     while (iter) {
147         if (iter->sleep_expiry <= system_time) {
148             ThreadListDelete(&sleep_overflow_list, iter);
149             iter->timed_out = true;
150             UnblockThread(iter);
151             iter = sleep_overflow_list.head;            // restart, as list changed
152
153         } else {
154             iter = iter->next[NEXT_INDEX_SLEEP];
155         }
156     }
157
158     UnlockScheduler();
159 }
160
161 /*
162  * Needs to be allowed at IRQL_PAGE_FAULT so the IDE driver can use it. Cannot be any higher, as otherwise
163  * the thread might not actually sleep (if IRQL_SCHEDULER or above, we won't actually do the 'blocked task switch'
164  * until it is released).
165  */
166 void SleepUntil(uint64_t system_time_ns) {
167     MAX_IRQL(IRQL_PAGE_FAULT);
168
169     if (system_time_ns < GetSystemTimer()) {
170         return;
171     }
172
173     LockScheduler();
174     GetThread()->sleep_expiry = system_time_ns;
175     QueueForSleep(GetThread());
176     BlockThread(THREAD_STATE_SLEEPING);
177     UnlockScheduler();
178 }
179
180 void SleepNano(uint64_t delta_ns) {
181     MAX_IRQL(IRQL_PAGE_FAULT);
```

```
182     SleepUntil(GetSystemTimer() + delta_ns);
183 }
184
185 void SleepMilli(uint32_t delta_ms) {
186     MAX_IRQL(IRQL_PAGE_FAULT);
187     SleepNano(((uint64_t) delta_ms) * 1000000ULL);
188 }
```

File: ./thread/process.c

```
1
2   /*
3    * thread/process.c - Processes
4    */
5
6   #include <arch.h>
7   #include <irql.h>
8   #include <thread.h>
9   #include <assert.h>
10  #include <virtual.h>
11  #include <sys/wait.h>
12  #include <sys/types.h>
13  #include <avl.h>
14  #include <panic.h>
15  #include <semaphore.h>
16  #include <filedes.h>
17  #include <spinlock.h>
18  #include <heap.h>
19  #include <process.h>
20  #include <log.h>
21  #include <linkedlist.h>
22
23  struct process {
24      pid_t pid;
25      struct vas* vas;
26      pid_t parent;
27      struct avl_tree* children;
28      struct avl_tree* threads;
29      struct semaphore* lock;
30      struct semaphore* killed_children_semaphore;
31      struct filedes_table* filedes_table;
32      int retv;
33      bool terminated;
34  };
35
36  struct process_table_node {
37      pid_t pid;
38      struct process* process;
39  };
40
41  static struct spinlock pid_lock;
42  static struct avl_tree* process_table;
43  static struct semaphore* process_table_mutex;
44
45  static int ProcessTableComparator(void* a_, void* b_) {
46      struct process_table_node* a = a_;
47      struct process_table_node* b = b_;
48      return COMPARE_SIGN(a->pid, b->pid);
49  }
50
51  static pid_t AllocateNextPid(void) {
52      static pid_t next_pid = 1;
53
54      AcquireSpinlockIrql(&pid_lock);
55      pid_t pid = next_pid++;
56      ReleaseSpinlockIrql(&pid_lock);
57
58      return pid;
59  }
60
61  static int InsertIntoProcessTable(struct process* prcss) {
62      pid_t pid = AllocateNextPid();
63
64      AcquireMutex(process_table_mutex, -1);
65
66      struct process_table_node* node = AllocHeap(sizeof(struct process_table_node));
67      node->pid = pid;
68      node->process = prcss;
69      AvlTreeInsert(process_table, (void*) node);
70
71      ReleaseMutex(process_table_mutex);
72
73      return pid;
74  }
75
76  static void RemoveFromProcessTable(pid_t pid) {
77      AcquireMutex(process_table_mutex, -1);
78
79      struct process_table_node dummy;
80      dummy.pid = pid;
81      struct process_table_node* actual = AvlTreeGet(process_table, (void*) &dummy);
82      AvlTreeDelete(process_table, (void*) actual);
83      FreeHeap(actual);       // this was allocated on 'InsertIntoProcessTable'
84
85      ReleaseMutex(process_table_mutex);
86  }
87
88  void LockProcess(struct process* prcss) {
89      AcquireMutex(prcss->lock, -1);
90  }
91
92  void UnlockProcess(struct process* prcss) {
93      ReleaseMutex(prcss->lock);
94  }
95
96  void InitProcess(void) {
97      InitSpinlock(&pid_lock, "pid", IRQL_SCHEDULER);
98      process_table_mutex = CreateMutex("prcss table");
99      process_table = AvlTreeCreate();
100     AvlTreeSetComparator(process_table, ProcessTableComparator);
101 }
102
103 struct process* CreateProcess(pid_t parent_pid) {
104     EXACT_IRQL(IRQL_STANDARD);
105
106     struct process* prcss = AllocHeap(sizeof(struct process));
107
108     prcss->lock = CreateMutex("prcss");
109     prcss->vas = CreateVas();
110     prcss->parent = parent_pid;
111     prcss->children = AvlTreeCreate();
112     prcss->threads = AvlTreeCreate();
113     prcss->killed_children_semaphore = CreateSemaphore("killed children", SEM_BIG_NUMBER, SEM_BIG_NUMBER);
114     prcss->retv = 0;
115     prcss->terminated = false;
116     prcss->pid = InsertIntoProcessTable(prcss);
117     prcss->filedes_table = CreateFileDescriptorTable();
```

```
118
119     if (parent_pid != 0) {
120         struct process* parent = GetProcessFromPid(parent_pid);
121         LockProcess(parent);
122         AvlTreeInsert(parent->children, (void*) prcss);
123         UnlockProcess(parent);
124     }
125
126     return prcss;
127 }
128
129 void AddThreadToProcess(struct process* prcss, struct thread* thr) {
130     LockProcess(prcss);
131     AvlTreeInsert(prcss->threads, (void*) thr);
132     thr->process = prcss;
133     UnlockProcess(prcss);
134 }
135
136 struct process* ForkProcess(void) {
137     MAX_IRQL(IRQL_PAGE_FAULT);
138
139     LockProcess(GetProcess());
140
141     struct process* new_process = CreateProcess(GetProcess()->pid);
142     DestroyVas(new_process->vas);
143
144     // TODO: there are probably more things to copy over in the future (e.g. list of open file descriptors, etc.)
145     //       the open files, etc.
146
147
148     // TODO: copy file descriptor table
149
150     new_process->vas = CopyVas();
151     //TODO: need to grab the first thread (I don't think we've ordered threads by thread id yet in the AVL)
152     //      so will need to fix that first.
153     //CopyThreadToNewProcess(new_process, )
154     UnlockProcess(GetProcess());
155
156     return new_process;
157 }
158
159 /**
160  * Directly reaps a process.
161  */
162 static void ReapProcess(struct process* prcss) {
163     // TOOD: there's more cleanup to be done here... ?
164     assert(prcss->vas != GetVas());
165
166     int res = DestroySemaphore(prcss->killed_children_semaphore, SEM_REQUIRE_FULL);
167     (void) res;
168     assert(res == 0);
169     DestroyVas(prcss->vas);
170
171     RemoveFromProcessTable(prcss->pid);
172     if (prcss->parent != 0) {
173         struct process* parent = GetProcessFromPid(prcss->parent);
174         AvlTreeDelete(parent->children, prcss);
175     }
176     FreeHeap(prcss);
177 }
178
179 /**
180  * Recursively goes through the children of a process, reaping the first child that is able to be reaped.
181  * Depending on the value of `target`, it will either reap the first potential candidate, or a particular candidate.
182  *
183  * @param parent The process whose children we are looking through
184  * @param node The current subtree of the parent process' children tree
185  * @param target Either the process ID of the child to reap, or -1 to reap the first valid candidate.
186  * @param status If a child is reaped, its return value will be written here.
187  * @return The process ID of the reaped child, or 0 if no children are reaped.
188  */
189 static pid_t RecursivelyTryReap(struct process* parent, struct avl_node* node, pid_t target, int* status) {
190     if (node == NULL) {
191         return 0;
192     }
193
194     struct process* child = (struct process*) AvlTreeGetData(node);
195
196     LockProcess(child);
197
198     if (child->terminated && (child->pid == target || target == (pid_t) -1)) {
199         *status = child->retv;
200         pid_t pid = child->pid;
201         UnlockProcess(child);       // needed in case someone is waiting on us, before our death
202         ReapProcess(child);
203         return pid;
204     }
205
206     UnlockProcess(child);
207
208     pid_t left_retv = RecursivelyTryReap(parent, AvlTreeGetLeft(node), target, status);
209     if (left_retv != 0) {
210         return left_retv;
211     }
212     return RecursivelyTryReap(parent, AvlTreeGetRight(node), target, status);
213 }
214
215
216 /**
217  * Changes the parent of a parentless process. Used to ensure the initial process can always reap orphaned
218  * processes.
219  */
220 static void AdoptOrphan(struct process* adopter, struct process* ophan) {
221     LockProcess(adopter);
222
223     ophan->parent = adopter->pid;
224     AvlTreeInsert(adopter->children, (void*) ophan);
225     ReleaseSemaphore(adopter->killed_children_semaphore);
226
227     UnlockProcess(adopter);
228 }
229
230 /**
231  * Recursively converts all child processes in a process' thread tree into zombie processes.
232  *
233  * @param node The subtree to start from. NULL is acceptable, and is the recursion base case.
234  */
235 static void RecursivelyMakeChildrenOrphans(struct avl_node* node) {
236     if (node == NULL) {
237         return;
238     }
239
240     RecursivelyMakeChildrenOrphans(AvlTreeGetLeft(node));
241     RecursivelyMakeChildrenOrphans(AvlTreeGetRight(node));
242     AdoptOrphan(GetProcessFromPid(1), AvlTreeGetData(node));
243 }
244
245 /**
246  * Recursively terminates all threads in a process' thread tree.
247  *
```

```
 * @param node The subtree to start from. NULL is acceptable, and is the recursion base case.
 */
static void RecursivelyKillRemainingThreads(struct avl_node* node) {
    if (node == NULL) {
        return;
    }

    RecursivelyKillRemainingThreads(AvlTreeGetLeft(node));
    RecursivelyKillRemainingThreads(AvlTreeGetRight(node));

    struct thread* victim = AvlTreeGetData(node);
    if (victim->state != THREAD_STATE_TERMINATED && (victim->needs_termination) {
        TerminateThread(victim);
    }
}

/**
 * Does all of the required operations to kill a process. This is run in its own thread, without an owning
 * process, so that a process doesn't try to delete itself (and therefore delete its stack).
 *
 * @param arg The process to kill (needs to be cast to struct process*)
 */
static void KillProcessHelper(void* arg) {
    struct process* prcss = arg;

    assert(GetProcess() == NULL);
    assert(GetVas() != prcss->vas);     // we should be on GetKernelVas()

    RecursivelyKillRemainingThreads(AvlTreeGetRootNode(prcss->threads));
    RecursivelyMakeChildrenOrphans(AvlTreeGetRootNode(prcss->children));

    AvlTreeDestroy(prcss->threads);
    AvlTreeDestroy(prcss->children);

    DestroyVas(prcss->vas);

    prcss->terminated = true;

    if (prcss->parent == 0) {
        ReapProcess(prcss);

    } else {
        struct process* parent = GetProcessFromPid(prcss->parent);
        ReleaseSemaphore(parent->killed_children_semaphore);
    }

    TerminateThread(GetThread());
}

/**
 * Deletes the current process and all its threads. Child processes have their parent switched to pid 1.
 * If the process being deleted has a parent, then it becomes a zombie process until the parent reaps it
 * If the process being deleted has no parent, it will be reaped and deallocated immediately.
 *
 * This function does not return.
 *
 * @param retv The return value the process being deleted will give.
 */
void KillProcess(int retv) {
    MAX_IRQL(IRQL_STANDARD);

    struct process* prcss = GetProcess();
    prcss->retv = retv;

    /**
     * Must run it in a different thread and process (a NULL process is fine), as it is going to kill all
     * threads in the process, and the process itself.
     */
    CreateThreadEx(KillProcessHelper, (void*) prcss, GetKernelVas(), "process killer", NULL,
        SCHEDULE_POLICY_FIXED, FIXED_PRIORITY_KERNEL_HIGH, 0);

    TerminateThread(GetThread());
}

/**
 * Returns a pointer to the process that the thread currently running on this CPU belongs to. If there is no
 * running thread (i.e. multitasking hasn't started yet), or the thread does not belong to a process, NULL
 * is returned.
 *
 * @return The process of the current thread, if it exists, or NULL otherwise.
 */
struct process* GetProcess(void) {
    MAX_IRQL(IRQL_HIGH);
    struct thread* thr = GetThread();
    return thr == NULL ? NULL : thr->process;
}

struct process* CreateProcessWithEntryPoint(pid_t parent, void(*entry_point)(void*), void* args) {
    EXACT_IRQL(IRQL_STANDARD);
    struct process* prcss = CreateProcess(parent);
    struct thread* thr = CreateThread(entry_point, args, prcss->vas, "prcssinit");
    AddThreadToProcess(prcss, thr);
    return prcss;
}

/**
 * Returns the file descriptor table of the given process. Returns NULL if
 * `prcss` is null.
 */
struct filedes_table* GetFileDescriptorTable(struct process* prcss) {
    if (prcss == NULL) {
        return NULL;
    }

    return prcss->filedes_table;
}

/**
 * Given a process id, returns the process object. Returns NULL for an invalid
 * `pid`.
 */
struct process* GetProcessFromPid(pid_t pid) {
    EXACT_IRQL(IRQL_STANDARD);

    AcquireMutex(process_table_mutex, -1);

    struct process_table_node dummy;
    dummy.pid = pid;
    struct process_table_node* node = AvlTreeGet(process_table, (void*) &dummy);

    ReleaseMutex(process_table_mutex);

    return node == NULL ? NULL : node->process;
}

/**
 * Returns the process id of a given process. If `prcss` is null, 0 is returned.
 */
pid_t GetPid(struct process* prcss) {
    MAX_IRQL(IRQL_HIGH);
```

```
    return prcss == NULL ? 0 : prcss->pid;
}

pid_t WaitProcess(pid_t pid, int* status, int flags) {
    EXACT_IRQL(IRQL_STANDARD);

    struct process* prcss = GetProcess();

    pid_t result = 0;
    int failed_reaps = 0;
    while (result == 0) {
        int res = AcquireSemaphore(prcss->killed_children_semaphore, (flags & WNOHANG) ? 0 : -1);
        if (res != 0) {
            break;
        }
        LockProcess(prcss);
        result = RecursivelyTryReap(prcss, AvlTreeGetRootNode(prcss->children), pid, status);
        UnlockProcess(prcss);
        if (result == 0 && pid != (pid_t) -1) {
            failed_reaps++;
        }
    }

    /*
     * Ensure that the next time we call WaitProcess(), we can immediately retry the reaps that
     * we increased the semaphore for, but didn't actually reap on.
     */
    while (failed_reaps--) {
        ReleaseSemaphore(prcss->killed_children_semaphore);
    }

    return result;
}
```

File: ./thread/progload.c

```

#include <thread.h>
#include <progload.h>
#include <assert.h>
#include <string.h>
#include <irql.h>
#include <fcntl.h>
#include <log.h>
#include <errno.h>
#include <virtual.h>
#include <panic.h>
#include <common.h>
#include <semaphore.h>
#include <sys/types.h>
#include <arch.h>
#include <vfs.h>

static size_t program_loader_addr;
static off_t program_loader_size;

void InitProgramLoader(void) {
    struct open_file* file;
    int res = OpenFile("sys:/progload.exe", O_RDONLY, 0, &file);
    if (res != 0) {
        PanicEx(PANIC_PROGRAM_LOADER, "program loader couldn't be loaded");
    }

    res = GetFileSize(file, &program_loader_size);
    if (res != 0) {
        PanicEx(PANIC_PROGRAM_LOADER, "program loader size couldn't be found");
    }

    program_loader_addr = MapVirt(0, 0, program_loader_size, VM_READ | VM_FILE, file, 0);
}

int CopyProgramLoaderIntoAddressSpace(void) {
    size_t mem = MapVirt(0, ARCH_PROG_LOADER_BASE, program_loader_size, VM_READ | VM_EXEC | VM_WRITE | VM_USER | VM_LOCAL, NULL, 0);
    if (mem != ARCH_PROG_LOADER_BASE) {
        return ENOMEM;
    }

    memcpy((void*) ARCH_PROG_LOADER_BASE, (void*) program_loader_addr, program_loader_size);
    return 0;
}
```

File: ./thread/cleaner.c

```
1
2  /***
3   * thread/cleaner.c - Thread Termination Cleanup
4   *
5   * Threads are unable to delete their own stacks. Therefore, we have a seperate thread which
6   * deletes the stacks (and any other leftover data) of threads that are marked as terminated.
7   */
8
9  #include <thread.h>
10 #include <virtual.h>
11 #include <threadlist.h>
12 #include <irql.h>
13 #include <heap.h>
14 #include <semaphore.h>
15 #include <log.h>
16 #include <panic.h>
17 #include <physical.h>
18
19 static struct thread_list terminated_list;
20 static struct semaphore* cleaner_semaphore;
21
22 static void DestroyThread(struct thread* thr) {
23     UnmapVirt(thr->kernel_stack_top - thr->kernel_stack_size, thr->kernel_stack_size);
24     FreeHeap(thr->name);
25     FreeHeap(thr);
26 }
27
28 static void CleanerThread(void*) {
29     while (true) {
30         AcquireSemaphore(cleaner_semaphore, -1);
31
32         LockScheduler();
33         struct thread* thr = terminated_list.head;
34         assert(thr != NULL);
35         ThreadListDeleteTop(&terminated_list);
36         UnlockScheduler();
37
38         DestroyThread(thr);
39     }
40 }
41
42 static void NotifyCleaner(void*) {
43     ReleaseSemaphore(cleaner_semaphore);
44 }
45
46 void TerminateThreadLockHeld(struct thread* thr) {
47     assert(thr != NULL);
48     EXACT_IRQL(IRQL_SCHEDULER);
49
50     if (thr == GetThread()) {
51         ThreadListInsert(&terminated_list, thr);
52
53         BlockThread(THREAD_STATE_TERMINATED);
54         DeferUntilIrql(IRQL_STANDARD, NotifyCleaner, NULL);
55
56     } else {
57         /*
58          * We can't terminate it directly, as it may be on any queue somewhere else. Instead, we
59          * will terminate it next time it is up for scheduling.
60          */
61         thr->needs_termination = true;
62     }
63 }
64
65 /**
66  * Terminates a thread. This function must not be called until after `InitCleaner` has been called.
67  * The scheduler lock should not be already held.
68  *
69  * @param thr The thread to terminated.
70  * @return This function does not return if thr == GetThread(), and returns void otherwise.
71  *
72  * @note MAX_IRQL(IRQL_SCHEDULER)
73  */
74 void TerminateThread(struct thread* thr) {
75     MAX_IRQL(IRQL_SCHEDULER);
76
77     LockScheduler();
78     TerminateThreadLockHeld(thr);
79     UnlockScheduler();
80
81     if (thr == GetThread()) {
82         Panic(PANIC_IMPOSSIBLE_RETURN);
83     }
84 }
85
86 /**
87  * Creates the cleaner thread. This must be called before any calls to `TerminateThread` are made.
88  */
89 void InitCleaner(void) {
90     ThreadListInit(&terminated_list, NEXT_INDEX_TERMINATED);
91     cleaner_semaphore = CreateSemaphore("cleaner", SEM_BIG_NUMBER, SEM_BIG_NUMBER);
92     CreateThread(CleanerThread, NULL, GetVas(), "cleaner");
93 }
```

File: ./thread/idle.c

```
1
2  /***
3   * thread/idle.c - System Idle Task
4   *
5   * A thread that is run if not other thread is available to run. The idle thread must therefore
6   * never block.
7   */
8
9  #include <arch.h>
10 #include <thread.h>
11 #include <virtual.h>
12 #include <irql.h>
13
14 static void IdleThread(void*) {
15     while (1) {
16         ArchStallProcessor();
17     }
18 }
19
20 void InitIdle(void) {
21     CreateThreadEx(IdleThread, NULL, GetVas(), "idle thread", NULL, SCHEDULE_POLICY_FIXED, FIXED_PRIORITY_IDLE, 0);
22 }
```

File: ./mem/swapfile.c

```c
#include <virtual.h>
#include <vfs.h>
#include <fcntl.h>
#include <log.h>
#include <errno.h>
#include <panic.h>
#include <transfer.h>
#include <irql.h>
#include <virtual.h>
#include <physical.h>
#include <spinlock.h>
#include <arch.h>

static struct open_file* swapfile = NULL;
static struct spinlock swapfile_lock;
static uint8_t* swapfile_bitmap;
static int number_on_swapfile = 0;
static size_t num_swapfile_bitmap_entries = 0;

static int GetPagesRequiredForAllocationBitmap(void) {
    uint64_t bits_per_page = ARCH_PAGE_SIZE * 8;
    uint64_t accessable_per_page = ARCH_PAGE_SIZE * bits_per_page;
    size_t max_swapfile_size = (GetTotalPhysKilobytes() * 1024) * 4 + (32 * 1024 * 1024);
    return (max_swapfile_size + accessable_per_page - 1) / accessable_per_page;
}

static void SetupSwapfileBitmap() {
    int num_pages_in_bitmap = GetPagesRequiredForAllocationBitmap();
    num_swapfile_bitmap_entries = 8 * num_pages_in_bitmap * ARCH_PAGE_SIZE;
    swapfile_bitmap = (uint8_t*) MapVirt(0, 0, num_pages_in_bitmap * ARCH_PAGE_SIZE, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
}

void InitSwapfile(void) {
    int res = OpenFile("swap:/", O_RDWR, 0, &swapfile);
    if (res != 0) {
        Panic(PANIC_NO_FILESYSTEM);
    }

