OS HW02 GROUP 11

311512011 楊士賢

Part 1:Trace Code

1. Explain function

1. threads/thread.cc

- **Overview of class "Thread"**

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int  *stackTop;                       // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

  public:
    Thread(char* debugName, int threadID);    // initialize a Thread
    ~Thread();                                // deallocate a Thread
    // basic thread operations
    void Fork(VoidFunctionPtr func, void *arg); // Make thread run (*func)(arg)
    void Yield();     // Relinquish the CPU if any
                      // other thread is runnable
    void Sleep(bool finishing);   // Put the thread to sleep and
                                  // relinquish the processor
    void Begin();    // Startup code for the thread
    void Finish();   // The thread is done executing

    void CheckOverflow();     // Check if thread stack has overflowed
    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
    char* getName() { return (name); }

    int getID() { return (ID); }
    void Print() { cout << name; }
    void SelfTest();    // test whether thread impl is working

  private:
    // some of the private data for this class is listed above

    int *stack;     // Bottom of the stack
        // NULL if this is the main thread
        // (If NULL, don't deallocate stack)
    ThreadStatus status;  // ready, running or blocked
    char* name;
    int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
           // Allocate a stack for thread.
        // Used internally by Fork()

// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.
    int userRegisters[NumTotalRegs];  // user-level CPU register state
  public:
    void SaveUserState();       // save user-level register state
    void RestoreUserState();    // restore user-level register state

    AddrSpace *space;     // User code this thread is running.
};
```

- Function explanation:

Thread::Thread()

- Set the thread's ID to the provided threadID.
- Set the thread's name to the provided threadName.
- Initialize stackTop and stack to NULL.
- Set the thread's status to JUST_CREATED.
- Loop over the machineState array and set each element to NULL.
-

Thread::~Thread()

- Deallocate resources associated with a thread, primarily its stack.
- Ensure that the current thread cannot delete itself directly and that the main thread does not attempt to delete its automatically obtained stack.

Thread::Fork()

- Acquire pointers to the interrupt and scheduler objects from the kernel.
- Output a debugging message indicating the forking of a thread, including the thread's name, the function pointer, and the argument pointer.
- Allocate a stack for the thread using the StackAllocate function.
- Disable interrupts and save the old interrupt status.
- Use the scheduler to mark the thread as ready to run (assuming interrupts are disabled to avoid race conditions).
- Restore the previous interrupt status.

Thread::Sleep()

- Assert() that the current thread is indeed the kernel's current thread and that interrupts are disabled.
- Output a debugging message indicating that the thread is going to sleep.
- Set the thread's status to BLOCKED.
- Enter a loop:
  - Attempt to find the next thread to run using the scheduler's FindNextToRun function.
  - If no runnable threads are found, invoke the kernel's Idle function, effectively waiting for an interrupt to occur.
  - Exit the loop when a thread becomes available for execution.
- Run the found nextThread using the scheduler's Run function, passing the finishing flag.

2. **userprog/addrspace.cc**

- **class 'AddrSpace'**

```
class AddrSpace {
  public:
    AddrSpace();      // Create an address space.
    ~AddrSpace();     // De-allocate an address space
    static bool usedPhysPage[NumPhysPages]; // record which page has used

    static int usedPhysPageNum; //record the total page has used

    bool Load(char *fileName);    // Load a program into addr space from
                                  // a file
          // return false if not found

    void Execute(char *fileName);               // Run a program
                              // assumes the program has already been loaded
    void SaveState();     // Save/restore address space-specific
    void RestoreState();    // info on a context switch

    // Translate virtual address _vaddr_
    // to physical address _paddr_. _mode_
    // is 0 for Read, 1 for Write.
    ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);

  private:
    TranslationEntry *pageTable;  // Assume linear page table translation
          // for now!
    unsigned int numPages;    // Number of pages in the virtual
          // address space

    void InitRegisters();   // Initialize user-level CPU registers,
          // before jumping to user code