    InitSpinlock(&swapfile_lock, "swapfile", IRQL_SCHEDULER);
    SetupSwapfileBitmap();
}

struct open_file* GetSwapfile(void) {
    return swapfile;
}

static bool GetBitmapEntry(size_t index) {
    return swapfile_bitmap[index / 8] & (1 << (index % 8));
}

static void SetBitmapEntry(size_t index, bool value) {
    if (value) {
        swapfile_bitmap[index / 8] |= 1 << (index % 8);
    } else {
        swapfile_bitmap[index / 8] &= ~(1 << (index % 8));
    }
}

uint64_t AllocateSwapfileIndex(void) {
    AcquireSpinlockIrql(&swapfile_lock);
    ++number_on_swapfile;

    for (size_t i = 0; i < num_swapfile_bitmap_entries; ++i) {
        if (!GetBitmapEntry(i)) {
            SetBitmapEntry(i, true);
            ReleaseSpinlockIrql(&swapfile_lock);
            return i;
        }
    }

    Panic(PANIC_OUT_OF_SWAPFILE);
}
void DeallocateSwapfileIndex(uint64_t index) {
    AcquireSpinlockIrql(&swapfile_lock);
    --number_on_swapfile;
    SetBitmapEntry(index, false);
    ReleaseSpinlockIrql(&swapfile_lock);
}

int GetNumberOfPagesOnSwapfile(void) {
    AcquireSpinlockIrql(&swapfile_lock);
    int val = number_on_swapfile;
    ReleaseSpinlockIrql(&swapfile_lock);
    return val;
}
```

File: ./mem/heap.c

```c
#include <common.h>
#include <assert.h>
#include <stdbool.h>
#include <panic.h>
#include <virtual.h>
#include <arch.h>
#include <string.h>
#include <spinlock.h>
#include <heap.h>
#include <log.h>
#include <semaphore.h>
#include <voidptr.h>
#include <irql.h>
#include <thread.h>

/*
 * TODO: once semaphores are allowed / virtual memory is initialised, we need
 * to swap from using spinlocks to using a mutex... the reason being that we
 * can't actually manipulate the pageable heap while a spinlock is held! (as
 * the block might be paged out!!)
 *
 */

#define BOOTSTRAP_AREA_SIZE (1024 * 16)
#define MAX_EMERGENCY_BLOCKS 16

/*
 * For requests larger than this, we'll issue a warning to say that MapVirt is a much better choice.
 */
#define WARNING_LARGE_REQUEST_SIZE (1024 * 3 + 512)

struct emergency_block {
    uint8_t* address;
    size_t size;
    bool valid;
};

```

```
38 /**
39  * Used as a system block that can be used even before the virtual memory manager is setup.
40  */
41 static uint8_t bootstrap_memory_area[BOOTSTRAP_AREA_SIZE] __attribute__ ((aligned(ARCH_PAGE_SIZE)));
42
43 /*
44  * Used to give us memory when we are not allowed to fault (i.e. can't allocate virtual memory).
45  */
46 static struct emergency_block emergency_blocks[MAX_EMERGENCY_BLOCKS] = {
47     {.address = bootstrap_memory_area, .size = BOOTSTRAP_AREA_SIZE, .valid = true}
48 };
49
50 static struct semaphore* heap_lock;
51 static struct spinlock heap_spinlock;
52
53 bool IsHeapReinitialised(void) {
54     return heap_lock != NULL;
55 }
56
57 static struct spinlock heap_locker_lock;
58 static struct thread* lock_entry_threads[2];
59
60 static bool LockHeap(bool paging) {
61     bool acquired = false;
62     struct thread* thr = GetThread();
63
64     AcquireSpinlockIrql(&heap_locker_lock);
65     if (lock_entry_threads[paging ? 1 : 0] != thr || lock_entry_threads[paging ? 1 : 0] == NULL) {
66         ReleaseSpinlockIrql(&heap_locker_lock);
67
68         // TODO: we have atomicity issues between the release of the previous lock
69         // and the acquisition of the next one... may need a retry loop
70
71         /*
72         e.g.
73         retry:
74         Acquire(locker_lock);
75         if (big long condition) {
76             if (paging) {
77                 TryAcquire...
78             } else {
79                 TryAcquire...
80             }
81
82             if (acquired) {
83                 if (!(the same big long condition)) {
84                     release
85                     goto retry;
86                 }
87             } else {
88                 goto retry:
89             }
90
91         }
92         */
93
94         if (paging) {
95             AcquireMutex(heap_lock, -1);
96         } else {
97             AcquireSpinlockIrql(&heap_spinlock);
98         }
99
100        // e.g. check that
101        // (lock_entry_threads[paging ? 1 : 0] != thr || lock_entry_threads[paging ? 1 : 0] == NULL)
102        // still holds,
103
104        acquired = true;
105        lock_entry_threads[paging ? 1 : 0] = thr;
106    }
107
108    return acquired;
109}
110
111static void UnlockHeap(bool paging) {
112    lock_entry_threads[paging ? 1 : 0] = NULL;
113
114    if (paging) {
115        ReleaseMutex(heap_lock);
116
117    } else {
118        ReleaseSpinlockIrql(&heap_spinlock);
119    }
120}
121
122static void* AllocateFromEmergencyBlocks(size_t size) {
123    int smallest_block = -1;
124
125    for (int i = 0; i < MAX_EMERGENCY_BLOCKS; ++i) {
126        if (emergency_blocks[i].valid && emergency_blocks[i].size >= size) {
127            if (smallest_block == -1 || emergency_blocks[i].size < emergency_blocks[smallest_block].size) {
128                smallest_block = i;
129            }
130        }
131    }
132
133    if (smallest_block == -1) {
134        Panic(PANIC_CANNOT_MALLOC_WITHOUT_FAULTING);
135    }
136
137    void* address = emergency_blocks[smallest_block].address;
138
139    emergency_blocks[smallest_block].address += size;
140    emergency_blocks[smallest_block].size -= size;
141
142    if (emergency_blocks[smallest_block].size < ARCH_PAGE_SIZE) {
143        emergency_blocks[smallest_block].valid = false;
144    }
145
146    return address;
147}
148
149/*
150 * This function needs to be called with the heap lock held.
151 */
152static void AddBlockToBackupHeap(size_t size) {
153    UnlockHeap(false);
154    void* address = (void*) MapVirt(0, 0, size, VM_READ | VM_WRITE | VM_LOCK, NULL, 0);
155    LockHeap(false);
156
157    int index_of_smallest_block = 0;
158    for (int i = 0; i < MAX_EMERGENCY_BLOCKS; ++i) {
159        if (emergency_blocks[i].valid) {
160            if (emergency_blocks[i].size < emergency_blocks[index_of_smallest_block].size) {
161                index_of_smallest_block = i;
162            }
163        } else {
164            emergency_blocks[i].valid = true;
165            emergency_blocks[i].size = size;
166            emergency_blocks[i].address = address;
167            return;
```

```
        }
    }

    LogDeveloperWarning("losing 0x%X bytes due to strangeness with backup heap.\n", emergency_blocks[index_of_smallest_block].size);
    LogDeveloperWarning("TODO: could probably just add this as a regular block to the regular heap.\n");

    emergency_blocks[index_of_smallest_block].size = size;
    emergency_blocks[index_of_smallest_block].address = address;
}

static void RestoreEmergencyPages(void* context) {
    (void) context;

    EXACT_IRQL(IRQL_STANDARD);

    /*
     * TODO: mabye make this greedier (i.e. grab larger blocks), but also have a way for the PMM to ask for larger
     * blocks back if needed. would still need to retain enough stashed away for PMM/VMM to use the heap, and would
     * need to unlock the pages, and ensure that allocations from emergency blocks are done via smallest-fit (so large
     * blocks aren't wasted).
     */

    size_t total_size = 0;
    size_t largest_block = 0;

    LockHeap(false);
    for (int i = 0; i < MAX_EMERGENCY_BLOCKS; ++i) {
        if (emergency_blocks[i].valid) {
            size_t size = emergency_blocks[i].size;
            total_size += size;
            if (size > largest_block) {
                largest_block = size;
            }
        }
    }

    while (largest_block < BOOTSTRAP_AREA_SIZE / 2 || total_size < BOOTSTRAP_AREA_SIZE) {
        AddBlockToBackupHeap(BOOTSTRAP_AREA_SIZE);
        total_size += BOOTSTRAP_AREA_SIZE;
        largest_block = BOOTSTRAP_AREA_SIZE;
    }

    UnlockHeap(false);
}

/**
 * Represents a section of memory that is either allocated or free. The memory address
 * it represents is itself, excluding the metadata at the start or end.
 *
 * See the report for further details.
 */
struct block {
    /*
     * The entire size of the block. The low 2 bits do not form part of the size,
     * the low bit is set for allocated blocks, and the second lowest bit is indicates
     * if it is on the swappable heap or not.
     */
    size_t size;

    /*
     * Only here on free blocks. Allocated blocks use this as the start of allocated
     * memory.
     */
    struct block* next;
    struct block* prev;

    /*
     * At position size - sizeof(size_t), there is the trailing size tag.
     * There are no flags in the low bit of this value, unlike the heading size tag.
     */
};

#ifndef NDEBUG
static int outstanding_allocations = 0;

int DbgGetOutstandingHeapAllocations(void) {
    return outstanding_allocations;
}
#endif

/**
 * Must be a power of 2.
 */
#define ALIGNMENT 8

/**
 * The amount of metadata at the start and end of allocated blocks. The next and free
 * pointers in free blocks do not count.
 */
#define METADATA_LEADING_AMOUNT (sizeof(size_t))
#define METADATA_TRIALING_AMOUNT (sizeof(size_t))
#define METADATA_TOTAL_AMOUNT (METADATA_LEADING_AMOUNT + METADATA_TRIALING_AMOUNT)

/**
 * (THIS COMMENT IS ABOUT x86-64 - halve the byte values for x86)
* Having blocks of size 8 is wasteful, as they need to be at least 32 bytes long total to fit
* the metadata when free, but only need 16 bytes metadata when allocated. Therefore, we have
* 8 spare bytes that are wasted.
*/
#define MINIMUM_REQUEST_SIZE_INTERNAL (2 * sizeof(size_t))

/**
 * The last size should be larger than the max allocation size, as otherwise we could allocate larger
 * than we have in the final list.
 */
#define TOTAL_NUM_FREE_LISTS 35

/**
 * An array which holds the minimum allocation sizes that each free list can hold.
 */
static size_t free_list_block_sizes[TOTAL_NUM_FREE_LISTS] = {
    8,          16,         24,         32,
    40,         48,         56,         64,
    72,         80,         88,         96,
    104,        112,        120,        128,
    160,        192,        224,        256,
    320,        384,        448,        512,
    768,        1024,       1536,       2048,       // 28 [27]
    1024 * 4,   1024 * 8,   1024 * 16,  1024 * 32,  // 32 [31]
    1024 * 64,  1024 * 128, 1024 * 256,
};

/**
 * Used to work out which free list a block should be in, when we are *reading* a block.
 * This rounds the size *up*, meaning it cannot be used to insert a block into a list.
 * NOT USED TO INSERT BLOCKS!!
 */
static int GetSmallestListIndexThatFits(size_t size_without_metadata) {
    int i = 0;
    while (true) {
```

```
        if (size_without_metadata <= free_list_block_sizes[i]) {
            return i;
        }
        ++i;
    }
}

/**
 * Calculates which free list a block should be inserted in. This one rounds *down*, and
 * so it should not normally be used to look up where a block should be.
 */
static int GetInsertionIndex(size_t size_without_metadata) {
    /*
     * This will only fail if we somehow get a block that is smaller than the minimum
     * possible size (i.e., something has gone very wrong.)
     */

    assert(size_without_metadata >= free_list_block_sizes[0]);

    /*
     * We can't round down to the next one when it's the smallest possible size,
     * so handle this case specially.
     */
    if (size_without_metadata == free_list_block_sizes[0]) {
        return 0;
    }

    /*
     * Look until we go past the block size we need, and then return the previous
     * value. This gives us the final block size that doesn't exceed the input value.
     */
    for (int i = 0; i < TOTAL_NUM_FREE_LISTS - 1; ++i) {
        if (size_without_metadata <= free_list_block_sizes[i]) {
            return i - 1;
        }
    }

    Panic(PANIC_HEAP_REQUEST_TOO_LARGE);
}

/**
 * Rounds up a user-supplied allocation size to the alignment. If the value is smaller than
 * the minimum request size that is internally supported, it will round it up to that size.
 */
static size_t RoundUpSize(size_t size) {
    /* Should have already been checked against. */
    assert(size != 0);

    if (size < MINIMUM_REQUEST_SIZE_INTERNAL) {
        size = MINIMUM_REQUEST_SIZE_INTERNAL;
    }

    return (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);
}

/**
 * Marks a block as being free (unallocated).
 */
static void MarkFree(struct block* block) {
    block->size &= ~1;
}

/**
 * Marks a block as being allocated.
 */
static void MarkAllocated(struct block* block) {
    block->size |= 1;
}

/**
 * Returns true if the block is allocated, or false if it is free.
 */
static bool IsAllocated(struct block* block) {
    return block->size & 1;
}

static void MarkSwappability(struct block* block, int can_swap) {
    if (can_swap) {
        block->size |= 2;
    } else {
        block->size &= ~2;
    }
}

static bool IsOnSwappableHeap(struct block* block) {
    return block->size & 2;
}

/**
 * Global arrays always initialise to zero (and therefore, to NULL).
 * Entries in free lists must have a user allocated size GREATER OR EQUAL TO the size in free_list_block_sizes.
 */
static struct block* _head_block[TOTAL_NUM_FREE_LISTS];
static struct block* _head_block_swappable[TOTAL_NUM_FREE_LISTS];

static struct block** GetHeap(bool swappable) {
    return swappable ? _head_block_swappable : _head_block;
}

static struct block** GetHeapForBlock(struct block* block) {
    return GetHeap(IsOnSwappableHeap(block));
}


/**
 * Given a block, returns its total size, including metadata. This takes into account
 * the flags on the size field and removes them from the return value.
 */
static size_t GetBlockSize(struct block* block) {
    size_t size = block->size & ~3;

    /*
     * Ensure the other size tag matches. If it doesn't, there has been memory corruption.
     */
    assert(*(((size_t*) block) + (size / sizeof(size_t)) - 1) == size);
    return size;
}

void DbgPrintListStats(void) {
    LogWriteSerial("\nDbgPrintListStats:\n");
    for (int i = 0; i < TOTAL_NUM_FREE_LISTS; ++i) {
        struct block* unswap = GetHeap(false)[i];
        struct block* swap = GetHeap(true)[i];

        size_t unswap_blocks = 0;
        size_t unswap_size = 0;
        size_t swap_blocks = 0;
        size_t swap_size = 0;

        int timeout = 0;
```

```
        while (unswap) {
            ++unswap_blocks;
            unswap_size += GetBlockSize(unswap);
            unswap = unswap->next;
            if (timeout >= 100000) {
                Panic(PANIC_DOUBLE_FREE_DETECTED);
            }
        }

        timeout = 0;
        while (swap) {
            ++swap_blocks;
            swap_size += GetBlockSize(swap);
            swap = swap->next;
            if (timeout >= 100000) {
                Panic(PANIC_DOUBLE_FREE_DETECTED);
            }
        }

        if ((unswap_blocks | swap_blocks) != 0) {
            LogWriteSerial("    Bucket %d [0x%X]: unswappable %d / 0x%X. swappable %d / 0x%X\n", i, free_list_block_sizes[i], unswap_blocks, unswap_size, swap_bl
        }
    }
    LogWriteSerial("\n\n");
}

/**
 * Sets the *total* size of a given block. This does not do any checking, so the caller must be
 * careful to ensure that calling this doesn't leak memory (by setting it too small), or cause
 * double-allocation of the same area (by setting it too large). Carries the swappability and allocation
 * tags with it from the front to the back tags.
 */
static void SetSizeTags(struct block* block, size_t size) {
    block->size = (block->size & 3) | size;
    *(((size_t*) block) + (size / sizeof(size_t)) - 1) = size;
}

static void* GetSystemMemory(size_t size, int flags) {
    EXACT_IRQL(IRQL_SCHEDULER);

    if (flags & HEAP_NO_FAULT) {
        if (flags & HEAP_ALLOW_PAGING) {
            Panic(PANIC_CONFLICTING_ALLOCATION_REQUIREMENTS);
        }
        if (IsVirtInitialised()) {
            DeferUntilIrql(IRQL_STANDARD, RestoreEmergencyPages, NULL);
        }
        return AllocateFromEmergencyBlocks(size);
    }

    if (flags & HEAP_ALLOW_PAGING) {
        LogWriteSerial("about to allocate %d KBs of pageable memory\n", (size + 4095) / 4096 * 4);
    }
    return (void*) MapVirt(0, 0, size, VM_READ | VM_WRITE | (flags & HEAP_ALLOW_PAGING ? 0 : VM_LOCK), NULL, 0);
}

/**
 * Allocates a new block from the system that is able to hold the amount of data
 * specified. Also allocated enough memory for fenceposts on either side of the data,
 * and sets up these fenceposts correctly.
 */
static struct block* RequestBlock(size_t total_size, int flags) {
    EXACT_IRQL(IRQL_SCHEDULER);

    /*
     * We need to add the extra bytes for fenceposts to be added. We must do this before we
     * round up to the nearest areana size (if we did it after, it wouldn't be aligned anymore).
     */
    total_size += MINIMUM_REQUEST_SIZE_INTERNAL * 2;
    total_size = (total_size + ARCH_PAGE_SIZE - 1) & ~(ARCH_PAGE_SIZE - 1);

    if (!IsVirtInitialised()) {
        flags |= HEAP_NO_FAULT;
    }

    /*
     * Get memory from the system.
     */
    struct block* block = (struct block*) GetSystemMemory(total_size, flags);
    if (block == NULL) {
        Panic(PANIC_OUT_OF_HEAP);
    }

    /*
     * Set the metadata for both the fenceposts and the main data block.
     * Keep in mind that total_size now includes the fencepost metadata (see top of function), so this
     * sometimes needs to be subtracted off.
     */
    struct block* left_fence = block;
    struct block* actual_block = (struct block*) (((size_t*) block) + MINIMUM_REQUEST_SIZE_INTERNAL / sizeof(size_t));
    struct block* right_fence  = (struct block*) (((size_t*) block) + (total_size - MINIMUM_REQUEST_SIZE_INTERNAL) / sizeof(size_t));

    SetSizeTags(left_fence, MINIMUM_REQUEST_SIZE_INTERNAL);
    SetSizeTags(actual_block, total_size - 2 * MINIMUM_REQUEST_SIZE_INTERNAL);
    SetSizeTags(right_fence, MINIMUM_REQUEST_SIZE_INTERNAL);

    actual_block->prev = NULL;
    actual_block->next = NULL;

    MarkAllocated(left_fence);
    MarkAllocated(right_fence);
    MarkFree(actual_block);

    MarkSwappability(left_fence, flags & HEAP_ALLOW_PAGING);
    MarkSwappability(right_fence, flags & HEAP_ALLOW_PAGING);
    MarkSwappability(actual_block, flags & HEAP_ALLOW_PAGING);

    return actual_block;
}

/**
 * Removes a block from a free list. It needs to take in the exact free list's index (as opposed to calculating
 * it itself), as this may be used halfway though allocations or deallocations where the block isn't yet in
 * its correct block.
 */
static void RemoveBlock(int free_list_index, struct block* block) {
    EXACT_IRQL(IRQL_SCHEDULER);

    struct block** head_list = GetHeapForBlock(block);

    /*
     * Perform a standard linked list deletion.
     */
    if (block->prev == NULL && block->next == NULL) {
        assert(head_list[free_list_index] == block);
        head_list[free_list_index] = NULL;

    } else if (block->prev == NULL) {
        head_list[free_list_index] = block->next;
        block->next->prev = NULL;
```

```
558
559     } else if (block->next == NULL) {
560         block->prev->next = NULL;
561
562     } else {
563         block->prev->next = block->next;
564         block->next->prev = block->prev;
565     }
566 }
567
568 /**
569  * Adds a block to its appropriate free list. It also coalesces the block with surrounding free blocks
570  * if possible.
571  */
572 static struct block* AddBlock(struct block* block) {
573     EXACT_IRQL(IRQL_SCHEDULER);
574
575     /*
576      * Although this function is technically recursive (because it needs to shuffle blocks into different
577      * lists by calling itself again), but there are a constant number of free lists, so it is still
578      * coalescing in constant time.
579      */
580     size_t size = GetBlockSize(block);
581     struct block** head_list = GetHeapForBlock(block);
582
583     int free_list_index = GetInsertionIndex(size - METADATA_TOTAL_AMOUNT);
584
585     size_t prev_block_size = *(((size_t*) block) - 1);
586     struct block* prev_block = (struct block*) (((size_t*) block) - prev_block_size / sizeof(size_t));
587     struct block* next_block = (struct block*) (((size_t*) block) + size / sizeof(size_t));
588
589     if (IsAllocated(prev_block) && IsAllocated(next_block)) {
590         /*
591          * Cannot coalesce here, so just add to the free list.
592          */
593         block->prev = NULL;
594         block->next = head_list[free_list_index];
595         if (block->next != NULL) {
596             block->next->prev = block;
597         }
598         head_list[free_list_index] = block;
599         MarkFree(block);
600         return block;
601
602     } else if (IsAllocated(prev_block) && !IsAllocated(next_block)) {
603         /*
604          * Need to coalesce with the one on the right.
605          */
606
607         /*
608          * Swappable and non-swappable blocks should be on entirely seperate heaps, and you can't look into
609          * the other because the fences should prevent anyone looking between them.
610          */
611         bool swappable = IsOnSwappableHeap(block);
612         assert(swappable == IsOnSwappableHeap(next_block));
613
614         RemoveBlock(GetInsertionIndex(GetBlockSize(next_block) - METADATA_TOTAL_AMOUNT), next_block);
615         SetSizeTags(block, size + GetBlockSize(next_block));
616         block->prev = NULL;
617         block->next = NULL;
618         MarkFree(block);
619         MarkSwappability(block, swappable);
620
621         return AddBlock(block);
622
623     } else if (!IsAllocated(prev_block) && IsAllocated(next_block)) {
624         /*
625          * Need to coalesce with the one on the left.
626          */
627
628         /*
629          * Swappable and non-swappable blocks should be on entirely seperate heaps, and you can't look into
630          * the other because the fences should prevent anyone looking between them.
631          */
632         bool swappable = IsOnSwappableHeap(block);
633         assert(swappable == IsOnSwappableHeap(prev_block));
634
635         RemoveBlock(GetInsertionIndex(GetBlockSize(prev_block) - METADATA_TOTAL_AMOUNT), prev_block);
636         SetSizeTags(prev_block, size + GetBlockSize(prev_block));
637         prev_block->prev = NULL;
638         prev_block->next = NULL;
639         MarkFree(prev_block);
640         MarkSwappability(prev_block, swappable);
641
642         return AddBlock(prev_block);
643
644     } else {
645         /*
646          * Coalesce with blocks on both sides.
647          */
648
649         /*
650          * Swappable and non-swappable blocks should be on entirely seperate heaps, and you can't look into
651          * the other because the fences should prevent anyone looking between them.
652          */
653         bool swappable = IsOnSwappableHeap(block);
654         assert(swappable == IsOnSwappableHeap(prev_block));
655         assert(swappable == IsOnSwappableHeap(next_block));
656
657         RemoveBlock(GetInsertionIndex(GetBlockSize(prev_block) - METADATA_TOTAL_AMOUNT), prev_block);
658         RemoveBlock(GetInsertionIndex(GetBlockSize(next_block) - METADATA_TOTAL_AMOUNT), next_block);
659
660         SetSizeTags(prev_block, size + GetBlockSize(prev_block) + GetBlockSize(next_block));
661         prev_block->prev = NULL;
662         prev_block->next = NULL;
663         MarkFree(prev_block);
664         MarkSwappability(prev_block, swappable);
665
666         return AddBlock(prev_block);
667     }
668 }
669
670 /*
671  * Allocates a block. The block to be allocated will be the first block in the given free
672  * list, and that free list must be non-empty, and be able to fit the requested size.
673  */
674 static struct block* AllocateBlock(struct block* block, int free_list_index, size_t user_requested_size) {
675     EXACT_IRQL(IRQL_SCHEDULER);
676     assert(block != NULL);
677
678     size_t total_size = user_requested_size + METADATA_TOTAL_AMOUNT;
679     size_t block_size = GetBlockSize(block);
680
681     assert(block_size >= total_size);
682
683     if (block_size - total_size < MINIMUM_REQUEST_SIZE_INTERNAL + METADATA_TOTAL_AMOUNT) {
684         /*
685          * We can just remove from the list altogether if the sizes match up exactly,
686          * or if there would be so little left over that we can't form a new block.
687          */
```

```
        RemoveBlock(free_list_index, block);
        /*
         * Prevent memory leak (from having a hole in memory), but do it after removing
         * the block, as this may change the list it needs to be in, and RemoveBlock
         * will not like that.
         */
        SetSizeTags(block, block_size);
        MarkAllocated(block);
        return block;

    } else {
        /*
         * We must split the block into two. If no list change is needed, we can leave the 'leftover' parts in the list
         * as is (just fixing up the size tags), and then return the new block.
         */

        RemoveBlock(free_list_index, block);

        size_t leftover = block_size - total_size;
        SetSizeTags(block, leftover);

        struct block* allocated_block = (struct block*) (((size_t*) block) + (leftover / sizeof(size_t)));
        SetSizeTags(allocated_block, total_size);

        /*
         * We are giving it new tags, so must set this correctly.
         */
        MarkSwappability(allocated_block, IsOnSwappableHeap(block));

        /*
         * Must be done before we try to move around the leftovers (or else it will actually
         * coalesce back into one block).
         */
        MarkAllocated(allocated_block);

        /*
        * We need to remove the leftover block from this list, and add it to the correct list.
        */
        MarkFree(block);
        AddBlock(block);
        return allocated_block;
    }
}

/**
 * Allocates a block that can fit the user requested size. It will request new memory from the
 * system if required. If it returns NULL, then there is not enough memory of the system to
 * satisfy the request.
 */
static struct block* FindBlock(size_t user_requested_size, int flags) {
    EXACT_IRQL(IRQL_SCHEDULER);

    struct block** head_list = GetHeap(flags & HEAP_ALLOW_PAGING);
    /*
     * Check the free lists in order, starting from the smallest one that will fit the block.
     * If we find one with a free block, then we allocate the head of that list.
     */
    int min_index = GetSmallestListIndexThatFits(user_requested_size);

    for (int i = min_index; i < TOTAL_NUM_FREE_LISTS; ++i) {
        if (head_list[i] != NULL) {
            return AllocateBlock(head_list[i], i, user_requested_size);
        }
    }

    /*
     * If we can't find a block that will fit, then we must allocate more memory.
     */
    size_t total_size = free_list_block_sizes[min_index + 1] + METADATA_TOTAL_AMOUNT;
    // round up to the size of the next block list size, so we guarantee we end up in the next bucket, to avoid
    // an issue where the user requests, e.g. 2.1KB, and we allocate 3.9KB, which means due to the two lookup types,
    // it gets it in the wrong bucket
    struct block* sys_block = RequestBlock(total_size, flags);

    /*
     * Put the new memory in the free list (which ought to be empty, as wouldn't need to
     * request new memory otherwise). Then we can allocate the block.
     */
    int sys_index = GetInsertionIndex(GetBlockSize(sys_block) - METADATA_TOTAL_AMOUNT);

    assert(head_list[sys_index] == NULL);
    head_list[sys_index] = sys_block;
    return AllocateBlock(head_list[sys_index], sys_index, user_requested_size);
}

/**
 * Allocates memory on the heap. Unless you *really* know what you're doing, you should always
 * pass HEAP_NO_FAULT. AllocHeap passes this automatically, but this one doesn't (in case you
 * want to allocate from the pagable pool).
 */
void* AllocHeapEx(size_t size, int flags) {
    MAX_IRQL(IRQL_SCHEDULER);

    if (flags & HEAP_ALLOW_PAGING) {
        if (GetIrql() != IRQL_STANDARD) {
            PanicEx(PANIC_INVALID_IRQL, "cannot AllocHeapEx(HEAP_ALLOW_PAGING) in irql > IRQL_STANDARD");
        }
    }

    /*
     * We cannot allocate zero blocks (as it would be useless, and couldn't be freed.)
     * Size cannot be negative as a size_t is an unsigned type.
     */
    if (size == 0) {
        return NULL;
    }

    if (size >= WARNING_LARGE_REQUEST_SIZE && ((flags & HEAP_ALLOW_PAGING) == 0 || (flags & HEAP_NO_FAULT) != 0)) {
        LogDeveloperWarning("AllocHeapEx called with allocation of size 0x%X. You should consider using MapVirt.\n", size);
    }

    if (flags == 0) {
        LogDeveloperWarning("AllocHeapEx called with flags = 0. You probably meant to pass either HEAP_ALLOW_PAGING,"
                            "or HEAP_NO_FAULT. Passing neither is valid and it puts it on the locked heap, but allocation"
                            "may cause faults. This is unlikely to be what you want.");
    }

    if (flags & HEAP_ALLOW_PAGING) {
        if (!IsHeapReinitialised()) {
            PanicEx(PANIC_ASSERTION_FAILURE, "can't allow paging without heap reinit");
        }
    }

    bool acquired = LockHeap(flags & HEAP_ALLOW_PAGING);

    size = RoundUpSize(size);

    struct block* block = FindBlock(size, flags);

#ifndef NDEBUG
```

```
818     outstanding_allocations++;
819 #endif
820
821     if (acquired) {
822         UnlockHeap(flags & HEAP_ALLOW_PAGING);
823     }
824
825     assert(((size_t) block & (ALIGNMENT - 1)) == 0);
826
827     void* ptr = AddVoidPtr(block, METADATA_LEADING_AMOUNT);
828     if (flags & HEAP_ZERO) {
829         inline_memset(ptr, 0, size);
830     }
831
832     return ptr;
833 }
834
835 void* AllocHeap(size_t size) {
836     return AllocHeapEx(size, HEAP_NO_FAULT);
837 }
838
839 void* AllocHeapZero(size_t size) {
840     return AllocHeapEx(size, HEAP_ZERO | HEAP_NO_FAULT);
841 }
842
843 void FreeHeap(void* ptr) {
844     MAX_IRQL(IRQL_SCHEDULER);
845
846     if (ptr == NULL) {
847         return;
848     }
849
850     struct block* block = SubVoidPtr(ptr, METADATA_LEADING_AMOUNT);
851     bool pagable = IsOnSwappableHeap(block);
852     block->prev = NULL;
853     block->next = NULL;
854
855     bool acquired = LockHeap(pagable);
856
857     AddBlock(block);
858
859 #ifndef NDEBUG
860     outstanding_allocations--;
861 #endif
862
863     if (acquired) {
864         UnlockHeap(pagable);
865     }
866 }
867
868 void ReinitHeap(void) {
869     heap_lock = CreateMutex("heap");
870 }
871
872 void InitHeap(void) {
873     InitSpinlock(&heap_spinlock, "heapspin", IRQL_SCHEDULER);
874     InitSpinlock(&heap_locker_lock, "locker", IRQL_HIGH);
875 }
```

File: ./mem/physical.c

```
1
2  /*
3   * mem/physical.c - Physical Memory Manager
4   *
5   * There are two allocation systems in use here. The first is a bitmap system, which has a
6   * bit for each page, which when set, indicates that a page is free. This can be used before
7   * virtual memory is available, and provides O(n) allocation time. After virtual memory is
8   * available, a stack-based system is used to provide O(1) allocation time. The bitmap is still
9   * kept in sync with the stack, to allow detection of double-free conditions and for other uses.
10  *
11  * When physical memory is low, we evict pages before we reach the out of memory condition. This
12  * allows eviction code to allocate physical memory without running out of memory.
13  */
14
15 #include <arch.h>
16 #include <physical.h>
17 #include <diskcache.h>
18 #include <common.h>
19 #include <spinlock.h>
20 #include <assert.h>
21 #include <string.h>
22 #include <irql.h>
23 #include <log.h>
24 #include <virtual.h>
25 #include <panic.h>
26
27 /*
28  * How many bits are in each entry of the bitmap allocation array. Should be the number
29  * of bits in a size_t.
30  */
31 #define BITS_PER_ENTRY (sizeof(size_t) * 8)
32
33 /*
34  * The maximum physical memory address we can use, in kilobytes.
35  */
36 #define MAX_MEMORY_KBS ARCH_MAX_RAM_KBS
37
38 /*
39  * The number of pages required to reach the maximum physical memory address.
40  */
41 #define MAX_MEMORY_PAGES (MAX_MEMORY_KBS / ARCH_PAGE_SIZE * 1024)
42
43 /*
44  * The number of entries in the bitmap allocation table required to keep track of
45  * any page up to the maximum usable physical memory address.
46  */
47 #define BITMAP_ENTRIES (MAX_MEMORY_PAGES / BITS_PER_ENTRY)
48
49 /*
50  * We will start evicting pages once we have fewer than this many pages left on the system.
51  * We can have less than this available, if in the process of eviction it causes more pages
52  * to be allocated (this is why we set it to something higher than 0 or 1, to provide a buffer
53  * for eviction to work in).
54  */
55 #define NUM_EMERGENCY_PAGES 32
56
57 /*
58  * One bit per page. Lower bits refer to lower pages. A clear bit indicates
59  * the page is unavailable (allocated / non-RAM), and a set bit indicates the
60  * page is free.
61  */
62 static size_t allocation_bitmap[BITMAP_ENTRIES];
63
64 /*
```

```
 * Stores pages that are available for us to allocate. If set to NULL, then we have yet to
 * reinitialise the physical memory manager, and so it cannot be used. The stack grows
 * upward, and the pointer is incremented after writing the value on a push. The stack stores
 * physical page numbers (indexes) instead of addresses.
 */
static size_t* allocation_stack = NULL;

/*
 * The index in to the allocation_stack bitmap where the next push operation will put
 * the value.
 */
static size_t allocation_stack_pointer = 0;

/*
 * The number of physical pages available (free) remaining in the system. Gets adjusted
 * on each allocation or deallocation. Gets set during InitPhys() when scanning the system's
 * memory map.
 */
static size_t pages_left = 0;

/*
 * The total number of allocatable pages on the system. Gets set during InitPhys() and ReinitPhys()
 */
static size_t total_pages = 0;

/*
 * The highest physical page number that exists on this system. Gets set during InitPhys() when
 * scanning the system's memory map.
 */
static size_t highest_valid_page_index = 0;

/*
 * A lock to prevent concurrent access to the physical memory manager.
 */
static struct spinlock phys_lock;

static inline bool IsBitmapEntryFree(size_t index) {
    size_t base = index / BITS_PER_ENTRY;
    size_t offset = index % BITS_PER_ENTRY;

    return allocation_bitmap[base] & (1 << offset);
}

static inline void AllocateBitmapEntry(size_t index) {
    size_t base = index / BITS_PER_ENTRY;
    size_t offset = index % BITS_PER_ENTRY;

    assert(IsBitmapEntryFree(index));

    allocation_bitmap[base] &= ~(1 << offset);
}

static inline void DeallocateBitmapEntry(size_t index) {
    size_t base = index / BITS_PER_ENTRY;
    size_t offset = index % BITS_PER_ENTRY;

    assert(!IsBitmapEntryFree(index));

    allocation_bitmap[base] |= 1 << offset;
}

static inline void PushIndex(size_t index) {
    assert(index <= highest_valid_page_index);
    allocation_stack[allocation_stack_pointer++] = index;
}

static inline size_t PopIndex(void) {
    assert(allocation_stack_pointer != 0);
    return allocation_stack[--allocation_stack_pointer];
}

/*
 * Removes an entry from the stack by value. Only to be used when absolutely required,
 * as it has O(n) runtime and is therefore very slow.
 */
static void RemoveStackEntry(size_t index) {
    for (size_t i = 0; i < allocation_stack_pointer; ++i) {
        if (allocation_stack[i] == index) {
            memmove(allocation_stack + i, allocation_stack + i + 1, (--allocation_stack_pointer - i) * sizeof(size_t));
            return;
        }
    }
}

/**
 * Deallocates a page of physical memory that was allocated with AllocPhys(). Does not affect virtual mappings -
 * that should be taken care of before deallocating.
 *
 * @param addr The address of the page to deallocate. Must be page-aligned.
 */
void DeallocPhys(size_t addr) {
    MAX_IRQL(IRQL_SCHEDULER);
    assert(addr % ARCH_PAGE_SIZE == 0);

    size_t page = addr / ARCH_PAGE_SIZE;

    AcquireSpinlockIrql(&phys_lock);

    ++pages_left;
    DeallocateBitmapEntry(page);
    if (allocation_stack != NULL) {
        PushIndex(page);
    }

    if (pages_left > NUM_EMERGENCY_PAGES * 2) {
        SetDiskCaches(DISKCACHE_NORMAL);
    }

    ReleaseSpinlockIrql(&phys_lock);
}

/**
 * Deallocates a section of physical memory that was allocated with AllocPhysContinuous(). The entire block
 * of memory must be deallocated at once, i.e. the start address of the memory should be passed in. Does not
 * affect virtual mappings - that should be taken care of before deallocating.
 *
 * @param addr The address of the section of memory to deallocate. Must be page-aligned.
 * @param size The size of the allocation. This should be the same value that was passed into AllocPhysContinuous().
 */
void DeallocPhysContiguous(size_t addr, size_t bytes) {
    MAX_IRQL(IRQL_SCHEDULER);

    size_t pages = BytesToPages(bytes);
    for (size_t i = 0; i < pages; ++i) {
        DeallocPhys(addr);
        addr += ARCH_PAGE_SIZE;
    }
}

static void EvictPagesIfNeeded(void* context) {
```

```
    (void) context;

    EXACT_IRQL(IRQL_STANDARD);

    /*
    * We can't fault later on, so we evict now if we are getting low on memory. If this faults,
    * the recursion will not cause the spinlock to be re-acquired, and so the evicted code won't
    * run again either - this prevents infinite recurison loops. These fault handlers and recursive
    * calls can allocate and make use of these 'emergency' pages that we keep by doing this eviction
    * before we actually run out of memory.
    *
    * We loop so that if the first evictions end up needing to allocate memory, we can hopefully
    * perform another eviction to make up for it (that shouldn't need extra memory).
    */

    // TODO: probs needs lock on pages_left

    extern int handling_page_fault;
    if (handling_page_fault > 0) {
        return;
    }

    if (pages_left < NUM_EMERGENCY_PAGES) {
        SetDiskCaches(DISKCACHE_TOSS);

    } else if (pages_left < NUM_EMERGENCY_PAGES * 3 / 2) {
        SetDiskCaches(DISKCACHE_REDUCE);
    }

    int timeout = 0;
    while (pages_left < NUM_EMERGENCY_PAGES && timeout < 5) {
        handling_page_fault++;
        EvictVirt();
        handling_page_fault--;
        ++timeout;
    }

    if (pages_left == 0) {
       Panic(PANIC_OUT_OF_PHYS);
    }
}

/**
 * Allocates a page of physical memory. The resulting memory can be freed with DeallocPhys(). May cause
 * pages to be evicted from RAM in order to have to enough physical memory. Direct users of this function
 * should be careful - any physical pages that have been allocated but not mapped into virtual memory
 * cannot be swapped out, and therefore they should be mapped into virtual memory.
 *
 * @return The start address of the page of physical memory, or 0 if a page could not be allocated.
 */
size_t AllocPhys(void) {
    MAX_IRQL(IRQL_SCHEDULER);

    AcquireSpinlockIrql(&phys_lock);

    if (pages_left <= NUM_EMERGENCY_PAGES) {
        LogWriteSerial("deferring EvictPagesIfNeeded (pages_left = %d)\n", pages_left);
        DeferUntilIrql(IRQL_STANDARD, EvictPagesIfNeeded, NULL);
    }

    if (pages_left == 0) {
        Panic(PANIC_OUT_OF_PHYS);
    }

    size_t index = 0;
    if (allocation_stack == NULL) {
        /*
         * No stack yet, so must use the bitmap. We could optimise and keep track of the most recently
         * returned index, but this code is only used during InitVirt(), so we'll try to keep the code
         * here short and simple.
         */
        while (!IsBitmapEntryFree(index)) {
            index = (index + 1) % MAX_MEMORY_PAGES;
        }
    } else {
        index = PopIndex();
    }

    AllocateBitmapEntry(index);
    --pages_left;

    if (pages_left == 0) {
        LogDeveloperWarning("THAT WAS THE LAST PAGE!\n");
    }

    ReleaseSpinlockIrql(&phys_lock);

    return index * ARCH_PAGE_SIZE;
}

/**
 * Allocates a section of contigous physical memory, that may or may not have requirements as
 * to where the memory can be located. Allocation in this way is very slow, and so should only
 * be called where absolutely necessary (e.g. initialising drivers). Must only be called after
 * a call to ReinitPhys() is made. Deallocation should be done by DeallocPhysContiguous(), passing
 * in the same size value as passed into AllocPhysContiguous() on allocation. Will not cause pages
 * to be evicted from RAM, so sufficient memory must exist on the system for this allocation to
 * succeed.
 *
 * @param bytes The size of the allocation, in bytes.
 * @param min_addr The allocated memory region will not contain any addresses that are lower than
 *                 this value.
 * @param max_addr The allocated memory region will not contain any addresses that are greater than
 *                 or equal to this value. If there is no maximum, set this to 0.
 * @param boundary The allocated memory will not contain any addresses that are an integer multiple
 *                 of this value (although it may start at an integer multiple of this address).
 *                 If there are no boundary requirements, set this to 0.
 * @return The start address of the returned physical memory area. If the request could not be
 *         satisfied (e.g. out of memory, no contiguous block, cannot meet requirements), then 0
 *         is returned.
 */
size_t AllocPhysContiguous(size_t bytes, size_t min_addr, size_t max_addr, size_t boundary) {
    /*
     * This function should not be called before the stack allocator is setup.
     * (There is no need for InitVirt() to use this function, and so checking here removes
     * a check that would have to be done in a loop later).
     */
    if (allocation_stack == NULL) {
        return 0;
    }

    size_t pages = BytesToPages(bytes);
    size_t min_index = (min_addr + ARCH_PAGE_SIZE - 1) / ARCH_PAGE_SIZE;
    size_t max_index = max_addr == 0 ? highest_valid_page_index + 1 : max_addr / ARCH_PAGE_SIZE;
    size_t count = 0;

    AcquireSpinlockIrql(&phys_lock);

    /*
     * We need to check we won't try to over-allocate memory, or allocate so much memory that it puts
```

```
     * us in a critical position.
     */
    if (pages + NUM_EMERGENCY_PAGES >= pages_left) {
        ReleaseSpinlockIrql(&phys_lock);
        return 0;
    }

    for (size_t index = min_index; index < max_index; ++index) {
        /*
         * Reset the counter if we are no longer contiguous, or if we have cross a boundary
         * that we can't cross.
         */
        if (!IsBitmapEntryFree(index) || (boundary != 0 && (index % (boundary / ARCH_PAGE_SIZE) == 0))) {
            count = 0;
            continue;
        }