};
```

- **function explanation:**

AddrSpace::AddrSpace() (already include my implementation)

- Allocates memory for a page table using new TranslationEntry[numPages].
- Initializes the page table entries for each virtual page.
- Sets the virtual page number equal to the physical page number for now.
- Finds the first available physical page by iterating through usedPhysPage array.
- Increments j until it finds an unused physical page (usedPhysPage[j++]== false).
- Marks the chosen physical page as occupied (usedPhysPage[j-1] = TRUE).
- Sets the physical page number in the page table entry.
- Sets other attributes in the page table entry:
  - valid: TRUE
  - use: FALSE
  - dirty: FALSE
  - readOnly: FALSE

AddrSpace::~AddrSpace()

- Iterates through each page in the page table (for(int i=0; i<numPages; i++)).
- Clears the corresponding entry in the usedPhysPage array by setting it to FALSE.
- pageTable[i].physicalPage is used to index into usedPhysPage.

- Decreases the total count of used physical pages (usedPhysPageNum) by the number of pages in the address space (usedPhysPageNum -= numPages).
- Deallocates the memory allocated for the page table using delete pageTable.

AddrSpace::Load(char *fileName)

- Opens the specified file (fileName) using the file system.
- Reads the header of the executable file (noffH) to determine its structure.
- Checks if the file was successfully opened; if not, it prints an error message and returns FALSE.
- Checks the file's magic number and performs byte-swapping if necessary.
- Calculates the total size required for the address space, including code, initialized data, uninitialized data, and stack (size).
- Computes the number of pages needed for the address space (numPages) based on the size.
- Checks if the total number of used physical pages plus the number of pages needed is within the allowed limit (ASSERT(usedPhysPageNum + numPages <= NumPhysPages)).
- Increments the total number of used physical pages (usedPhysPageNum) by the number of pages needed for this address space.
- Allocates memory for the page table (pageTable) based on the calculated number of pages.
- Initializes the page table entries, associating virtual pages with physical pages, marking them as occupied in usedPhysPage.
- Copies the code and data segments from the executable file into the corresponding locations in physical memory.
- Closes the executable file (delete executable).
- Returns TRUE to indicate success.

AddrSpce:: Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)

- Declares variables for TranslationEntry pointer (pte), page frame number (pfn), virtual page number (vpn), and offset within a page (offset).
- Calculates the virtual page number (vpn) and offset within a page (offset) based on the provided virtual address (vaddr).
- Checks if the calculated virtual page number is greater than or equal to the total number of pages in the address space (numPages). If true, returns AddressErrorException.
- Retrieves the TranslationEntry corresponding to the virtual page number (vpn) from the page table.
- Checks if the operation is read-write and the page is marked as read-only. If true, returns ReadOnlyException.
- Retrieves the physical page number (pfn) from the TranslationEntry.
- Checks if the physical page number is valid (less than NumPhysPages). If not, returns BusErrorException.
- Sets the "use" bit in the TranslationEntry.
- If the operation is read-write, sets the "dirty" bit in the TranslationEntry.
- Calculates the physical address (*paddr) based on the physical page number and offset.
- Performs an assertion check to ensure the calculated physical address is within

the valid memory size.

- Returns NoException to indicate successful translation.

3. threads/kernel.cc

The Kernel class acts as the central controller for the NachOS operating system. It manages threads, handles interrupts, provides an interface for I/O devices, and orchestrates various components to create a simple and modular operating system environment for educational purposes. The kernel support for thread management, basic file system operations, and interactive testing of various system components… etc.

```
class Kernel {
  public:
    Kernel(int argc, char **argv);
            // Interpret command line arguments
    ~Kernel();            // deallocate the kernel

    void Initialize();    // initialize the kernel -- separated
        // from constructor because
        // refers to "kernel" as a global
  void ExecAll();
  int Exec(char* name);
    void ThreadSelfTest();  // self test of threads and synchronization

    void ConsoleTest();         // interactive console self test
    void NetworkTest();         // interactive 2-machine network test
  Thread* getThread(int threadID){return t[threadID];}

  int CreateFile(char* filename); // fileSystem call

// These are public for notational convenience; really,
// they're global variables used everywhere.