        ++count;
        if (count == pages) {
            /*
             * Go back to the start of the section and mark it all as allocated.
             */
            size_t start_index = index - count + 1;
            while (start_index <= index) {
                AllocateBitmapEntry(start_index);
                RemoveStackEntry(start_index);
                ++start_index;
            }

            ReleaseSpinlockIrql(&phys_lock);
            return start_index * ARCH_PAGE_SIZE;
        }
    }

    ReleaseSpinlockIrql(&phys_lock);
    return 0;
}

/**
 * Initialises the physical memory manager for the first time. Must be called before any other
 * memory management function is called. It determines what memory is available on the system
 * and prepares the O(n) bitmap allocator. This will be slow, but is only needed until ReinitHeap()
 * gets called. Must only be called once.
 */
void InitPhys(void) {
    InitSpinlock(&phys_lock, "phys", IRQL_SCHEDULER);

 /*
 * Scan the memory tables and fill in the memory that is there.
 */
 while (true) {
  struct arch_memory_range* range = ArchGetMemory();

  if (range == NULL) {
   /* No more memory exists */
   break;

  } else {
   /*
   * Round conservatively (i.e., round the first page up, and the last page down)
   * so we don't accidentally allow non-existant memory to be allocated.
   */
   size_t first_page = (range->start + ARCH_PAGE_SIZE - 1) / ARCH_PAGE_SIZE;
   size_t last_page = (range->start + range->length) / ARCH_PAGE_SIZE;

   while (first_page < last_page && first_page < MAX_MEMORY_PAGES) {
                DeallocateBitmapEntry(first_page);
                ++pages_left;
                ++total_pages;

                if (first_page > highest_valid_page_index) {
                    highest_valid_page_index = first_page;
                }

                ++first_page;
   }
  }
 }
}

static void ReclaimBitmapSpace(void) {
    /*
     * We can save a tiny bit of extra physical memory on low-memory systems by deallocating the memory
     * in the bitmap that can't be reached (due to the system not having memory that goes up that high).
     * e.g. if we allow the system to access 16GB of RAM, but we only have 4MB, then we can save 31 pages
     * (or 124KB), which on a system with only 4MB of RAM is 3% of total physical memory).
     *
     * Be careful! If we do this incorrectly we will get memory corruption and some *very* mysterious bugs.
     */

    size_t num_unreachable_pages = MAX_MEMORY_PAGES - (highest_valid_page_index + 1);
    size_t num_unreachable_entries = num_unreachable_pages / BITS_PER_ENTRY;
    size_t num_unreachable_bitmap_pages = num_unreachable_entries / ARCH_PAGE_SIZE;

    size_t end_bitmap = ((size_t) allocation_bitmap) + sizeof(allocation_bitmap);

    /*
     * DO NOT ROUND UP, or variables in the same page as the end of the bitmap will also be counted as 'free',
     * causing kernel memory corruption for whatever comes in RAM after that.
     */
    size_t unreachable_region = ((end_bitmap - ARCH_PAGE_SIZE * num_unreachable_bitmap_pages)) & ~(ARCH_PAGE_SIZE - 1);

    while (num_unreachable_bitmap_pages--) {
        DeallocPhys(ArchVirtualToPhysical(unreachable_region));
        unreachable_region += ARCH_PAGE_SIZE;
        ++total_pages;
    }
}

/**
 * Reinitialises the physical memory manager with a constant-time page allocation system.
 * Must be called after virtual memory has been initialised. Must only be called once. Must
 * be called before calling AllocPageContigous() is called. Should be called as soon as
 * possible after virtual memory is available.
 */
void ReinitPhys(void) {
    assert(allocation_stack == NULL);

    allocation_stack = (size_t*) MapVirt(0, 0, (highest_valid_page_index + 1) * sizeof(size_t), VM_READ | VM_WRITE | VM_LOCK, NULL, 0);

    for (size_t i = 0; i < MAX_MEMORY_PAGES; ++i) {
        if (IsBitmapEntryFree(i)) {
            PushIndex(i);
        }
    }

    ReclaimBitmapSpace();
}
```

```
size_t GetTotalPhysKilobytes(void) {
    return total_pages * (ARCH_PAGE_SIZE / 1024);
}

size_t GetFreePhysKilobytes(void) {
    return pages_left * (ARCH_PAGE_SIZE / 1024);
}
```

File: ./mem/virtual.c

```

/**
 * mem/virtual.c - Virtual Memory Manager
 */

#include <virtual.h>
#include <avl.h>
#include <heap.h>
#include <common.h>
#include <arch.h>
#include <physical.h>
#include <assert.h>
#include <panic.h>
#include <string.h>
#include <timer.h>
#include <driver.h>
#include <thread.h>
#include <spinlock.h>
#include <log.h>
#include <sys/types.h>
#include <irql.h>
#include <irq.h>
#include <cpu.h>
#include <transfer.h>
#include <console.h>
#include <swapfile.h>
#include <vfs.h>
#include <errno.h>

// TODO: lots of locks! especially the global cpu one

static size_t SplitLargePageEntryIntoMultiple(struct vas* vas, size_t virtual, struct vas_entry* entry, int num_to_leave);
static struct vas_entry* GetVirtEntry(struct vas* vas, size_t virtual);

/**
 * Stores a pointer to any kernel VAS. Ensures that when processes are destroyed, we are using a VAS
 * that is different from the VAS that's being deleted.
 */
static struct vas* kernel_vas;

/**
 * Used in debugging as to print out the contents of the mappings tree.
 */
void AvlPrinter(void* entry_) {
    struct vas_entry* entry = (struct vas_entry*) entry_;

    LogWriteSerial(
        "[v: 0x%X, p: 0x%X; acr1: %d%d%d%d. ref: %d]; ",
        entry->virtual, entry->physical, entry->allocated, entry->cow, entry->in_ram, entry->lock, entry->ref_count
    );
}

/**
 * Whether or not virtual memory is available for use. Can be read with IsVirtInitialised(), and is set when
 * InitVirt() has completed.
 *
 * Does not have a lock, as only the bootstrap CPU should be modifiying it, and this happens before threads
 * are set up. Once set to true, it is never changed again, so there is no read/write problems.
 */
static bool virt_initialised = false;

/**
 * Used by the mappings tree as its comparison operator. Allows us to properly maintain the AVL properties,
 * and for `GetVirtEntry` to work.
 *
 * Both parameters should be cast to `struct vas_entry*` and then used as normal.
 */
static int VirtAvlComparator(void* a, void* b) {
    struct vas_entry* a_entry = (struct vas_entry*) a;
    struct vas_entry* b_entry = (struct vas_entry*) b;

    assert((a_entry->virtual & (ARCH_PAGE_SIZE - 1)) == 0);
    assert((b_entry->virtual & (ARCH_PAGE_SIZE - 1)) == 0);

    /*
     * Check for overlapping regions for multi-mapping entries, and count that as equal. This allows us
     * to return the correct entry if one of them is part of a multi-mapping entry.
     */
    size_t a_page = a_entry->virtual / ARCH_PAGE_SIZE;
    size_t b_page = b_entry->virtual / ARCH_PAGE_SIZE;
    if (a_page >= b_page && a_page < b_page + b_entry->num_pages) {
        return 0;
    }
    if (b_page >= a_page && b_page < a_page + a_entry->num_pages) {
        return 0;
    }

    return COMPARE_SIGN(a_entry->virtual, b_entry->virtual);
}

/**
 * Initialises a virtual address space in an already allocated section of memory.
 *
 * @param vas The memory to initialise a virtual address space object in.
 * @param flags Can be 0 or VAS_NO_ARCH_INIT. If VAS_NO_ARCH_INIT is provided, then no architecture-specific
 *              code will be called. This flag should only be set if called by architecture-specific functions
 *              (e.g. to create the initial address space). Normally, 0 should be passed in.
 *
 * @maxirql IRQL_SCHEDULER
 */
void CreateVasEx(struct vas* vas, int flags) {
    MAX_IRQL(IRQL_SCHEDULER);

    vas->mappings = AvlTreeCreate();

    /*
     * We are in for a world of hurt if someone is able to page fault while
     * holding the lock on a virtual address space, so better make it IRQL_SCHEDULER.
     */
    InitSpinlock(&vas->lock, "vas", IRQL_SCHEDULER);
    AvlTreeSetComparator(vas->mappings, VirtAvlComparator);
    if (!(flags & VAS_NO_ARCH_INIT)) {
        ArchInitVas(vas);
    }
}

```

```
/**
 * Allocates and initialises a new virtual address space.
 *
 * @return The virtual address space which was created.
 *
 * @maxirql IRQL_SCHEDULER
 */
struct vas* CreateVas() {
    MAX_IRQL(IRQL_SCHEDULER);

    struct vas* vas = AllocHeap(sizeof(struct vas));
    CreateVasEx(vas, 0);
    return vas;
}

struct defer_disk_access {
    struct open_file* file;
    struct vas_entry* entry;
    off_t offset;
    size_t address;
    int direction;
    bool deallocate_swap_on_read;
};

static void PerformDeferredAccess(void* data) {
    // TODO: see comment in BringIntoMemoryFromFile

    struct defer_disk_access* access = (struct defer_disk_access*) data;

    bool write = access->direction == TRANSFER_WRITE;
    size_t target_address = access->address;
    if (!write) {
        /*
         * If we're reading, the page is not yet allocated or in memory (this is so we don't have other threads
         * trying to use the partially-filled page). Therefore, we allocate a temporary page to write the data in, and
         * then we can allocate the page and copy the data while we hold a lock.
         *
         * We can't just allocate the proper page entry now, as we can't hold the spinlock over the call to ReadFile.
         */
        target_address = MapVirt(0, 0, ARCH_PAGE_SIZE, VM_LOCK | VM_READ | VM_WRITE, NULL, 0);
    }

    struct transfer tr = CreateKernelTransfer((void*) target_address, ARCH_PAGE_SIZE, access->offset, access->direction);

    int res = (write ? WriteFile : ReadFile)(access->file, &tr);
    if (res != 0) {
        /*
         * TODO: it's not actually always a failure. the only 'panic' condition is when it involves
         *       the swapfile, but this code is also used for dealing with normal file-mapped pages.
         *
         *       for file-mapped pages, failures due to reading past the end of the file should always
         *       be okay - we need to fill the rest of the page with zero though (even if that page has
         *       no file data on it, e.g. if we read really past the end of the array).
         */
        if (access->entry->swapfile) {
            Panic(PANIC_DISK_FAILURE_ON_SWAPFILE);
        } else {
            // I think for reads, it's okay to not do anything here on error, and just make use of
            // the number of bytes that were actually transfered (and therefore complete failure means
            // we just end up with a blanked-out page being allocated).
            Panic(PANIC_NOT_IMPLEMENTED);
        }
    }

    if (write) {
        UnmapVirt(access->address, ARCH_PAGE_SIZE);

    } else {
        /*
         * Now we can actually lock the page and allocate the actual mapping.
         */
        struct vas* vas = GetVas();
        AcquireSpinlockIrql(&vas->lock);

        struct vas_entry* entry = GetVirtEntry(vas, access->address);
        assert(entry->num_pages == 1);
        assert(entry->swapfile || entry->file);

        entry->lock = true;
        entry->physical = AllocPhys();
        entry->allocated = true;
        entry->allow_temp_write = true;
        entry->in_ram = true;
        entry->swapfile = false;
        ArchUpdateMapping(vas, entry);
        ArchFlushTlb(vas);

        // TODO: this should use the actual amount that was read...

        inline_memcpy((void*) access->address, (const char*) target_address, ARCH_PAGE_SIZE);

        entry->allow_temp_write = false;

        /*
         * If it was on the swapfile, we now need to mark that slot in the swapfile as free for future use.
         */
        if (access->deallocate_swap_on_read) {
            DeallocateSwapfileIndex(access->offset / ARCH_PAGE_SIZE);
        }

        /*
         * Don't perform relocations on the first load, as the first load will be when 'proper' relocation
         * happens (i.e. the 'all at once' relocations) - and therefore the quick relocation table will not
         * be created yet and we'll crash.
         *
         * The reason we can't just not do the initial big relocation and make it all work though demand loading
         * is because not all pages with driver code/data end up being marked as VM_RELOCATABLE (e.g. for small
         * parts of data segments, etc.).
         */
        bool needs_relocations = entry->relocatable && !entry->first_load;

        ArchUpdateMapping(vas, entry);
        ArchFlushTlb(vas);

        /*
         * Need to keep page locked if we're doing relocations on it - otherwise by the time that we actually
         * load in all the data we need to do the relocations (e.g. ELF headers, the symbol table), we have probably
         * already swapped out the page we are relocating (which leads to us getting nowhere).
         */
        if (!needs_relocations) {
            entry->first_load = false;
            entry->load_in_progress = false;
            entry->lock = false;
        }
        ReleaseSpinlockIrql(&vas->lock);

        UnmapVirt(target_address, ARCH_PAGE_SIZE);

        if (needs_relocations) {
            LogWriteSerial(" ----> ABOUT TO PERFORM RELOCATION FIXUPS\n");
```

```
            PerformDriverRelocationOnPage(vas, entry->relocation_base, access->address);
            AcquireSpinlockIrql(&vas->lock);
            entry->first_load = false;
            entry->load_in_progress = false;
            UnlockVirtEx(vas, access->address);
            ReleaseSpinlockIrql(&vas->lock);
            LogWriteSerial(" ----> PERFORMED RELOCATION FIXUPS\n");
        }

        LogWriteSerial(" ----> FINISHED RELOADING FROM DISK 0x%X\n", entry->virtual);
    }

    FreeHeap(access);
}

/**
 * Given a virtual page, it defers a write to disk. It creates a copy of the virtual page, so that it may be safely
 * deleted as soon as this gets called.
 */
static void DeferDiskWrite(size_t old_addr, struct open_file* file, off_t offset) {
    size_t new_addr = MapVirt(0, 0, ARCH_PAGE_SIZE, VM_LOCK | VM_READ | VM_WRITE | VM_RECURSIVE, NULL, 0);
    inline_memcpy((void*) new_addr, (const char*) old_addr, ARCH_PAGE_SIZE);

    struct defer_disk_access* access = AllocHeap(sizeof(struct defer_disk_access));
    access->address = new_addr;
    access->file = file;
    access->direction = TRANSFER_WRITE;
    access->offset = offset;
    access->deallocate_swap_on_read = false;
    DeferUntilIrql(IRQL_STANDARD_HIGH_PRIORITY, PerformDeferredAccess, (void*) access);
}

static void DeferDiskRead(size_t new_addr, struct open_file* file, off_t offset, bool deallocate_swap_on_read) {
    struct defer_disk_access* access = AllocHeap(sizeof(struct defer_disk_access));
    access->address = new_addr;
    access->file = file;
    access->direction = TRANSFER_READ;
    access->offset = offset;
    access->deallocate_swap_on_read = deallocate_swap_on_read;
    DeferUntilIrql(IRQL_STANDARD_HIGH_PRIORITY, PerformDeferredAccess, (void*) access);
}

/**
 * Evicts a particular page mapping from virtual memory, freeing up its physical page (if it had one).
 * This will often involve accessing the disk to put it on swapfile (or save modifications to a file-backed
 * page).
 *
 * @param vas The virtual address space that we're evicting from. Does not have to be the current one.
 * @param entry The virtual page to remove from virtual memory.
 *
 * @maxirql IRQL_SCHEDULER
 */
void EvictPage(struct vas* vas, struct vas_entry* entry) {
    MAX_IRQL(IRQL_PAGE_FAULT);

    assert(!entry->lock);
    assert(!entry->cow);

    AcquireSpinlockIrql(&vas->lock);

    if (!entry->in_ram) {
        /*
         * Nothing happens, as this page isn't even in RAM.
         */

    } else if (entry->file) {
        /*
         * We will just reload it from disk next time.
         */

        if (entry->write && !entry->relocatable) {
            DeferDiskWrite(entry->virtual, entry->file_node, entry->file_offset);
        }

        entry->in_ram = false;
        entry->allocated = false;
        DeallocPhys(entry->physical);
        ArchUnmap(vas, entry);
        ArchFlushTlb(vas);

    } else {
        /*
         * Otherwise, we need to mark it as swapfile.
         */
        entry->in_ram = false;
        entry->swapfile = true;
        entry->allocated = false;

        uint64_t offset = AllocateSwapfileIndex() * ARCH_PAGE_SIZE;

        //PutsConsole("PAGE OUT\n");
        LogWriteSerial(" ----> WRITING VIRT 0x%X TO SWAP: DISK INDEX 0x%X (offset 0x%X)\n", entry->virtual, (int) offset / ARCH_PAGE_SIZE, (int) offset);
        DeferDiskWrite(entry->virtual, GetSwapfile(), offset);
        entry->swapfile_offset = offset;

        ArchUnmap(vas, entry);
        DeallocPhys(entry->physical);
        ArchFlushTlb(vas);
    }

    ReleaseSpinlockIrql(&vas->lock);
}

/*
 * Lower value means it should be swapped out first.
 */
static int GetPageEvictionRank(struct vas* vas, struct vas_entry* entry) {
    (void) vas;

    /*
     * Want to evict in this order:
     *      - file and non-writable
     *      - file and writable
     *      - non-writable
     *      - writable
     *
     * When we have a way of dealing with accessed / dirty, it should be in this order:
     *
     *   0 VM_EVICT_FIRST
     *  10 FILE, NON-WRITABLE, NON-ACCESSED
     *  20 FILE, WRITABLE, NON-DIRTY, NON-ACCESSED
     *  30 FILE, NON-WRITABLE, ACCESSED
     *  40 FILE, WRITABLE, NON-DIRTY, ACCESSED
     *  50 NORMAL, NON-DIRTY, NON-ACCESSED
     *  60 NORMAL, NON-DIRTY, ACCESSED
     *  70 FILE, WRITABLE, DIRTY
     *  80 NORMAL, DIRTY
     *  90 COW
     *  150 RELOCATABLE
     *
```

```
     *  Globals add 3 points.
     */

    bool accessed;
    bool dirty;
    ArchGetPageUsageBits(vas, entry, &accessed, &dirty);
    ArchSetPageUsageBits(vas, entry, false, false);

    int penalty = (entry->global ? 3 : 0) + entry->times_swapped * 8;

    if (entry->evict_first) {
        return entry->times_swapped;

    } else if (entry->relocatable) {
        return 150;

    } else if (entry->cow) {
        return 90 + penalty;

    } else if (entry->file && !entry->write) {
        return (accessed ? 30 : 10) + penalty;

    } else if (entry->file && entry->write) {
        return (dirty ? 70 : (accessed ? 40 : 20)) + penalty;

    } else if (!dirty) {
        return (accessed ? 60 : 50) + penalty;

    } else {
        return 80 + penalty;
    }
}

struct eviction_candidate {
    struct vas* vas;
    struct vas_entry* entry;
};

#define PREV_SWAP_LIMIT 24

void FindVirtToEvictFromSubtree(struct vas* vas, struct avl_node* node, int* lowest_rank, struct eviction_candidate* lowest_ranked, int* count, struct vas_entry
    static uint8_t rand = 0;

    if (node == NULL) {
        return;
    }

    if (*lowest_rank < 10) {
        /*
        * No need to look anymore - we've already a best possible page.
        */
       return;
    }

    *count += 1;

    /*
     * After scanning through 500 entries, we'll allow early exits for less optimal pages.
     */
    int limit = (((*count - 500) / 75) + 10);
    if (*count > 500 && *lowest_rank < limit) {
        return;
    }

    struct vas_entry* entry = AvlTreeGetData(node);
    if (!entry->lock && entry->allocated) {
        int rank = GetPageEvictionRank(vas, entry);

        /*
         * To ensure we mix up who gets evicted, when there's an equality, we use it 1/4 times.
         * It is likely there are more than 4 to replace, so this ensures that we cycle through many of them.
         */
        bool equal = rank == *lowest_rank;
        if (equal) {
            equal = (rand++ & 3) == 0;
        }

        bool prev_swap = false;
        for (int i = 0; i < PREV_SWAP_LIMIT; ++i) {
            if (prev_swaps[i] == entry) {
                prev_swap = true;
                break;
            }
        }

        if ((rank < *lowest_rank || equal) && !prev_swap) {
            lowest_ranked->vas = vas;
            lowest_ranked->entry = entry;
            *lowest_rank = rank;

            if (rank == 0) {
                return;
            }
        }
    }

    FindVirtToEvictFromSubtree(vas, AvlTreeGetLeft(node), lowest_rank, lowest_ranked, count, prev_swaps);
    FindVirtToEvictFromSubtree(vas, AvlTreeGetRight(node), lowest_rank, lowest_ranked, count, prev_swaps);
}

void FindVirtToEvictFromAddressSpace(struct vas* vas, int* lowest_rank, struct eviction_candidate* lowest_ranked, bool include_globals, struct vas_entry** prev_s
    int count = 0;
    FindVirtToEvictFromSubtree(vas, AvlTreeGetRootNode(vas->mappings), lowest_rank, lowest_ranked, &count, prev_swaps);
    if (include_globals) {
        FindVirtToEvictFromSubtree(vas, AvlTreeGetRootNode(GetCpu()->global_vas_mappings), lowest_rank, lowest_ranked, &count, prev_swaps);
    }
}

/**
 * Searches through virtual memory (that doesn't necessarily have to be in the current virtual address space),
 * and finds and evicts a page of virtual memory, to try free up physical memory.
 *
 * @maxirql IRQL_STANDARD
 */
void EvictVirt(void) {
    MAX_IRQL(IRQL_PAGE_FAULT);

    if (GetSwapfile() == NULL) {
        return;
    }

    // TODO: we need to ensure that EvictVirt(), when called from the defer, does not evict any pages that were just
    //       loaded in!! This is an issue when we need to perform relocations during page faults, as that brings in a
    //       whole heap of other pages, and that often causes TryEvictPages() to straight away get rid of the page we just
    //       loaded in. Alternatively, TryEvictPages() can be a NOP the first time it is called after a page fault.
    //       This would give the code on the page that we loaded in time to 'progress' before being swapped out again.

    /*
    void TryEvictPages() {
     if (had_page_fault) {
```

```
        had_page_fault = false;
        return;
    }
    EvictVirt()
    }

    void HandlePageFault() {
        had_page_fault = true;
    }
    */

    // don't allow any of the last 8 swaps to be repeated (as an instruction may require at least 6 pages on x86
    // if it straddles many boundaries)
    static struct vas_entry* previous_swaps[PREV_SWAP_LIMIT] = {0};
    static int swap_num = 0;

    int lowest_rank = 10000;
    struct eviction_candidate lowest_ranked;
    lowest_ranked.entry = NULL;

    AcquireSpinlockIrql(&GetVas()->lock);
    FindVirtToEvictFromAddressSpace(GetVas(), &lowest_rank, &lowest_ranked, true, previous_swaps);
    ReleaseSpinlockIrql(&GetVas()->lock);

    // TODO: go through other address spaces

    while (false) {
        struct vas* vas = NULL;
        if (vas != GetVas()) {
            FindVirtToEvictFromAddressSpace(GetVas(), &lowest_rank, &lowest_ranked, false, previous_swaps);
        }
    }

    if (lowest_ranked.entry != NULL) {
        previous_swaps[swap_num++ % PREV_SWAP_LIMIT] = lowest_ranked.entry;
        EvictPage(lowest_ranked.vas, lowest_ranked.entry);
        lowest_ranked.entry->times_swapped++;
    }
}

static void InsertIntoAvl(struct vas* vas, struct vas_entry* entry) {
    assert(IsSpinlockHeld(&vas->lock));

    if (entry->global) {
        AcquireSpinlockIrql(&GetCpu()->global_mappings_lock);
        AvlTreeInsert(GetCpu()->global_vas_mappings, entry);
        ReleaseSpinlockIrql(&GetCpu()->global_mappings_lock);

    } else {
        AvlTreeInsert(vas->mappings, entry);
    }
}

static void DeleteFromAvl(struct vas* vas, struct vas_entry* entry) {
    assert(IsSpinlockHeld(&vas->lock));
    if (entry->global) {
        AcquireSpinlockIrql(&GetCpu()->global_mappings_lock);
        AvlTreeDelete(GetCpu()->global_vas_mappings, entry);
        ReleaseSpinlockIrql(&GetCpu()->global_mappings_lock);

    } else {
        AvlTreeDelete(vas->mappings, entry);
    }
}

/**
 * Adds a virtual page mapping to the specified virtual address space. This will add it both to the mapping tree
 * and the architectural paging structures (so that page faults can be raised, etc., if there is no backing yet).
 *
 * @param vas       The virtual address space to map this page to
 * @param physical  Only used if VM_LOCK is specified in flags. Determines the physical page that will back the
 *                      virtual mapping. If VM_LOCK is set, and this is 0, then a physical page will be allocated. If
 *                      VM_LOCK is set, and this is non-zero, then that physical address will be used.
 * @param virtual   The virtual address to map the memory to. This should be non-zero.
 * @param flags     Various bitflags to affect the attributes of the mapping. Flags that are used here are:
 *                      VM_READ     : if set, the page will be marked as readable
 *                      VM_WRITE    : if set, the page will be marked as writable
 *                      VM_USER     : if set, then usermode can access this page without faulting
 *                      VM_EXEC     : if set, then code can be executed in this page
 *                      VM_LOCK     : if set, the page will immediately get a physical memory backing, and will not be
 *                                    paged out
 *                      VM_FILE     : if set, this page is file-backed
 *                  All other flags are ignored by this function.
 * @param file      If VM_FILE is set, then the page is backed by this file, starting at the position specified by pos.
 * @param pos       If VM_FILE is set, then this is the offset into the file where the page is mapped to.
 *
 * @maxirql IRQL_SCHEDULER
 */
static void AddMapping(struct vas* vas, size_t physical, size_t virtual, int flags, struct open_file* file, off_t pos, size_t number) {
    MAX_IRQL(IRQL_SCHEDULER);

    assert(!(file != NULL && (flags & VM_FILE) == 0));

    struct vas_entry* entry = AllocHeapZero(sizeof(struct vas_entry));
    entry->allocated = false;

    bool lock = flags & VM_LOCK;
    entry->lock = lock;
    entry->in_ram = lock;

    size_t relocation_base = 0;
    if (flags & VM_RELOCATABLE) {
        relocation_base = physical;
        physical = 0;
    }

    if (lock) {
        /*
         * We are not allowed to check if the physical page is allocated/free, because it might come
         * from a VM_MAP_HARDWARE request, which can map non-RAM pages.
         */
        if (physical == 0) {
            physical = AllocPhys();
            entry->allocated = true;
        }
    }

    /*
     * MapVirt checks for conflicting flags and returns, so this code doesn't need to worry about that.
     */
    entry->virtual = virtual;
    entry->times_swapped = 0;
    entry->read = (flags & VM_READ) ? 1 : 0;
    entry->write = (flags & VM_WRITE) ? 1 : 0;
    entry->exec = (flags & VM_EXEC) ? 1 : 0;
    entry->file = (flags & VM_FILE) ? 1 : 0;
    entry->user = (flags & VM_USER) ? 1 : 0;
    entry->evict_first = (flags & VM_EVICT_FIRST) ? 1 : 0;
    entry->relocatable = (flags & VM_RELOCATABLE) ? 1 : 0;
    entry->allow_temp_write = false;
```

```
637     entry->load_in_progress = false;
638     entry->global = !(flags & VM_LOCAL);
639     entry->physical = physical;
640     entry->ref_count = 1;
641     entry->file_offset = pos;
642     entry->file_node = file;
643     entry->swapfile = false;
644     entry->first_load = entry->relocatable;
645     entry->num_pages = number;
646 
647     if (entry->relocatable) {
648         entry->relocation_base = relocation_base;
649     } else {
650         entry->swapfile_offset = 0xDEADDEAD;
651     }
652 
653     LogWriteSerial("Adding mapping at 0x%X to vas 0x%X - num is %d. flags = 0x%X, rwxgu'lfia = %d%d%d%d%d'%d%d%d%d. p 0x%X.\n",
654         entry->virtual, vas, number, flags,
655         entry->read, entry->write, entry->exec, entry->global, entry->user, entry->lock, entry->file, entry->in_ram, entry->allocated,
656         entry->physical
657     );
658 
659     /*
660      * TODO: later on, check if shared, and add phys->virt entry if needed
661      */
662 
663     if (!(flags & VM_RECURSIVE) == 0) {
664         AcquireSpinlockIrql(&vas->lock);
665     }
666     InsertIntoAvl(vas, entry);
667     ArchAddMapping(vas, entry);
668 
669     if (entry->lock && (flags & VM_MAP_HARDWARE) == 0) {
670         if (GetVas() == vas) {
671             /*
672              * Need to zero out the page - this must happen on first load in, and as we have to load in
673              * locked pages now, we must do it now.
674              */
675             memset((void*) entry->virtual, 0, entry->num_pages * ARCH_PAGE_SIZE);
676         } else {
677             LogDeveloperWarning("yuck. PAGE HAS NOT BEEN ZEROED!\n");
678         }
679     }
680 
681     if (!(flags & VM_RECURSIVE) == 0) {
682         ReleaseSpinlockIrql(&vas->lock);
683     }
684 }
685 
686 static bool IsRangeInUse(struct vas* vas, size_t virtual, size_t pages) {
687     bool in_use = false;
688 
689     struct vas_entry dummy;
690     dummy.num_pages = 1;
691     dummy.virtual = virtual;
692 
693     /*
694      * We have to loop over the local one, and if it isn't there, the global one. We do this
695      * in separate loops to prevent the need to acquire both spinlocks at once, which could lead
696      * to a deadlock.
697      */
698 
699     AcquireSpinlockIrql(&vas->lock);
700     for (size_t i = 0; i < pages; ++i) {
701         if (AvlTreeContains(vas->mappings, (void*) &dummy)) {
702             in_use = true;
703             break;
704         }
705         dummy.virtual += ARCH_PAGE_SIZE;
706     }
707     ReleaseSpinlockIrql(&vas->lock);
708 
709     if (in_use) {
710         return true;
711     }
712 
713     AcquireSpinlockIrql(&GetCpu()->global_mappings_lock);
714     dummy.virtual = virtual;
715     for (size_t i = 0; i < pages; ++i) {
716         if (AvlTreeContains(GetCpu()->global_vas_mappings, (void*) &dummy)) {
717             in_use = true;
718             break;
719         }
720         dummy.virtual += ARCH_PAGE_SIZE;
721     }
722     ReleaseSpinlockIrql(&GetCpu()->global_mappings_lock);
723 
724     return in_use;
725 }
726 
727 static size_t AllocVirtRange(struct vas* vas, size_t pages, int flags) {
728     /*
729      * TODO: make this deallocatable, and not x86 specific (with that memory address)
730      */
731     if (flags & VM_LOCAL) {
732         /*
733          * Also needs to use the vas to work out what's allocated in that vas
734          */
735         (void) vas;
736         static size_t hideous_allocator = 0x20000000U;
737         size_t retv = hideous_allocator;
738         hideous_allocator += pages * ARCH_PAGE_SIZE;
739         return retv;
740 
741     } else {
742         /*
743          * TODO: this probably needs a global lock of some sort.
744          */
745         static size_t hideous_allocator = ARCH_KRNL_SBRK_BASE;
746         size_t retv = hideous_allocator;
747         hideous_allocator += pages * ARCH_PAGE_SIZE;
748         return retv;
749     }
750 }
751 
752 static void FreeVirtRange(struct vas* vas, size_t virtual, size_t pages) {
753     (void) virtual;
754     (void) vas;
755     (void) pages;
756 }
757 
758 /**
759  * Creates a virtual memory mapping.
760  *
761  * All mapped pages will be zeroed out (either on first use, or if locked, when allocated) - except if VM_MAP_HARDWARE or
762  * VM_FILE is set. If VM_FILE is set, reading beyond the end of the file, but within the page limit, will read zeroes.
763  *
764  * @param vas       The virtual address space to map this page to
765  * @param physical  Only used if VM_LOCK is specified in flags. Determines the physical page that will back the
766  *                      virtual mapping. If VM_LOCK is set, and this is 0, then a physical page will be allocated. If
```

```
 *                      VM_LOCK is set, and this is non-zero, then that physical address will be used. In this instance,
 *                      VM_MAP_HARDWARE must also be set. If VM_MAP_HARDWARE is not set, this value must be 0.
 * @param virtual   The virtual address to map the memory to. If this is 0, then a virtual memory region of the correct
 *                      size will be allocated.
 * @param pages     The number of contiguous pages to map in this way
 * @param flags     Various bitflags to affect the attributes of the mapping. Flags that are used here are:
 *                      VM_READ         : if set, the page will be marked as readable
 *                      VM_WRITE        : if set, the page will be marked as writable. On some architectures, this may have the
 *                                        effect of implying VM_READ as well.
 *                      VM_USER         : if set, then usermode can access this page without faulting
 *                      VM_EXEC         : if set, then code can be executed in this page
 *                      VM_LOCK         : if set, the page will immediately get a physical memory backing, and will not be
 *                                        paged out
 *                      VM_FILE         : if set, this page is file-backed. Cannot be combined with VM_MAP_HARDWARE.
 *                                        Cannot be combined with VM_LOCK.
 *                      VM_MAP_HARDWARE : if set, a physical address can be specified for the backing. If this flag is set,
 *                                        then VM_LOCK must also be set, and VM_FILE must be clear.
 *                      VM_LOCAL        : if set, it is only mapped into the current virtual address space. If set, it is mapped
 *                                        into the kernel virtual address space.
 *                      VM_RECURSIVE    : must be set if and only if this call to MapVirtEx is being called with the virtual
 *                                        address space lock already held. Does not affect the page, only the call to MapVirtEx
 *                                        When set, the lock is not automatically acquired or released as it is assumed to be
 *                                        already held.
 *                      VM_RELOCATABLE  : if set, then this page will have driver relocations applied to it when it is swapped
 *                                        in. VM_FILE must be set as well. `file` should be set to the driver's file.
 *                      VM_FIXED_VIRT   : if set, then the virtual address specified in `virtual` will be required to be used -
 *                                        if any page required for the mapping of this size is already allocated, the allocation
 *                                        will fail. If clear, then another virtual address may be used in order to satisfy a
 *                                        request.
 *                      VM_EVICT_FIRST  : indicates to the virtual memory manager that when memory is low, this page should be
 *                                        evicted before other pages
 * @param file      If VM_FILE is set, then the page is backed by this file, starting at the position specified by pos.
 *                      If VM_FILE is clear, then this value must be NULL.
 * @param pos       If VM_FILE is set, then this is the offset into the file where the page is mapped to. If VM_FILE is clear,
 *                      then this value must be 0.
 *
 * @maxirql IRQL_SCHEDULER
 */
static size_t MapVirtEx(struct vas* vas, size_t physical, size_t virtual, size_t pages, int flags, struct open_file* file, off_t pos) {
    MAX_IRQL(IRQL_SCHEDULER);

    /*
     * We only specify a physical page when we need to map hardware directly (i.e. it's not
     * part of the available RAM the physical memory manager can give).
     */
    if (physical != 0 && (flags & (VM_MAP_HARDWARE | VM_RELOCATABLE)) == 0) {
        return 0;
    }

    if ((flags & VM_MAP_HARDWARE) && (flags & VM_FILE)) {
        return 0;
    }

    if ((flags & VM_MAP_HARDWARE) && (flags & VM_LOCK) == 0) {
        return 0;
    }

    if ((flags & VM_FILE) && file == NULL) {
        return 0;
    }

    if ((flags & VM_FILE) == 0 && (file != NULL || pos != 0)) {
        return 0;
    }

    if ((flags & VM_RELOCATABLE) && (flags & VM_FILE) == 0) {
        return 0;
    }

    if ((flags & VM_RELOCATABLE) && physical == 0) {
        return 0;
    }

    if ((flags & VM_FILE) && (flags & VM_LOCK)) {
        return 0;
    }

    /*
     * Get a virtual memory range that is not currently in use.
     */
    if (virtual == 0) {
        virtual = AllocVirtRange(vas, pages, flags & VM_LOCAL);

    } else {
        // TODO: need to lock here to make the israngeinuse and allocvirtrange to be atomic
        if (IsRangeInUse(vas, virtual, pages)) {
            if (flags & VM_FIXED_VIRT) {
                return 0;
            }

            virtual = AllocVirtRange(vas, pages, flags & VM_LOCAL);
        }
    }

    /*
     * No point doing the multi-page mapping with only 2 pages, as the splitting cost is probably
     * going to be greater than actually just adding 2 pages in the first place.
     *
     * May want to increase this value furher in the future (e.g. maybe to 4 or 8)?
     */
    bool multi_page_mapping = (((flags & VM_LOCK) == 0) || ((flags & VM_MAP_HARDWARE) != 0)) && pages >= 3;

    for (size_t i = 0; i < (multi_page_mapping ? 1 : pages); ++i) {
        AddMapping(
            vas,
            (flags & VM_RELOCATABLE) ? physical : (physical == 0 ? 0 : (physical + i * ARCH_PAGE_SIZE)),
            virtual + i * ARCH_PAGE_SIZE,
            flags,
            file,
            pos + i * ARCH_PAGE_SIZE,
            multi_page_mapping ? pages : 1
        );
    }

    if (vas == GetVas()) {
        ArchFlushTlb(vas);
    }

    return virtual;
}

/**
 * Creates a virtual memory mapping in the current virtual address space.
 *
 * @param physical  See `MapVirtEx`
 * @param virtual   See `MapVirtEx`
 * @param pages     The minimum number of bytes to map
 * @param flags     See `MapVirtEx`
 * @param file      See `MapVirtEx`
 * @param pos       See `MapVirtEx`
```

```
 *
 * @maxirql IRQL_SCHEDULER
 */
size_t MapVirt(size_t physical, size_t virtual, size_t bytes, int flags, struct open_file* file, off_t pos) {
    MAX_IRQL(IRQL_SCHEDULER);
    size_t pages = BytesToPages(bytes);
    size_t ret = MapVirtEx(GetVas(), physical, virtual, pages, flags, file, pos);
    return ret;
}

static struct vas_entry* GetVirtEntry(struct vas* vas, size_t virtual) {
    struct vas_entry dummy;
    dummy.num_pages = 1;
    dummy.virtual = virtual & ~(ARCH_PAGE_SIZE - 1);

    assert(IsSpinlockHeld(&vas->lock));

    struct vas_entry* res = (struct vas_entry*) AvlTreeGet(vas->mappings, (void*) &dummy);
    if (res == NULL) {
        AcquireSpinlockIrql(&GetCpu()->global_mappings_lock);
        res = (struct vas_entry*) AvlTreeGet(GetCpu()->global_vas_mappings, (void*) &dummy);
        ReleaseSpinlockIrql(&GetCpu()->global_mappings_lock);
    }
    return res;
}

size_t GetPhysFromVirt(size_t virtual) {
    struct vas* vas = GetVas();
    AcquireSpinlockIrql(&vas->lock);
    struct vas_entry* entry = GetVirtEntry(vas, virtual);
    size_t result = entry->physical;

    /*
     * Handle mappings of more than 1 page at a time by adding the extra offset
     * from the start of the mapping.
     */
    size_t target_page = virtual / ARCH_PAGE_SIZE;
    size_t entry_page = entry->virtual / ARCH_PAGE_SIZE;
    if (entry_page < target_page) {
        result += (target_page - entry_page) * ARCH_PAGE_SIZE;
    }

    ReleaseSpinlockIrql(&vas->lock);
    return result;
}

static size_t SplitLargePageEntryIntoMultiple(struct vas* vas, size_t virtual, struct vas_entry* entry, int num_to_leave) {
    if (entry->num_pages == 1) {
        return 1;
    }

    if (entry->ref_count != 1) {
        LogDeveloperWarning("Splitting multi-mapping with ref_count != 1, this hasn't been tested!\n");
    }

    /*
     * Although it can't be allocated, it can be in RAM (e.g. for VM_MAP_HARDWARE).
     */
    assert(!entry->allocated);
    assert(!entry->swapfile);

    size_t entry_page = entry->virtual / ARCH_PAGE_SIZE;
    size_t target_page = virtual / ARCH_PAGE_SIZE;

    /*
     * Split off anything before this page.
     */
    if (entry_page < target_page) {
        size_t num_beforehand = target_page - entry_page;

        struct vas_entry* pre_entry = AllocHeap(sizeof(struct vas_entry));
        *pre_entry = *entry;

        pre_entry->num_pages = num_beforehand;
        entry->num_pages -= num_beforehand;
        entry->virtual += num_beforehand * ARCH_PAGE_SIZE;

        /*
         * For multi-mapping for VM_MAP_HARDWARE
         */
        if (entry->physical != 0) {
            entry->physical += num_beforehand * ARCH_PAGE_SIZE;
        }

        if (entry->file) {
            entry->file_offset += num_beforehand * ARCH_PAGE_SIZE;
        }

        InsertIntoAvl(vas, pre_entry);
    }

    /*
     * There's now no pages beforehand. Now we need to check if there are any other pages
     * after this.
     */
    if (entry->num_pages > num_to_leave) {
        struct vas_entry* post_entry = AllocHeap(sizeof(struct vas_entry));
        *post_entry = *entry;

        post_entry->num_pages -= num_to_leave;
        entry->num_pages = num_to_leave;

        post_entry->virtual += ARCH_PAGE_SIZE * num_to_leave;

        /*
         * For multi-mapping for VM_MAP_HARDWARE
         */
        if (entry->physical != 0) {
            post_entry->physical += ARCH_PAGE_SIZE * num_to_leave;
        }

        if (entry->file) {
            post_entry->file_offset += ARCH_PAGE_SIZE * num_to_leave;
        }

        InsertIntoAvl(vas, post_entry);
    }

    return entry->num_pages;
}

static void BringIntoMemoryFromCow(struct vas_entry* entry) {
    /*
    * If someone deallocates a COW page in another process to get the ref
    * count back to 1 already, then we just have the page to ourselves again.
    */
    if (entry->ref_count == 1) {
        entry->cow = false;
        ArchUpdateMapping(GetVas(), entry);
        ArchFlushTlb(GetVas());
```

```
        return;
    }

    uint8_t page_data[ARCH_PAGE_SIZE];
    inline_memcpy(page_data, (void*) entry->virtual, ARCH_PAGE_SIZE);

    entry->ref_count--;

    if (entry->ref_count == 1) {
        entry->cow = false;
    }

    struct vas_entry* new_entry = AllocHeap(sizeof(struct vas_entry));
    *new_entry = *entry;
    new_entry->ref_count = 1;
    new_entry->physical = AllocPhys();
    new_entry->allocated = true;
    DeleteFromAvl(GetVas(), entry);
    FreeHeap(entry);
    ArchUpdateMapping(GetVas(), entry);
    ArchFlushTlb(GetVas());
    inline_memcpy((void*) entry->virtual, page_data, ARCH_PAGE_SIZE);
}

static void BringIntoMemoryFromFile(struct vas_entry* entry, size_t faulting_virt) {
    // TODO: need to test that you're allowed to read past the end of the file (even into other pages)
    //       if the size mapped allows it, and just get zeros