    Thread *currentThread;  // the thread holding the CPU
    Scheduler *scheduler; // the ready list
    Interrupt *interrupt; // interrupt status
    Statistics *stats;    // performance metrics
    Alarm *alarm;   // the software alarm clock
    Machine *machine;           // the simulated CPU
    SynchConsoleInput *synchConsoleIn;
    SynchConsoleOutput *synchConsoleOut;
    SynchDisk *synchDisk;
    FileSystem *fileSystem;
    PostOfficeInput *postOfficeIn;
    PostOfficeOutput *postOfficeOut;

    int hostName;               // machine identifier

  private:

  Thread* t[10];
  char*   execfile[10];
  int execfileNum;
  int threadNum;
    bool randomSlice;   // enable pseudo-random time slicing
    bool debugUserProg;         // single step user program
    double reliability;         // likelihood messages are dropped
    char *consoleIn;            // file to read console input from
    char *consoleOut;           // file to send console output to
#ifndef FILESYS_STUB
    bool formatFlag;          // format the disk if this is true
#endif
};
```

- **function explanation:**

Kernel::Kernek(int argc, char **argv)

- Initializes various flags and parameters based on command line arguments.
- Handles options such as enabling random time slicing, single-step user program debugging, specifying executable files, console input/output files, disk format flag, network reliability, and machine ID.

Kernel::ForkExecute(Thread *t)

- **Function Purpose:**

  The purpose of this function is to load and execute a program associated with a given thread (t).
- **Executable Loading:**

  It checks if the executable associated with the thread (t) can be loaded into its address space using t->space->Load(t->getName()).If the executable is not found or cannot be loaded, the function returns early.
- **Executable Execution:**

  If the executable is successfully loaded, it proceeds to execute the program using t->space->Execute(t->getName()).The Execute function is assumed to be responsible for running the loaded program.

Kernel::ExecAll()

- **Loop Over Executable Files:**

  The function uses a loop to iterate over the array of executable file names (execfile) stored in the execfileNum variable.
- **Execute Each Program:**

  Inside the loop, it calls the Exec function for each executable file, passing the file name as an argument. The result of the execution (return value of Exec) is stored in the variable a.
- **Thread Finishing:**

  After executing all programs specified in the command line arguments, the function calls currentThread->Finish(). This likely finishes the current thread's execution.
- **Assumptions about Exec Function:**

  The details of the Exec function are not provided in this snippet, but it is assumed to be a function responsible for executing a single program. The return value a is stored but not used in the function.
- **Cleanup and Termination:**

  After executing all programs, the current thread is finished. This suggests that the ExecAll function might be used in the context of a multi-threaded environment, where each thread can independently execute a program.

Kernel::Exec(char *name)

- **Thread Creation:**

  It creates a new Thread object (t[threadNum]) with a specified name and thread ID (threadNum).
- **Address Space Initialization:**

  It creates a new address space (AddrSpace) for the thread.
- Thread Forking:

  It forks the newly created thread (t[threadNum]) by calling the Fork method.
- **The Fork method takes two parameters:**

A function pointer to the function that the thread will execute (&ForkExecute). And a pointer to the thread itself ((void *)t[threadNum]).

4. **threads/scheduler.cc**

```
class Scheduler {
  public:
    Scheduler();    // Initialize list of ready threads
    ~Scheduler();   // De-allocate ready list

    void ReadyToRun(Thread* thread);
            // Thread can be dispatched.
    Thread* FindNextToRun();  // Dequeue first thread on the ready
        // list, if any, and return thread.
    void Run(Thread* nextThread, bool finishing);
            // Cause nextThread to start running
    void CheckToBeDestroyed();// Check if thread that had been
            // running needs to be deleted
    void Print();   // Print contents of ready list