    SplitLargePageEntryIntoMultiple(GetVas(), faulting_virt, entry, 1);
    entry->load_in_progress = true;
    ArchUpdateMapping(GetVas(), entry);
    ArchFlushTlb(GetVas());
    DeferDiskRead(entry->virtual, entry->file_node, entry->file_offset, false);
}

static void BringIntoMemoryFromSwapfile(struct vas_entry* entry) {
    assert(!entry->file);

    uint64_t offset = entry->swapfile_offset;
    entry->load_in_progress = true;
    ArchUpdateMapping(GetVas(), entry);
    ArchFlushTlb(GetVas());
    DeferDiskRead(entry->virtual, GetSwapfile(), offset, true);
}

static void BringInBlankPage(struct vas* vas, struct vas_entry* entry, size_t faulting_virt, int fault_type) {
    if ((fault_type & VM_READ) && !entry->read) {
        UnhandledFault();
    }
    if ((fault_type & VM_WRITE) && !entry->write) {
        UnhandledFault();
    }
    if ((fault_type & VM_EXEC) && !entry->exec) {
        UnhandledFault();
    }

    SplitLargePageEntryIntoMultiple(vas, faulting_virt, entry, 1);
    assert(entry->num_pages == 1);

    entry->physical = AllocPhys();
    entry->allocated = true;
    entry->in_ram = true;
    entry->allow_temp_write = true;
    assert(!entry->swapfile);
    ArchUpdateMapping(vas, entry);
    ArchFlushTlb(vas);

    inline_memset((void*) entry->virtual, 0, ARCH_PAGE_SIZE);
    entry->allow_temp_write = false;
    ArchUpdateMapping(vas, entry);
    ArchFlushTlb(vas);
}

static int BringIntoMemory(struct vas* vas, struct vas_entry* entry, bool allow_cow, size_t faulting_virt, int fault_type) {
    (void) vas;
    assert(IsSpinlockHeld(&vas->lock));

    if (entry->cow && allow_cow) {
        assert(entry->num_pages == 1);
        BringIntoMemoryFromCow(entry);
        return 0;
    }

    if (entry->file && !entry->in_ram) {
        BringIntoMemoryFromFile(entry, faulting_virt);
        return 0;
    }

    if (entry->swapfile) {
        assert(entry->num_pages == 1);
        BringIntoMemoryFromSwapfile(entry);
        return 0;
    }

    if (!entry->in_ram) {
        BringInBlankPage(vas, entry, faulting_virt, fault_type);
        return 0;
    }

    return EINVAL;


bool LockVirtEx(struct vas* vas, size_t virtual) {
    struct vas_entry* entry = GetVirtEntry(vas, virtual);

    if (!entry->in_ram) {
        SplitLargePageEntryIntoMultiple(vas, virtual, entry, 1);
        int res = BringIntoMemory(vas, entry, true, virtual, 0);
        if (res != 0) {
            Panic(PANIC_CANNOT_LOCK_MEMORY);
        }
        assert(entry->in_ram);
    }

    bool old_lock = entry->lock;
    entry->lock = true;
    return old_lock;
}

void UnlockVirtEx(struct vas* vas, size_t virtual) {
    struct vas_entry* entry = GetVirtEntry(vas, virtual);
    SplitLargePageEntryIntoMultiple(vas, virtual, entry, 1);
    entry->lock = false;
}

bool LockVirt(size_t virtual) {
    struct vas* vas = GetVas();
    AcquireSpinlockIrql(&vas->lock);
    bool res = LockVirtEx(vas, virtual);
    ReleaseSpinlockIrql(&vas->lock);
```

```
1157    return res;
1158}
1159
1160void UnlockVirt(size_t virtual) {
1161    struct vas* vas = GetVas();
1162    AcquireSpinlockIrql(&vas->lock);
1163    UnlockVirtEx(vas, virtual);
1164    ReleaseSpinlockIrql(&vas->lock);
1165}
1166
1167void SetVirtPermissions(size_t virtual, int set, int clear) {
1168    /*
1169     * Only allow these flags to be set / cleared.
1170     */
1171    if ((set | clear) & ~(VM_READ | VM_WRITE | VM_EXEC | VM_USER)) {
1172        assert(false);
1173        return;
1174    }
1175
1176    struct vas* vas = GetVas();
1177    AcquireSpinlockIrql(&vas->lock);
1178
1179    struct vas_entry* entry = GetVirtEntry(vas, virtual);
1180    if (entry == NULL) {
1181        PanicEx(PANIC_ASSERTION_FAILURE, "[SetVirtPermissions] got null back for virt entry");
1182    }
1183
1184    SplitLargePageEntryIntoMultiple(vas, virtual, entry, 1);
1185
1186    entry->read = (set & VM_READ) ? true : (clear & VM_READ ? false : entry->read);
1187    entry->write = (set & VM_WRITE) ? true : (clear & VM_WRITE ? false : entry->write);
1188    entry->exec = (set & VM_EXEC) ? true : (clear & VM_EXEC ? false : entry->exec);
1189    entry->user = (set & VM_USER) ? true : (clear & VM_USER ? false : entry->user);
1190
1191    ArchUpdateMapping(vas, entry);
1192    ArchFlushTlb(vas);
1193
1194    ReleaseSpinlockIrql(&vas->lock);
1195}
1196
1197int GetVirtPermissions(size_t virtual) {
1198    struct vas* vas = GetVas();
1199    AcquireSpinlockIrql(&vas->lock);
1200    struct vas_entry* entry_ptr = GetVirtEntry(GetVas(), virtual);
1201    if (entry_ptr == NULL)
1202        ReleaseSpinlockIrql(&vas->lock);
1203        return 0;
1204    }
1205    struct vas_entry entry = *entry_ptr;
1206    ReleaseSpinlockIrql(&vas->lock);
1207
1208    int permissions = 0;
1209    if (entry.read) permissions |= VM_READ;
1210    if (entry.write) permissions |= VM_WRITE;
1211    if (entry.exec) permissions |= VM_EXEC;
1212    if (entry.lock) permissions |= VM_LOCK;
1213    if (entry.file) permissions |= VM_FILE;
1214    if (entry.user) permissions |= VM_USER;
1215    if (!entry.global) permissions |= VM_LOCAL;
1216    if (entry.relocatable) permissions |= VM_RELOCATABLE;
1217
1218    return permissions;
1219}
1220
1221int UnmapVirtEx(struct vas* vas, size_t virtual, size_t pages) {
1222    bool needs_tlb_flush = false;
1223
1224    for (size_t i = 0; i < pages; ++i) {
1225        struct vas_entry* entry = GetVirtEntry(vas, virtual + i * ARCH_PAGE_SIZE);
1226        if (entry == NULL) {
1227            return ENOMEM;
1228        }
1229
1230        SplitLargePageEntryIntoMultiple(vas, virtual, entry, 1);        // TODO: multi-pages
1231
1232        assert(entry->ref_count > 0);
1233        entry->ref_count--;
1234
1235        if (entry->ref_count == 0) {
1236            if (entry->file && entry->write && entry->in_ram) {
1237                DeferDiskWrite(entry->virtual, entry->file_node, entry->file_offset);
1238            }
1239            if (entry->in_ram) {
1240                ArchUnmap(vas, entry);
1241                needs_tlb_flush = true;
1242            }
1243            if (entry->swapfile) {
1244                assert(!entry->allocated);
1245                DeallocateSwapfileIndex(entry->physical / ARCH_PAGE_SIZE);
1246            }
1247            if (entry->allocated) {
1248                assert(!entry->swapfile);   // can't be on swap, as putting on swap clears allocated bit
1249                DeallocPhys(entry->physical);
1250            }
1251
1252            DeleteFromAvl(vas, entry);
1253            FreeHeap(entry);
1254            FreeVirtRange(vas, virtual + i * ARCH_PAGE_SIZE, entry->num_pages);
1255        }
1256    }
1257
1258    if (needs_tlb_flush) {
1259        ArchFlushTlb(vas);
1260    }
1261
1262    return 0;
1263}
1264
1265int UnmapVirt(size_t virtual, size_t bytes) {
1266    struct vas* vas = GetVas();
1267    AcquireSpinlockIrql(&vas->lock);
1268    int res = UnmapVirtEx(vas, virtual, BytesToPages(bytes));
1269    ReleaseSpinlockIrql(&vas->lock);
1270    return res;
1271}
1272
1273static void CopyVasRecursive(struct avl_node* node, struct vas* new_vas) {
1274    if (node == NULL) {
1275        return;
1276    }
1277
1278    CopyVasRecursive(AvlTreeGetLeft(node), new_vas);
1279    CopyVasRecursive(AvlTreeGetRight(node), new_vas);
1280
1281    struct vas_entry* entry = AvlTreeGetData(node);
1282
1283    if (entry->lock) {
1284        /*
1285        * Got to add the new entry right now. We know it must be in memory as it
1286        * is locked.
```

```
1287        */
1288        assert(entry->in_ram);
1289
1290        if (entry->allocated) {
1291            /*
1292            * Copy the physical page. We do this by copying the data into a buffer,
1293            * putting a new physical page in the existing VAS and then copying the
1294            * data there. Then the original physical page that was there is free to use
1295            * as the copy.
1296            */
1297            uint8_t page_data[ARCH_PAGE_SIZE];  // TODO: MapVirt this ?
1298            inline_memcpy(page_data, (void*) entry->virtual, ARCH_PAGE_SIZE);
1299            size_t new_physical = entry->physical;
1300            entry->physical = AllocPhys();
1301            ArchUpdateMapping(GetVas(), entry);
1302            ArchFlushTlb(GetVas());
1303            inline_memcpy((void*) entry->virtual, page_data, ARCH_PAGE_SIZE);
1304
1305            struct vas_entry* new_entry = AllocHeap(sizeof(struct vas_entry));
1306            *new_entry = *entry;
1307            new_entry->ref_count = 1;
1308            new_entry->physical = new_physical;
1309            new_entry->allocated = true;
1310            AvlTreeInsert(new_vas->mappings, entry);        // don't need to insert global - we're copying so it's already in global
1311            ArchAddMapping(new_vas, entry);
1312
1313        } else {
1314            LogWriteSerial("fork() on a hardware-mapped page is not implemented yet");
1315            PanicEx(PANIC_NOT_IMPLEMENTED, "CopyVasRecursive");
1316        }
1317
1318    } else {
1319        /*
1320        * If it's on swap, it's okay to still mark it as COW, as when we reload we will
1321        * try to do the 'copy'-on-write, and then we will reload from swap, and it will
1322        * then reload and then be copied. Alternatively, if it is read, then it gets brought
1323        * back into memory, but as a COW page still.
1324        *
1325        * BSS memory works fine like this too (but will incur another fault when it is used).
1326        *
1327        * At this stage (where shared memory doesn't exist yet), file mapped pages will also
1328        * be COWed. This means there will two copies of the file in memory should they write
1329        * to it. The final process to release memory will ultimately 'win' and have its changes
1330        * perserved to disk (the others will get overwritten).
1331        */
1332        entry->cow = true;
1333        entry->ref_count++;
1334
1335        // again, no need to add to global - it's already there!
1336        AvlTreeInsert(new_vas->mappings, entry);
1337
1338        ArchUpdateMapping(GetVas(), entry);
1339        ArchAddMapping(new_vas, entry);
1340    }
1341}
1342
1343struct vas* CopyVas(void) {
1344    struct vas* vas = GetVas();
1345    struct vas* new_vas = CreateVas();
1346
1347    AcquireSpinlockIrql(&vas->lock);
1348    // no need to change global - it's already there!
1349    CopyVasRecursive(AvlTreeGetRootNode(vas->mappings), new_vas);
1350    ArchFlushTlb(vas);
1351    ReleaseSpinlockIrql(&vas->lock);
1352
1353    return new_vas;
1354}
1355
1356struct vas* GetVas(void) {
1357    // TODO: cpu probably needs to have a lock object in it called current_vas_lock, which needs to be held whenever
1358    //       someone reads or writes to  current_vas;
1359    return GetCpu()->current_vas;
1360}
1361
1362void SetVas(struct vas* vas) {
1363    GetCpu()->current_vas = vas;
1364    ArchSetVas(vas);
1365}
1366
1367struct vas* GetKernelVas(void) {
1368    return kernel_vas;
1369}
1370
1371void InitVirt(void) {
1372    // TODO: cpu probably needs to have a lock object in it called current_vas_lock, which needs to be held whenever
1373    //       someone reads or writes to current_vas;
1374
1375    assert(!virt_initialised);
1376    GetCpu()->global_vas_mappings = AvlTreeCreate();
1377    AvlTreeSetComparator(GetCpu()->global_vas_mappings, VirtAvlComparator);
1378    ArchInitVirt();
1379
1380    kernel_vas = GetVas();
1381    virt_initialised = true;
1382}
1383
1384
1385/**
1386 * Handles a page fault. Only to be called by the low-level, platform specific interrupt handler when a page
1387 * fault occurs. It will attempt to resolve any fault (e.g. handling copy-on-write, swapfile, file-backed, etc.).
1388 *
1389 * @param faulting_virt The virtual address that was accessed that caused the page fault
1390 * @param fault_type The reason why a page fault occured. Is a bitfield of VM_WRITE, VM_READ, VM_USER and VM_EXEC.
1391 *                   VM_READ should be set if a non-present page was accessed. VM_USER should be set for permission
1392 *                   faults, and VM_WRITE should be set if the operation was caused by a write (as opposed to a read).
1393 *                   VM_EXEC should be set if execution tried to occur in a non-executable page.
1394 *
1395 * @maxirql IRQL_PAGE_FAULT
1396 */
1397int handling_page_fault = 0;
1398
1399void HandleVirtFault(size_t faulting_virt, int fault_type) {
1400    if (GetIrql() >= IRQL_SCHEDULER) {
1401        PanicEx(PANIC_INVALID_IRQL, "page fault while IRQL >= IRQL_SCHEDULER. is some clown holding a spinlock while "
1402                                    "executing pageable code? or calling AllocHeapEx wrong with a lock held?");
1403    }
1404
1405    struct vas* vas = GetVas();
1406    AcquireSpinlockIrql(&vas->lock);
1407    ++handling_page_fault;
1408
1409    struct vas_entry* entry = GetVirtEntry(vas, faulting_virt);
1410
1411    if (entry == NULL) {
1412        UnhandledFault();
1413    }
1414
1415    if (entry->load_in_progress) {
1416        --handling_page_fault;
```

```
        ReleaseSpinlockIrql(&vas->lock);
        Schedule();
        return;
    }

    /*
     * Sanity check that our flags are configured correctly.
     */
    assert(!(entry->in_ram && entry->swapfile));
    assert(!(entry->file && entry->swapfile));
    assert(!(!entry->in_ram && entry->lock));
    assert(!(entry->cow && entry->lock));

    // TODO: check for access violations (e.g. user using a supervisor page)
    //          (read / write is not necessarily a problem, e.g. COW)

    int result = BringIntoMemory(vas, entry, fault_type & VM_WRITE, faulting_virt, fault_type);
    if (result != 0) {
        UnhandledFault();
    }

    --handling_page_fault;
    ReleaseSpinlockIrql(&vas->lock);
}

/**
 * Determines whether or not virtual memory has been initialised yet. This can be used to determine if it
 * is possible to call any virtual memory functions (e.g. in the physical memory and heap allocators).
 *
 * @return True if virtual memory is available, false otherwise.
 *
 * @maxirql IRQL_HIGH
 */
bool IsVirtInitialised(void) {
    return virt_initialised;
}

size_t BytesToPages(size_t bytes) {
    return (bytes + ARCH_PAGE_SIZE - 1) / ARCH_PAGE_SIZE;
}

void DestroyVas(struct vas* vas) {
    /*
     * TODO: implement this
     */
    (void) vas;

    if (vas == GetVas()) {
        Panic(PANIC_VAS_TRIED_TO_SELF_DESTRUCT);
    }

    // TODO: may need to add reference counting later on (depending on what we need),
    //       just decrement here, and only delete if got to 0.
}
```

File: ./debug/.DS_Store

[binary]

File: ./debug/framework/hostio.c

```

#ifndef NDEBUG

#include <common.h>
#include <debug/hostio.h>
#include <log.h>
#include <assert.h>

static void outb(uint16_t port, uint8_t value)
{
 asm volatile ("outb %0, %1" : : "a"(value), "Nd"(port));
}

static uint8_t inb(uint16_t port)
{
 uint8_t value;
 asm volatile ("inb %1, %0"
  : "=a"(value)
  : "Nd"(port));
 return value;
}

static void DbgWriteByte(uint8_t value) {
    while ((inb(0x3E8 + 5) & 0x20) == 0) {
  ;
 }
 outb(0x3E8, value);
}

static uint8_t DbgReadByte() {
    while ((inb(0x2F8 + 5) & 1) == 0) {
        ;
    }

    return inb(0x2F8);
}

void DbgWritePacket(int type, uint8_t* data, int size) {
    DbgWriteByte(0xAA);
    DbgWriteByte(type);
    DbgWriteByte((size >> 16) & 0xFF);
    DbgWriteByte((size >> 8) & 0xFF);
    DbgWriteByte((size >> 0) & 0xFF);
    DbgWriteByte(0xBB);

    for (int i = 0; i < size; ++i) {
        DbgWriteByte(data[i]);
    }

    DbgWriteByte(0xCC);
}

void DbgReadPacket(int* type, uint8_t* data, int* size) {
    uint8_t v = DbgReadByte();
    assert(v == 0xAA);

    *type = DbgReadByte();

    int size_tmp = DbgReadByte();
    size_tmp <<= 8;
    size_tmp |= DbgReadByte();
    size_tmp <<= 8;
    size_tmp |= DbgReadByte();

    int max_size = *size;

    *size = size_tmp;

    v = DbgReadByte();
    assert(v == 0xBB);

    for (int i = 0; i < size_tmp; ++i) {
        if (i < max_size) {
            data[i] = DbgReadByte();
        }
    }

    v = DbgReadByte();
    assert(v == 0xCC);
}

#else
extern int make_iso_compilers_happy;
#endif
```

File: ./debug/framework/tfw.c

```

/*
 * tfw - Testing Framework
 */

#include <debug/tfw.h>
#include <debug/hostio.h>
#include <string.h>
#include <log.h>
#include <assert.h>
#include <panic.h>
#include <debug/tfw_tests.h>

#ifndef NDEBUG

#define RESULT_NOT_STARTED  0
#define RESULT_IN_PROGRESS  1
#define RESULT_SUCCESS      2
#define RESULT_FAILURE      3
#define RESULT_SKIPPED      4

struct host_state {
    int test_num;
    uint8_t test_results[MAX_TWF_TESTS];
};

static struct host_state test_state;
static struct tfw_test registered_tests[MAX_TWF_TESTS];
static int num_tests_registered = 0;
static bool nightly_mode = false;

#define PACKET_BUFFER_SIZE (MAX_TWF_TESTS + 256)
uint8_t packet_buffer[PACKET_BUFFER_SIZE];

static bool all_tests_done = false;

/*
 * Send a 0x11 byte to host to ask for current data.
 * Host sends back a packet with 0x22, then the data at data + 8 onwards.
 * It may also send back 0x55, in which case it means it the start of the run (no data yet).
```

```
 */
static void GetHostState() {
    if (all_tests_done) {
        return;
    }

    packet_buffer[0] = 0x11;
    DbgWritePacket(DBGPKT_TFW, packet_buffer, 1);

    while (true) {
        int type;
        int size = PACKET_BUFFER_SIZE - 32;
        DbgReadPacket(&type, packet_buffer, &size);
        if (size < 1024 && type == DBGPKT_TFW) {
            if (packet_buffer[0] == 0x55) {
                test_state.test_num = 0;
                inline_memset(test_state.test_results, 0, sizeof(test_state.test_results));
                break;
            }
            if (packet_buffer[0] == 0x66 || packet_buffer[0] == 0x67) {
                nightly_mode = packet_buffer[0] == 0x67;
                continue;
            }
            assert(packet_buffer[0] == 0x22);
            test_state = *((struct host_state*) (void*) (packet_buffer + 8));
            break;
        }
    }
}

static void ReadAck(void) {
    int type;
    int size = 10;
    uint8_t d[32];
    DbgReadPacket(&type, d, &size);
    assert(d[0] == 0x66 || d[0] == 0x67);
}

/*
 * To set the host state, send a 0x33/0x44, then the data at +8.
 * 0x33 - starting a test
 * 0x34 - finished test successfully
 * 0x35 - finidhed test unsuccessfully
 * 0x44 - finished last test successfully
 * 0x45 - finidhed last test unsuccessfully
 */
static void SetHostState(int code) {
    if (all_tests_done) {
        return;
    }

    assert(sizeof(struct host_state) < 4096 - 8);

    inline_memset(packet_buffer, 0, sizeof(test_state) + 8 + (code == 0x33 ? MAX_NAME_LENGTH : 0));
    packet_buffer[0] = code;
    inline_memcpy(packet_buffer + 8, &test_state, sizeof(test_state));

    if (code == 0x33) {
        strncpy((char*) (packet_buffer + 8 + sizeof(test_state)), registered_tests[test_state.test_num].name, MAX_NAME_LENGTH);
    }
    packet_buffer[1] = registered_tests[test_state.test_num].nightly_only ? 1 : 0;

    DbgWritePacket(DBGPKT_TFW, packet_buffer, sizeof(test_state) + 8 + (code == 0x33 ? MAX_NAME_LENGTH : 0));
    ReadAck();

    if (code >> 4 == 0x4) {
        LogWriteSerial("\n\nFINISHED ALL TESTS:\n");
        for (int i = 0; i < num_tests_registered; ++i) {
            LogWriteSerial("    %s - %s\n", registered_tests[i].name, test_state.test_results[i] == RESULT_SUCCESS ? "passed" : "failed");
        }
        LogWriteSerial("\n");
        all_tests_done = true;
    }
}

static bool in_test = false;

bool IsInTfwTest(void) {
    return in_test;
}

void RegisterTfwTest(const char* name, int start_point, void (*code)(struct tfw_test*, size_t), int expected_panic, size_t context) {
    struct tfw_test test;
    inline_memset(test.name, 0, MAX_NAME_LENGTH);
    strncpy(test.name, name, MAX_NAME_LENGTH - 1);
    test.code = code;
    test.expected_panic_code = expected_panic;
    test.start_point = start_point;
    test.context = context;
    test.nightly_only = false;
    registered_tests[num_tests_registered++] = test;
}

void RegisterNightlyTfwTest(const char* name, int start_point, void (*code)(struct tfw_test*, size_t), int expected_panic, size_t context) {
    RegisterTfwTest(name, start_point, code, expected_panic, context);
    registered_tests[num_tests_registered - 1].nightly_only = true;
}

void FinishedTfwTest(int panic_code) {
    in_test = false;

    bool success = registered_tests[test_state.test_num].expected_panic_code == panic_code;
    LogWriteSerial("FinishedTfwTest: finished test %d, expected %d vs. actual %d\n", test_state.test_num, registered_tests[test_state.test_num].expected_panic_coc

    test_state.test_results[test_state.test_num] = success ? RESULT_SUCCESS : RESULT_FAILURE;

    if (registered_tests[test_state.test_num].nightly_only && !nightly_mode) {
        test_state.test_results[test_state.test_num] = RESULT_SKIPPED;
    }

    test_state.test_num++;
    if (test_state.test_num >= num_tests_registered) {
        SetHostState(success ? 0x44 : 0x45);
    } else {
        SetHostState(success ? 0x34 : 0x35);
    }
}

void MarkTfwStartPoint(int id) {
    LogWriteSerial("Reached TFW_SP %d\n", id);

    if (test_state.test_num >= num_tests_registered || all_tests_done) {
        return;
    }

    if (registered_tests[test_state.test_num].start_point == id) {
        test_state.test_results[test_state.test_num] = RESULT_IN_PROGRESS;
        SetHostState(0x33);
        in_test = true;
        if (!registered_tests[test_state.test_num].nightly_only || nightly_mode) {
```

```
171             LogWriteSerial("MarkTfwStartPoint: running test %d\n", test_state.test_num);
172             registered_tests[test_state.test_num].code(registered_tests + test_state.test_num, registered_tests[test_state.test_num].context);
173         }
174         Panic(PANIC_UNIT_TEST_OK);
175     }
176 }
177
178 void RegisterTfwTests(void) {
179     RegisterTfwInitTests();
180     RegisterTfwWaitTests();
181     RegisterTfwSemaphoreTests();
182     RegisterTfwPhysTests();
183     RegisterTfwAVLTreeTests();
184     RegisterTfwPriorityQueueTests();
185     RegisterTfwIrqlTests();
186 }
187
188 void InitTfw(void) {
189     RegisterTfwTests();
190     ReadAck();
191     GetHostState();
192     test_state.test_results[test_state.test_num] = RESULT_NOT_STARTED;
193 }
194
195 #endif
```

File: ./debug/tests/avl.c

```
1
2  #include <debug.h>
3  #include <assert.h>
4  #include <panic.h>
5  #include <string.h>
6  #include <log.h>
7  #include <avl.h>
8  #include <heap.h>
9  #include <physical.h>
10
11 #ifndef NDEBUG
12
13 /*
14 typedef void (*avl_deletion_handler)(void*);
15 typedef int (*avl_comparator)(void*, void*);
16
17 void* AvlTreeGet(struct avl_tree* tree, void* data);
18 struct avl_node* AvlTreeGetRootNode(struct avl_tree* tree);
19 struct avl_node* AvlTreeGetLeft(struct avl_node* node);
20 struct avl_node* AvlTreeGetRight(struct avl_node* node);
21 void* AvlTreeGetData(struct avl_node* node);
22 avl_deletion_handler AvlTreeSetDeletionHandler(struct avl_tree* tree, avl_deletion_handler handler);
23 avl_comparator AvlTreeSetComparator(struct avl_tree* tree, avl_comparator comparator);*/
24
25 TFW_CREATE_TEST(AVLTreeBasic) { TFW_IGNORE_UNUSED
26     int heap_allocations = DbgGetOutstandingHeapAllocations();
27     struct avl_tree* tree = AvlTreeCreate();
28     assert(AvlTreeSize(tree) == 0);
29     AvlTreeInsert(tree, (void*) 5);
30     assert(AvlTreeSize(tree) == 1);
31     assert(AvlTreeContains(tree, (void*) 5));
32     assert(!AvlTreeContains(tree, (void*) 7));
33     assert(AvlTreeGet(tree, (void*) 5) == (void*) 5);
34     AvlTreeInsert(tree, (void*) 7);
35     AvlTreeInsert(tree, (void*) 9);
36     AvlTreeInsert(tree, (void*) 11);
37     AvlTreeInsert(tree, (void*) 8);
38     AvlTreeInsert(tree, (void*) 6);
39     assert(AvlTreeSize(tree) == 6);
40     AvlTreeInsert(tree, (void*) 4);
41     assert(AvlTreeSize(tree) == 7);
42     AvlTreeDelete(tree, (void*) 5);
43     assert(AvlTreeSize(tree) == 6);
44     assert(AvlTreeContains(tree, (void*) 4));
45     assert(!AvlTreeContains(tree, (void*) 5));
46     assert(AvlTreeContains(tree, (void*) 6));
47     assert(AvlTreeContains(tree, (void*) 7));
48     AvlTreeDestroy(tree);
49
50     /*
51      * Ensure there aren't any memory leaks.
52      */
53     assert(DbgGetOutstandingHeapAllocations() == heap_allocations);
54 }
55
56
57 void RegisterTfwAVLTreeTests(void) {
58     RegisterTfwTest("AVL trees (basic tests)", TFW_SP_AFTER_HEAP, AVLTreeBasic, PANIC_UNIT_TEST_OK, 0);
59 }
60
61 #endif
62
```

File: ./debug/tests/priorityqueue.c

```
1
2  #include <debug.h>
3  #include <assert.h>
4  #include <panic.h>
5  #include <string.h>
6  #include <log.h>
7  #include <arch.h>
8  #include <priorityqueue.h>
9  #include <stdlib.h>
10
11 #ifndef NDEBUG
12
13 static void PQInsertionAndDeletionTest(bool max) {
14     struct priority_queue* queue = PriorityQueueCreate(100, max, 4);
15     assert(PriorityQueueGetCapacity(queue) == 100);
16     int elem;
17     for (int i = 0; i < 100; ++i) {
18         elem = i;
19         PriorityQueueInsert(queue, &elem, i * 3);
20         assert(PriorityQueueGetUsedSize(queue) == i + 1);
21     }
22
23     for (int i = 0; i < 100; ++i) {
24         struct priority_queue_result res = PriorityQueuePeek(queue);
25         int* d = res.data;
26         if (max) {
27             assert(*d == 99 - i);
28             assert((int) res.priority == (99 - i) * 3);
29         } else {
30             assert(*d == i);
31             assert((int) res.priority == i * 3);
```

```
        }
        assert((int) PriorityQueueGetUsedSize(queue) == 100 - i);
        PriorityQueuePop(queue);
        assert((int) PriorityQueueGetUsedSize(queue) == 99 - i);
    }
    PriorityQueueDestroy(queue);
}

TFW_CREATE_TEST(PriorityQueueCombined) { TFW_IGNORE_UNUSED
    PQInsertionAndDeletionTest(true);
    PQInsertionAndDeletionTest(false);
}

TFW_CREATE_TEST(PriorityQueueStress) { TFW_IGNORE_UNUSED
    int expected_size = 0;

    srand(1);

    struct priority_queue* queue = PriorityQueueCreate(100, true, 8);
    assert(PriorityQueueGetCapacity(queue) == 100);

    uint32_t data[2];

    for (int i = 0; i < 1500000; ++i) {
        int rng = rand();
        for (int j = 0; j < 2; ++j) {
            data[j] = rand();
        }

        if (rng % 3 && expected_size < 100) {
            PriorityQueueInsert(queue, data, rng % 10000);
            ++expected_size;

        } else if (rng % 3 == 0 && expected_size > 1) {
            struct priority_queue_result r1 = PriorityQueuePeek(queue);
            PriorityQueuePop(queue);
            struct priority_queue_result r2 = PriorityQueuePeek(queue);
            PriorityQueuePop(queue);
            assert(r1.priority >= r2.priority);
            expected_size -= 2;

        } else {
            --i;
        }

        assert(PriorityQueueGetUsedSize(queue) == expected_size);
    }

    uint64_t prev = 99999999;
    while (PriorityQueueGetUsedSize(queue) > 0) {
        struct priority_queue_result r1 = PriorityQueuePeek(queue);
        PriorityQueuePop(queue);
        assert(r1.priority <= prev);
        prev = r1.priority;
    }

    PriorityQueueDestroy(queue);
}

TFW_CREATE_TEST(PriorityQueueInsertWhenFill) { TFW_IGNORE_UNUSED
    int i = 0;
    struct priority_queue* queue = PriorityQueueCreate(1, true, 4);
    PriorityQueueInsert(queue, &i, 0);
    PriorityQueueInsert(queue, &i, 0);
}


TFW_CREATE_TEST(PriorityQueuePeekWhenEmpty) { TFW_IGNORE_UNUSED
    PriorityQueuePeek(PriorityQueueCreate(1, true, 4));
}

TFW_CREATE_TEST(PriorityQueuePopWhenEmpty) { TFW_IGNORE_UNUSED
    PriorityQueuePop(PriorityQueueCreate(1, true, 4));
}

TFW_CREATE_TEST(PriorityQueueStangeSizes1) { TFW_IGNORE_UNUSED
    PriorityQueueCreate(0, true, 4);
}

TFW_CREATE_TEST(PriorityQueueStangeSizes2) { TFW_IGNORE_UNUSED
    PriorityQueueCreate(-1, true, 4);
}

TFW_CREATE_TEST(PriorityQueueStangeSizes3) { TFW_IGNORE_UNUSED
    PriorityQueueCreate(1, true, 0);
}

TFW_CREATE_TEST(PriorityQueueStangeSizes4) { TFW_IGNORE_UNUSED
    PriorityQueueCreate(1, true, -5);
}

void RegisterTfwPriorityQueueTests(void) {
    RegisterTfwTest("Priority queues (general tests)", TFW_SP_AFTER_HEAP, PriorityQueueCombined, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Priority queues (stress tests)", TFW_SP_AFTER_HEAP, PriorityQueueStress, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Priority queues (insert when full)", TFW_SP_AFTER_HEAP, PriorityQueueInsertWhenFill, PANIC_PRIORITY_QUEUE, 0);
    RegisterTfwTest("Priority queues (peek when empty)", TFW_SP_AFTER_HEAP, PriorityQueuePeekWhenEmpty, PANIC_PRIORITY_QUEUE, 0);
    RegisterTfwTest("Priority queues (pop when empty)", TFW_SP_AFTER_HEAP, PriorityQueuePopWhenEmpty, PANIC_PRIORITY_QUEUE, 0);
    RegisterTfwTest("Priority queues (strange sizes, 1)", TFW_SP_AFTER_HEAP, PriorityQueueStangeSizes1, PANIC_ASSERTION_FAILURE, 0);
    RegisterTfwTest("Priority queues (strange sizes, 2)", TFW_SP_AFTER_HEAP, PriorityQueueStangeSizes2, PANIC_ASSERTION_FAILURE, 0);
    RegisterTfwTest("Priority queues (strange sizes, 3)", TFW_SP_AFTER_HEAP, PriorityQueueStangeSizes3, PANIC_ASSERTION_FAILURE, 0);
    RegisterTfwTest("Priority queues (strange sizes, 4)", TFW_SP_AFTER_HEAP, PriorityQueueStangeSizes4, PANIC_ASSERTION_FAILURE, 0);
}

#endif

```

File: ./debug/tests/init.c

```

#include <debug.h>
#include <assert.h>
#include <panic.h>
#include <string.h>
#include <log.h>
#include <arch.h>
#include <physical.h>

#ifndef NDEBUG

TFW_CREATE_TEST(BootSuccessful) { TFW_IGNORE_UNUSED
}

void RegisterTfwInitTests(void) {
    RegisterTfwTest("Is boot successful", TFW_SP_ALL_CLEAR, BootSuccessful, PANIC_UNIT_TEST_OK, 0);
}

#endif
void RegisterTfwAVLTreeTests(void);
void RegisterTfwPriorityQueueTests(void);
```

File: ./debug/tests/physical.c

```
#include <debug.h>
#include <assert.h>
#include <panic.h>
#include <string.h>
#include <log.h>
#include <arch.h>
#include <physical.h>
#include <stdlib.h>

#ifndef NDEBUG

TFW_CREATE_TEST(IsPageAligned) { TFW_IGNORE_UNUSED
    assert(AllocPhys() % ARCH_PAGE_SIZE == 0);
}

TFW_CREATE_TEST(DeallocationChecksForPageAlignment) { TFW_IGNORE_UNUSED
    size_t p = AllocPhys();
    DeallocPhys(p + context);
}

TFW_CREATE_TEST(DoubleDeallocationFails) { TFW_IGNORE_UNUSED
    size_t p = AllocPhys();
    DeallocPhys(p);
    DeallocPhys(p);
}

TFW_CREATE_TEST(SanityCheck) { TFW_IGNORE_UNUSED
    AllocPhys();
    AllocPhys();
}

TFW_CREATE_TEST(BasicAllocationTest) { TFW_IGNORE_UNUSED
    size_t a = AllocPhys();
    size_t b = AllocPhys();
    size_t c = AllocPhys();
    assert(a != b);
    assert(a != c);
    assert(b != c);
}

TFW_CREATE_TEST(BasicDeallocationTest) { TFW_IGNORE_UNUSED
    size_t a = AllocPhys();
    size_t b = AllocPhys();
    DeallocPhys(a);
    DeallocPhys(b);
    DeallocPhys(AllocPhys());
}

TFW_CREATE_TEST(StressTest) { TFW_IGNORE_UNUSED
    size_t frames[512];
    inline_memset(frames, 0, sizeof(frames));
    int allocated = 0;

    srand(context * 1234 + 12);

    // context 0: 40,000                    (should be around 200ms)
    // context 1: 250,000                   (should be around 1s)
    // context 2: 3,640,000 (NIGHTLY)       (should be around 18s)
    // context 3: 24,070,000 (NIGHTLY)      (should be around 120s)

    int limit = 10000 * (context * context * context * 3 + 2) * (context * context * 3 + 2);
    for (int i = 0; i < limit; ++i) {
        /*
         * This all gets a bit dodgy if it's too much higher than 400 on a 4MB system, as it will
         * eventually need to evict pages, but because we haven't actually mapped any of them into
         * virtual memory, we just hang.
         */
        if (allocated < (300 + (rand() % 80))) {
            size_t f = AllocPhys();
            for (int j = 0; j < 512; ++j) {
                if (frames[j] == f) {
                    Panic(PANIC_MANUALLY_INITIATED);
                }
            }
            for (int j = 0; j < 512; ++j) {
                if (frames[j] == 0) {
                    ++allocated;
                    frames[j] = f;
                    break;
                }
            }
        } else {
            while (allocated > 0) {
                for (int j = 0; j < 512; ++j) {
                    int k = rand() % 256;
                    if (frames[k] != 0) {
                        DeallocPhys(frames[k]);
                        --allocated;
                        frames[k] = 0;
                        break;
                    }
                }
                if (rand() % 17 == 0) {
                    break;
                }
            }
        }
    }
}

TFW_CREATE_TEST(ContiguousAllocationRequiresStackAllocator) { TFW_IGNORE_UNUSED
    assert(AllocPhysContiguous(ARCH_PAGE_SIZE, 0, 0, 0) == 0);
}

void RegisterTfwPhysTests(void) {
    RegisterTfwTest("Is AllocPhys sane", TFW_SP_AFTER_PHYS, SanityCheck, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Basic AllocPhys test (bitmap)", TFW_SP_AFTER_PHYS, BasicAllocationTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Basic AllocPhys test (stack)", TFW_SP_AFTER_PHYS_REINIT, BasicAllocationTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Basic DeallocPhys test (bitmap)", TFW_SP_AFTER_PHYS, BasicDeallocationTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Basic DeallocPhys test (stack)", TFW_SP_AFTER_PHYS_REINIT, BasicDeallocationTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("AllocPhys and DeallocPhys stress test (bitmap 1)", TFW_SP_AFTER_PHYS, StressTest, PANIC_UNIT_TEST_OK, 0);
    RegisterNightlyTfwTest("AllocPhys and DeallocPhys stress test (bitmap 2)", TFW_SP_AFTER_PHYS, StressTest, PANIC_UNIT_TEST_OK, 1);
    RegisterTfwTest("AllocPhys and DeallocPhys stress test (stack 1)", TFW_SP_AFTER_PHYS_REINIT, StressTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("AllocPhys and DeallocPhys stress test (stack 2)", TFW_SP_AFTER_PHYS_REINIT, StressTest, PANIC_UNIT_TEST_OK, 1);
    RegisterNightlyTfwTest("AllocPhys and DeallocPhys stress test (stack 3)", TFW_SP_AFTER_PHYS_REINIT, StressTest, PANIC_UNIT_TEST_OK, 2);
    RegisterNightlyTfwTest("AllocPhys and DeallocPhys stress test (stack 4)", TFW_SP_AFTER_PHYS_REINIT, StressTest, PANIC_UNIT_TEST_OK, 3);
    RegisterTfwTest("AllocPhys returns page aligned addresses", TFW_SP_AFTER_HEAP, IsPageAligned, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeallocPhys only accepts page aligned addresses (1)", TFW_SP_AFTER_HEAP, DeallocationChecksForPageAlignment, PANIC_ASSERTION_FAILURE, 1);
    RegisterTfwTest("DeallocPhys only accepts page aligned addresses (2)", TFW_SP_AFTER_HEAP, DeallocationChecksForPageAlignment, PANIC_ASSERTION_FAILURE, ARCH_PA
    RegisterTfwTest("DeallocPhys checks for double allocation", TFW_SP_AFTER_HEAP, DoubleDeallocationFails, PANIC_ASSERTION_FAILURE, 0);

    RegisterTfwTest("AllocPhysContiguous requires the stack allocator", TFW_SP_AFTER_PHYS, ContiguousAllocationRequiresStackAllocator, PANIC_UNIT_TEST_OK, 0);

    // TODO: contiguous tests...
}

#endif
```

File: ./debug/tests/semaphore.c

```
1
2  #include <semaphore.h>
3  #include <debug.h>
4  #include <assert.h>
5  #include <panic.h>
6  #include <string.h>
7  #include <log.h>
8  #include <arch.h>
9  #include <irql.h>
10 #include <thread.h>
11 #include <errno.h>
12 #include <virtual.h>
13 #include <stdlib.h>
14
15 #ifndef NDEBUG
16
17 static bool Thread1Ok = false;
18 static void Thread1(void* sem_) {
19     struct semaphore* sem = (struct semaphore*) sem_;
20
21     int res = AcquireSemaphore(sem, 1500);
22     Thread1Ok = true;
23     assert(res == ETIMEDOUT);
24
25     while (true) {
26         Schedule();
27     }
28 }
29
30 TFW_CREATE_TEST(SemaphoreTimeout1) { TFW_IGNORE_UNUSED
31     EXACT_IRQL(IRQL_STANDARD);
32
33     struct semaphore* sem = CreateSemaphore("", 1, 0);
34     AcquireSemaphore(sem, -1);
35
36     CreateThread(Thread1, (void*) sem, GetVas(), "");
37
38     SleepMilli(1000);
39     assert(!Thread1Ok);
40     SleepMilli(1000);
41     assert(Thread1Ok);
42 }
43
44 static void Thread2(void* sem_) {
45     struct semaphore* sem = (struct semaphore*) sem_;
46
47     int res = AcquireSemaphore(sem, 15000);
48     Thread1Ok = true;
49     assert(res == 0);
50
51     while (true) {
52         Schedule();
53     }
54 }
55
56 TFW_CREATE_TEST(SemaphoreTimeout2) { TFW_IGNORE_UNUSED
57     EXACT_IRQL(IRQL_STANDARD);
58
59     struct semaphore* sem = CreateSemaphore("", 1, 0);
60     AcquireSemaphore(sem, -1);
61
62     CreateThread(Thread2, (void*) sem, GetVas(), "");
63
64     SleepMilli(1000);
65     ReleaseSemaphore(sem);
66     SleepMilli(500);
67     assert(Thread1Ok);
68 }
69
70 static void Thread3(void* ignored) {
71     (void) ignored;
72
73     while (true) {
74         if (rand() % 10 == 0) {
75             for (int j = 0; j < rand() % 1000000; ++j) {
76                 DbgScreenPrintf("z");
77             }
78         }
79         SleepMilli(rand() % 150);
80     }
81 }
82
83 static void Thread3B(void* ignored) {
84     (void) ignored;
85
86     while (true) {
87         SleepMilli(rand() % 100);
88     }
89 }
90
91 static void Thread4(void* ignored) {
92     (void) ignored;
93
94     while (true) {
95         DbgScreenPrintf("y");
96     }
97 }
98
99 static struct semaphore* sems[20];
100
101 void Thread5(void* delay) {
102     int loops = 0;
103
104     while (true) {
105         int a = rand() % 20;
106         int b = rand() % 8;
107         int c = rand() % 3;
108         if (a == b || a == c || b == c) {
109             continue;
110         }
111
112         int ares = AcquireSemaphore(sems[a], (rand() % 10) * (rand() % 10));
113         int bres = AcquireSemaphore(sems[b], (rand() % 10) * (rand() % 10) * 2);
114         int cres = AcquireSemaphore(sems[c], (rand() % 10) * (rand() % 10) * 3);
115         if (delay != NULL) {
116             SleepMilli(5);
117         }
118         if (ares == 0) {
119             ReleaseSemaphore(sems[a]);
120         }
121         if (cres == 0) {
122             ReleaseSemaphore(sems[c]);
123         }
124         if (bres == 0) {
```