    // SelfTest for scheduler is implemented in class Thread

  private:
    List<Thread *> *readyList;  // queue of threads that are ready to run,
        // but not running
    Thread *toBeDestroyed;  // finishing thread to be destroyed
            // by the next thread that runs
};
```

- **function explanation**

  Schheduler::Scheduler()

  - **Ready List Initialization:**

    readyList = new List<Thread *>: Allocates memory for a new instance of the List class (presumably a linked list) to manage the queue of threads that are ready to run.

  - **toBeDestroyed Initialization:**

    toBeDestroyed = NULL: Sets the toBeDestroyed pointer to NULL. This pointer likely points to a thread that is finishing its execution and needs to be destroyed by the next thread that runs.

  Scheduler::ReadyToRun(Thread *thread)

  - **Interrupt Level Assertion:**

    ASSERT(kernel->interrupt->getLevel() == IntOff): Asserts that interrupts are disabled (IntOff). This is a safety check to ensure that the function is called with interrupts disabled.

  - **Thread Status Setting:**

    thread->setStatus(READY): Sets the status of the thread to READY. This indicates that the thread is now ready to be scheduled for execution.

  - **Appending to Ready List:**

    readyList->Append(thread): Appends the thread to the ready list. The ready list typically contains threads that are ready to run but are not currently executing.

  Scheduler::FindNextToRun()

  - **Check for Empty Ready List:**

    if (readyList->IsEmpty()) { return NULL; }: Checks if the ready list is empty. If it is, the

function returns NULL indicating that there are no threads ready to run.

- **Remove and Return Front Thread:**

  return readyList->RemoveFront(): If the ready list is not empty, the function removes and returns the front thread from the ready list. This operation signifies that the thread is chosen to run next.

- **Thread Scheduling Strategy:**

  The function follows a simple strategy of selecting the front thread from the ready list (first-come-first-served (FCFS) or Round-Robin(RR)).

2. Answer question

1. Explain how NachOS creates a thread (process), load it into memory and place it into the scheduling queue.

- **Thread Creation:** NachOS creates a thread using a system call or a dedicated function. Initializes a Thread Control Block (TCB) with relevant information. Allocates necessary resources, including a stack for the thread.
- **Loading into Memory:** Reads the executable file (e.g., COFF format) containing the program's code and data. Loads the instructions and data into the thread's virtual memory space. Sets up initial values of registers and program counter.
- **Placement into Scheduling Queue:** Thread is added to a scheduling queue (e.g., ready queue). Scheduling algorithm determines the order in which threads will run. Various scheduling policies (e.g., FCFS, Round Robin) may be used.

2. How does Nachos allocate the memory space for a new thread(process)?

- **Virtual Memory Management:**

  Nachos uses virtual memory management to provide an abstraction of the underlying physical memory. Each thread is given its own virtual address space.

- **Address Space Initialization:**

  When a new thread is created, Nachos initializes a data structure called the address space. The address space defines the virtual memory layout for the thread, including code, data, heap, and stack segments.

- **Page Table Setup:**

  Nachos maintains a page table for each thread, mapping virtual addresses to physical addresses. The page table is used by the hardware and operating system to translate virtual addresses used by the thread into actual physical memory addresses.

- **Memory Initialization:**

  After the virtual memory space is set up, Nachos may initialize it with appropriate values. For example, it may set the initial values of the program counter, registers, and other relevant state information.

3. How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?

- **Loading the User Binary:**

  Nachos needs to load the binary executable of the user program into the memory space of the new thread. This binary contains the machine code instructions that the processor will execute.

- **Segmentation of Memory:**

  The memory space assigned to the thread is often divided into segments, such as the code segment, data segment, heap, and stack. These segments correspond to different parts of the user program's memory layout.

- **Copying Binary Code to Memory:**
  The code segment of the user binary is copied into the designated region of the thread's virtual memory space. This involves reading the binary code from the executable file on disk and writing it to the appropriate location in the thread's virtual memory.
- **Setting Initial Program Counter (PC):**
  The initial program counter (PC) is set to the starting address of the code segment. This is the address where the execution of the user program will begin.
- **Setting Up Stack:**
  The stack segment is initialized, and the stack pointer (SP) is set to the initial top of the stack. The stack will be used for managing function calls, local variables, and other runtime data.
- **Initializing Data Segment:**
  The data segment may contain initialized data for global and static variables. This segment is initialized with the appropriate values from the binary file.
- **Initializing Heap:**
  If applicable, the heap segment is initialized. This is the region of memory used for dynamic memory allocation during the program's execution.
- **Setting Up Page Table Entries:**
  Nachos maintains a page table for each thread, mapping virtual addresses to physical addresses. The page table is set up to reflect the segmentation and layout of the thread's virtual memory.
- **Setting Register Values:**
  The initial values of registers (including the program counter and stack pointer) are set to appropriate values for the start of execution.
- **Marking Thread as Ready:**
  Finally, the thread is marked as "ready" to execute. It can now be placed into the scheduling queue to await its turn for execution.

4. How does Nachos create and manage the page table?

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[numPages]; // create an enthough big pagetable
    for (unsigned int i = 0, j = 0; i < numPages; i++){

        pageTable[i].virtualPage = i; // for now, virt page # = phys page #

        while(j < NumPhysPages && usedPhysPage[j++]){}
        // It's means this page is occupy by other threads
        // so j++ to find next page

        usedPhysPage[j-1]=TRUE; //turn it to occupy

        pageTable[i].physicalPage = j-1;
        // pageTable[i].physicalPage = i;

        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
```

- **Page Table Creation:**
  When a new thread is created in Nachos, part of the initialization process involves creating a page table specific to that thread. The page table is typically implemented as an

array or another suitable data structure.