```
            ReleaseSemaphore(sems[b]);
        }
        ++loops;
        if (rand() % 3 == 0) {
            DbgScreenPrintf("%d,", loops);
        }
    }
}

TFW_CREATE_TEST(SchedulerHeartAttack) { TFW_IGNORE_UNUSED
    EXACT_IRQL(IRQL_STANDARD);

    for (int i = 0; i < 20; ++i) {
        sems[i] = CreateSemaphore("", rand() % 3 + 1, 0);
    }

    for (int i = 0; i < 10; ++i) {
        CreateThread(Thread3, NULL, GetVas(), "");
    }
    for (int i = 0; i < 10; ++i) {
        CreateThread(Thread3B, NULL, GetVas(), "");
    }
    for (int i = 0; i < 5; ++i) {
        CreateThread(Thread4, NULL, GetVas(), "");
    }
    for (int i = 0; i < 100; ++i) {
        CreateThread(Thread5, context % 2 == 0 ? NULL : ((void*) 1), GetVas(), "");
    }

    for (int i = 0; i < (int) context; ++i) {
        SleepMilli(1000);
    }
}

void RegisterTfwSemaphoreTests(void) {
    RegisterTfwTest("Semaphores with timeouts can be woken via timeout", TFW_SP_ALL_CLEAR, SemaphoreTimeout1, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Semaphores with timeouts can be woken via release", TFW_SP_ALL_CLEAR, SemaphoreTimeout2, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("Scheduler stress test with semaphores (1)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 3);
    RegisterTfwTest("Scheduler stress test with semaphores (2)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 4);
    RegisterNightlyTfwTest("Scheduler stress test with semaphores (3)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 80);
    RegisterNightlyTfwTest("Scheduler stress test with semaphores (4)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 125);
    RegisterNightlyTfwTest("Scheduler stress test with semaphores (5)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 750);
    RegisterNightlyTfwTest("Scheduler stress test with semaphores (6)", TFW_SP_ALL_CLEAR, SchedulerHeartAttack, PANIC_UNIT_TEST_OK, 755);
}

#endif
```

File: ./debug/tests/irql.c

```
1
2  #include <debug.h>
3  #include <assert.h>
4  #include <panic.h>
5  #include <string.h>
6  #include <log.h>
7  #include <arch.h>
8  #include <irql.h>
9
10 #ifndef NDEBUG
11
12 static int counter = 0;
13
14 static void defer_me(void* context) {
15     counter = (size_t) context;
16 }
17
18 static void defer_me_2(void* context) {
19     counter += (size_t) context;
20 }
21
22 static void defer_me_3(void* context) {
23     int irql = RaiseIrql(IRQL_HIGH);
24     LowerIrql(irql);
25
26     counter += (size_t) context;
27 }
28
29 static void internal_def4_1(void* context) {
30     (void) context;
31
32     assert(counter == 20);
33     counter = 30;
34 }
35
36 static void internal_def4_2(void* context) {
37     (void) context;
38
39     assert(counter == 40);
40     counter = 50;
41 }
42
43 static void defer_me_4(void* context) {
44     (void) context;
45
46     assert(counter == 0);
47
48     int irql = RaiseIrql(IRQL_HIGH);
49     DeferUntilIrql(IRQL_TIMER, internal_def4_1, NULL);
50     DeferUntilIrql(IRQL_PAGE_FAULT, internal_def4_2, NULL);
51     counter = 20;
52     LowerIrql(irql);
53     assert(counter == 30);
54 }
55
56 static void defer_me_5(void* context) {
57     (void) context;
58
59     assert(counter == 30);
60     counter = 40;
61 }
62
63 static void defer_me_6(void* context) {
64     (void) context;
65     int v = (size_t) context;
66     assert(counter == v - 1);
67     counter = v;
68 }
69
70 TFW_CREATE_TEST(RaiseLowerTest) { TFW_IGNORE_UNUSED
71     int irql_a = GetIrql();
72     assert(irql_a == IRQL_STANDARD);
73     int irql_b = RaiseIrql(IRQL_SCHEDULER);
74     assert(irql_a == irql_b);
75     assert(GetIrql() == IRQL_SCHEDULER);
76     int irql_c = RaiseIrql(IRQL_HIGH);
77     assert(irql_c == IRQL_SCHEDULER);
78     assert(GetIrql() == IRQL_HIGH);
```

```
79      LowerIrql(irql_c);
80      assert(GetIrql() == IRQL_SCHEDULER);
81      LowerIrql(irql_a);
82      assert(GetIrql() == irql_a);
83  }
84
85  TFW_CREATE_TEST(DeferRunsImmediatelyAtLevel) { TFW_IGNORE_UNUSED
86      EXACT_IRQL(IRQL_STANDARD);
87      counter = 0;
88      DeferUntilIrql(IRQL_STANDARD, defer_me, (void*) 1);
89      assert(counter == 1);
90      RaiseIrql(IRQL_SCHEDULER);
91      DeferUntilIrql(IRQL_SCHEDULER, defer_me, (void*) 2);
92      assert(counter == 2);
93      RaiseIrql(IRQL_HIGH);
94      DeferUntilIrql(IRQL_HIGH, defer_me, (void*) 3);
95      assert(counter == 3);
96  }
97
98  TFW_CREATE_TEST(DeferDoesntWorkBeforeHeap) { TFW_IGNORE_UNUSED
99      EXACT_IRQL(IRQL_STANDARD);
100     counter = 0;
101     DeferUntilIrql(IRQL_STANDARD, defer_me, (void*) 1);
102     assert(counter == 1);
103     RaiseIrql(IRQL_SCHEDULER);
104     DeferUntilIrql(IRQL_STANDARD, defer_me, (void*) 2);
105     assert(counter == 1);
106     LowerIrql(IRQL_SCHEDULER);
107     assert(counter == 1);
108 }
109
110 TFW_CREATE_TEST(DeferWorksNormally) { TFW_IGNORE_UNUSED
111     EXACT_IRQL(IRQL_STANDARD);
112     counter = 0;
113     DeferUntilIrql(IRQL_STANDARD, defer_me, (void*) 1);
114     assert(counter == 1);
115     RaiseIrql(IRQL_SCHEDULER);
116     DeferUntilIrql(IRQL_STANDARD, defer_me, (void*) 2);
117     assert(counter == 1);
118     LowerIrql(IRQL_STANDARD);
119     assert(counter == 2);
120 }
121
122 TFW_CREATE_TEST(DeferWorksThroughLevelsInOrder) { TFW_IGNORE_UNUSED
123     EXACT_IRQL(IRQL_STANDARD);
124     counter = 0;
125
126     RaiseIrql(IRQL_HIGH);
127     DeferUntilIrql(IRQL_HIGH, defer_me_6, (void*) 1);
128     assert(counter == 1);
129     DeferUntilIrql(IRQL_TIMER, defer_me_6, (void*) 2);
130     assert(counter == 1);
131     DeferUntilIrql(IRQL_DRIVER, defer_me_6, (void*) 3);
132     assert(counter == 1);
133     DeferUntilIrql(IRQL_SCHEDULER, defer_me_6, (void*) 4);
134     assert(counter == 1);
135     DeferUntilIrql(IRQL_PAGE_FAULT, defer_me_6, (void*) 5);
136     assert(counter == 1);
137     DeferUntilIrql(IRQL_STANDARD, defer_me_6, (void*) 6);
138     assert(counter == 1);
139
140     LowerIrql(IRQL_STANDARD);
141     assert(counter == 6);
142 }
143
144
145 TFW_CREATE_TEST(DeferWorksThroughLevels) { TFW_IGNORE_UNUSED
146     EXACT_IRQL(IRQL_STANDARD);
147     counter = 0;
148
149     RaiseIrql(IRQL_HIGH);
150     DeferUntilIrql(IRQL_HIGH, defer_me_2, (void*) 1);
151     assert(counter == 1);
152     DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 2);
153     assert(counter == 1);
154     DeferUntilIrql(IRQL_DRIVER, defer_me_2, (void*) 4);
155     assert(counter == 1);
156     DeferUntilIrql(IRQL_SCHEDULER, defer_me_2, (void*) 8);
157     assert(counter == 1);
158     DeferUntilIrql(IRQL_PAGE_FAULT, defer_me_2, (void*) 16);
159     assert(counter == 1);
160     DeferUntilIrql(IRQL_STANDARD, defer_me_2, (void*) 32);
161     assert(counter == 1);
162
163     LowerIrql(IRQL_STANDARD);
164     assert(counter == 63);
165 }
166
167 TFW_CREATE_TEST(DeferMultipleAtSameLevel) { TFW_IGNORE_UNUSED
168     EXACT_IRQL(IRQL_STANDARD);
169     counter = 0;
170
171     RaiseIrql(IRQL_HIGH);
172     DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 1);
173     assert(counter == 0);
174     DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 2);
175     assert(counter == 0);
176     DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 4);
177     assert(counter == 0);
178
179     LowerIrql(IRQL_STANDARD);
180     assert(counter == 7);
181 }
182
183 TFW_CREATE_TEST(DeferDoesntRunLowHandlers) { TFW_IGNORE_UNUSED
184     EXACT_IRQL(IRQL_STANDARD);
185     counter = 0;
186
187     RaiseIrql(IRQL_HIGH);
188     DeferUntilIrql(IRQL_HIGH, defer_me_2, (void*) 1);
189     assert(counter == 1);
190     DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 2);
191     assert(counter == 1);
192     DeferUntilIrql(IRQL_DRIVER, defer_me_2, (void*) 4);
193     assert(counter == 1);
194     DeferUntilIrql(IRQL_SCHEDULER, defer_me_2, (void*) 8);
195     assert(counter == 1);
196     DeferUntilIrql(IRQL_PAGE_FAULT, defer_me_2, (void*) 16);
197     assert(counter == 1);
198     DeferUntilIrql(IRQL_STANDARD, defer_me_2, (void*) 32);
199     assert(counter == 1);
200
201     LowerIrql(IRQL_SCHEDULER);
202     assert(counter == 15);
203 }
204
205 TFW_CREATE_TEST(DeferWorksThroughLevelsStepping) { TFW_IGNORE_UNUSED
206     EXACT_IRQL(IRQL_STANDARD);
207     counter = 0;
208
```

```
    RaiseIrql(IRQL_HIGH);
    DeferUntilIrql(IRQL_HIGH, defer_me_2, (void*) 1);
    assert(counter == 1);
    DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 2);
    assert(counter == 1);
    DeferUntilIrql(IRQL_TIMER, defer_me_2, (void*) 4);
    assert(counter == 1);
    DeferUntilIrql(IRQL_DRIVER, defer_me_2, (void*) 8);
    assert(counter == 1);
    DeferUntilIrql(IRQL_SCHEDULER, defer_me_2, (void*) 16);
    assert(counter == 1);
    DeferUntilIrql(IRQL_PAGE_FAULT, defer_me_2, (void*) 32);
    assert(counter == 1);
    DeferUntilIrql(IRQL_STANDARD, defer_me_2, (void*) 64);
    assert(counter == 1);

    LowerIrql(IRQL_DRIVER);
    assert(counter == 15);
    RaiseIrql(IRQL_HIGH);
    LowerIrql(IRQL_DRIVER);
    assert(counter == 15);
    LowerIrql(IRQL_PAGE_FAULT);
    assert(counter == 63);
    LowerIrql(IRQL_STANDARD);
    assert(counter == 127);
}

TFW_CREATE_TEST(DeferWithLoweringInHandler) { TFW_IGNORE_UNUSED
    EXACT_IRQL(IRQL_STANDARD);
    counter = 0;

    RaiseIrql(IRQL_SCHEDULER);
    DeferUntilIrql(IRQL_PAGE_FAULT, defer_me_3, (void*) 55);
    assert(counter == 0);
    LowerIrql(IRQL_STANDARD);
    assert(counter == 55);
}

TFW_CREATE_TEST(DeferWithDeferringInHandler) { TFW_IGNORE_UNUSED
    EXACT_IRQL(IRQL_STANDARD);
    counter = 0;

    RaiseIrql(IRQL_HIGH);
    DeferUntilIrql(IRQL_DRIVER, defer_me_4, NULL);
    DeferUntilIrql(IRQL_SCHEDULER, defer_me_5, NULL);
    assert(counter == 0);
    LowerIrql(IRQL_STANDARD);
    assert(counter == 50);
}

void RegisterTfwIrqlTests(void) {
    RegisterTfwTest("RaiseIrql, LowerIrql and GetIrql work", TFW_SP_AFTER_HEAP, RaiseLowerTest, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql gets run immediately at level", TFW_SP_AFTER_HEAP, DeferRunsImmediatelyAtLevel, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql gets run on level lowering", TFW_SP_AFTER_HEAP, DeferWorksNormally, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql defers get ignored before heap", TFW_SP_AFTER_PHYS, DeferDoesntWorkBeforeHeap, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql runs multiple handlers at same level", TFW_SP_AFTER_HEAP, DeferMultipleAtSameLevel, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql runs multiple handlers at different levels (1)", TFW_SP_AFTER_HEAP, DeferWorksThroughLevels, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql runs multiple handlers at different levels (2)", TFW_SP_AFTER_HEAP, DeferWorksThroughLevelsStepping, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql runs multiple handlers in the correct order", TFW_SP_AFTER_HEAP, DeferWorksThroughLevelsInOrder, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql doesn't run handlers below current level", TFW_SP_AFTER_HEAP, DeferDoesntRunLowHandlers, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql when handler calls LowerIrql", TFW_SP_AFTER_HEAP, DeferWithLoweringInHandler, PANIC_UNIT_TEST_OK, 0);
    RegisterTfwTest("DeferUntilIrql when handler calls DeferUntilIrql", TFW_SP_AFTER_HEAP, DeferWithDeferringInHandler, PANIC_UNIT_TEST_OK, 0);
}

#endif
```

File: ./debug/tests/wait.c

```
1
2  #include <semaphore.h>
3  #include <debug.h>
4  #include <assert.h>
5  #include <panic.h>
6  #include <string.h>
7  #include <log.h>
8  #include <arch.h>
9  #include <irql.h>
10 #include <thread.h>
11 #include <process.h>
12 #include <errno.h>
13 #include <virtual.h>
14 #include <stdlib.h>
15
16 #ifndef NDEBUG
17
18 static void SecondProcessThread(void* arg) {
19     int status = (int) (size_t) arg;
20     SleepMilli(1000);
21     KillProcess(status);
22 }
23
24 static void ZombieProcess(void*) {
25     KillProcess(99);
26 }
27
28 static bool ok = false;
29
30 static void InitialProcessThread1(void*) {
31     struct process* child1 = CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 111);
32     SleepMilli(300);
33     struct process* child2 = CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 222);
34     SleepMilli(300);
35     struct process* child3 = CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 333);
36     int retv;
37     WaitProcess(GetPid(child3), &retv, 0);
38     assert(retv == 333);
39     WaitProcess(GetPid(child2), &retv, 0);
40     assert(retv == 222);
41     WaitProcess(GetPid(child1), &retv, 0);
42     assert(retv == 111);
43     ok = true;
44 }
45
46 static void InitialProcessThread2(void*) {
47     pid_t c1pid = GetPid(CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 111));
48     SleepMilli(500);
49     pid_t c2pid = GetPid(CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 222));
50     SleepMilli(500);
51     pid_t c3pid = GetPid(CreateProcessWithEntryPoint(1, SecondProcessThread, (void*) (size_t) 333));
52
53     int retv;
54     pid_t pid = WaitProcess(-1, &retv, 0);
55     assert(retv == 111);
56     assert(pid == c1pid);
57
58     pid = WaitProcess(c3pid, &retv, 0);
```

```
59      assert(retv == 333);
60      assert(pid == c3pid);
61
62      pid = WaitProcess(-1, &retv, 0);
63      assert(retv == 222);
64      assert(pid == c2pid);
65
66      ok = true;
67  }
68
69  static void InitialProcessThread3(void*) {
70      pid_t zombie = GetPid(CreateProcessWithEntryPoint(1, ZombieProcess, NULL));
71      SleepMilli(500);
72
73      int retv;
74      pid_t pid = WaitProcess(-1, &retv, 0);
75      assert(retv == 99);
76      assert(pid == zombie);
77      ok = true;
78  }
79
80  static void InitialProcessThread4(void*) {
81      pid_t zombie = GetPid(CreateProcessWithEntryPoint(1, ZombieProcess, NULL));
82      SleepMilli(500);
83
84      int retv;
85      pid_t pid = WaitProcess(zombie, &retv, 0);
86      assert(retv == 99);
87      assert(pid == zombie);
88      ok = true;
89  }
90
91  static void InitialProcessThread5(void* mode_) {
92      size_t mode = (size_t) mode_;
93
94      pid_t pids[30];
95      for (int i = 0; i < 30; ++i) {
96          pids[i] = GetPid(CreateProcessWithEntryPoint(1, ZombieProcess, NULL));
97      }
98
99      int retv;
100     if (mode == 0) {
101         for (int i = 0; i < 30; ++i) {
102             WaitProcess(-1, &retv, 0);
103             assert(retv == 99);
104         }
105     } else if (mode == 1) {
106         for (int i = 0; i < 30; ++i) {
107             pid_t pid = WaitProcess(pids[i], &retv, 0);
108             assert(retv == 99);
109             assert(pid == pids[i]);
110         }
111     } else {
112         for (int i = 0; i < 30; ++i) {
113             pid_t pid = WaitProcess(pids[29 - i], &retv, 0);
114             assert(retv == 99);
115             assert(pid == pids[29 - i]);
116         }
117     }
118
119     ok = true;
120 }
121
122 TFW_CREATE_TEST(BasicWaitTest) { TFW_IGNORE_UNUSED
123     EXACT_IRQL(IRQL_STANDARD);
124     CreateProcessWithEntryPoint(0, InitialProcessThread1, NULL);
125     SleepMilli(2000);
126     assert(ok);
127 }
128
129 TFW_CREATE_TEST(WaitTestWithNeg1) { TFW_IGNORE_UNUSED
130     EXACT_IRQL(IRQL_STANDARD);
131     CreateProcessWithEntryPoint(0, InitialProcessThread2, NULL);
132     SleepMilli(3000);
133     assert(ok);
134 }
135
136 TFW_CREATE_TEST(WaitOnZombieTest1) { TFW_IGNORE_UNUSED
137     EXACT_IRQL(IRQL_STANDARD);
138     CreateProcessWithEntryPoint(0, InitialProcessThread3, NULL);
139     SleepMilli(1000);
140     assert(ok);
141 }
142
143 TFW_CREATE_TEST(WaitOnZombieTest2) { TFW_IGNORE_UNUSED
144     EXACT_IRQL(IRQL_STANDARD);
145     CreateProcessWithEntryPoint(0, InitialProcessThread4, NULL);
146     SleepMilli(1000);
147     assert(ok);
148 }
149
150 TFW_CREATE_TEST(WaitOnManyTest1) { TFW_IGNORE_UNUSED
151     EXACT_IRQL(IRQL_STANDARD);
152     CreateProcessWithEntryPoint(0, InitialProcessThread5, (void*) (size_t) 0);
153     SleepMilli(6000);
154     assert(ok);
155 }
156
157 TFW_CREATE_TEST(WaitOnManyTest2) { TFW_IGNORE_UNUSED
158     EXACT_IRQL(IRQL_STANDARD);
159     CreateProcessWithEntryPoint(0, InitialProcessThread5, (void*) (size_t) 1);
160     SleepMilli(6000);
161     assert(ok);
162 }
163
164 TFW_CREATE_TEST(WaitOnManyTest3) { TFW_IGNORE_UNUSED
165     EXACT_IRQL(IRQL_STANDARD);
166     CreateProcessWithEntryPoint(0, InitialProcessThread5, (void*) (size_t) 2);
167     SleepMilli(6000);
168     assert(ok);
169 }
170
171
172 void RegisterTfwWaitTests(void) {
173     RegisterTfwTest("WaitProcess works (explict ids)", TFW_SP_ALL_CLEAR, BasicWaitTest, PANIC_UNIT_TEST_OK, 0);
174     RegisterTfwTest("WaitProcess works (-1)", TFW_SP_ALL_CLEAR, WaitTestWithNeg1, PANIC_UNIT_TEST_OK, 0);
175     RegisterTfwTest("WaitProcess allows waiting on zombie (general)", TFW_SP_ALL_CLEAR, WaitOnZombieTest1, PANIC_UNIT_TEST_OK, 0);
176     RegisterTfwTest("WaitProcess allows waiting on zombie (explicit)", TFW_SP_ALL_CLEAR, WaitOnZombieTest2, PANIC_UNIT_TEST_OK, 0);
177     RegisterTfwTest("WaitProcess works when waiting on many (general)", TFW_SP_ALL_CLEAR, WaitOnManyTest1, PANIC_UNIT_TEST_OK, 0);
178     RegisterTfwTest("WaitProcess works when waiting on many (explicit, in order)", TFW_SP_ALL_CLEAR, WaitOnManyTest2, PANIC_UNIT_TEST_OK, 0);
179     RegisterTfwTest("WaitProcess works when waiting on many (explicit, reversed)", TFW_SP_ALL_CLEAR, WaitOnManyTest3, PANIC_UNIT_TEST_OK, 0);
180     // todo: stress tests
181 }
182
183 #endif
```