- **Mapping Virtual to Physical Addresses:**
  The page table contains entries that map virtual page numbers to corresponding physical page numbers. Each entry in the page table corresponds to a page of virtual memory. When a thread accesses a virtual address, the page table is consulted to determine the corresponding physical address.
- **Initialization with Invalid Entries:**
  Initially, the page table entries are often marked as "invalid" or "unmapped." This means that the virtual pages are not yet mapped to any physical pages. This state allows Nachos to handle page faults efficiently.
- **Handling Page Faults:**
  When a thread tries to access a virtual page that is not currently in physical memory (a page fault), Nachos needs to handle this situation. It may trigger a page fault exception, and the Nachos kernel's page fault handler is responsible for loading the required page into physical memory and updating the page table.
- **Updating Page Table on Page Fault**:
  The page table is updated during a page fault to reflect the new mapping of virtual pages to physical pages. The entry for the accessed virtual page is modified to contain the appropriate physical page number.
- **Maintaining Protection and Permissions:**
  The page table is used to enforce memory protection and permissions. Entries in the page table can include information about whether a page is read-only, read-write, executable, etc. The Nachos kernel checks these permissions when a thread attempts to access memory.
- **Sharing Memory:**
  In some cases, threads might share portions of memory. The page table can be configured to allow multiple threads to map the same physical page, enabling shared memory.
- **Destruction of Page Table:**
  When a thread is terminated, its page table is typically destroyed to release the associated resources.

5. How does Nachos translate addresses?

```
ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    int i;
    unsigned int vpn, offset;
    TranslationEntry *entry;
    unsigned int pageFrame;

    DEBUG(dbgAddr, "\tTranslate " << virtAddr << (writing ? " , write" : " , read"));

// check for alignment errors
    if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){
    DEBUG(dbgAddr, "Alignment problem at " << virtAddr << ", size " << size);
    return AddressErrorException;
    }
    // we must have either a TLB or a page table, but not both!
    ASSERT(tlb == NULL || pageTable == NULL);
    ASSERT(tlb != NULL || pageTable != NULL);

// calculate the virtual page number, and offset within the page,
// from the virtual address
```

```
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    if (tlb == NULL) {      // => page table => vpn is index into table
    if (vpn >= pageTableSize) {
        DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
        return AddressErrorException;
    } else if (!pageTable[vpn].valid) {
        DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
        return PageFaultException;
    }
    entry = &pageTable[vpn];
    } else {
        for (entry = NULL, i = 0; i < TLBSize; i++)
            if (tlb[i].valid && (tlb[i].virtualPage == ((int)vpn))) {
        entry = &tlb[i];            // FOUND!
        break;
        }
    if (entry == NULL) {                // not found
            DEBUG(dbgAddr, "Invalid TLB entry for this virtual page!");
            return PageFaultException;      // really, this is a TLB fault,
                        // the page may be in memory,
                        // but not in the TLB
    }
    }

    if (entry->readOnly && writing) {   // trying to write to a read-only page
    DEBUG(dbgAddr, "Write to read-only page at " << virtAddr);
    return ReadOnlyException;
    }
    pageFrame = entry->physicalPage;

    // if the pageFrame is too big, there is something really wrong!
    // An invalid translation was loaded into the page table or TLB.
    if (pageFrame >= NumPhysPages) {
    DEBUG(dbgAddr, "Illegal pageframe " << pageFrame);
    return BusErrorException;
    }
    entry->use = TRUE;      // set the use, dirty bits
    if (writing)
    entry->dirty = TRUE;
    *physAddr = pageFrame * PageSize + offset;
    ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
    DEBUG(dbgAddr, "phys addr = " << *physAddr);
    return NoException;
}
```

- **Virtual Addresses:**
  Threads in Nachos use virtual addresses when accessing memory.
- **Page Table Lookup:**
  The page table is consulted to map virtual page numbers to physical page numbers.
- **Page Number Extraction:**
  Virtual addresses are split into a virtual page number and an offset within the page.
- **Page Table Indexing:**
  The VPN is used as an index to find the corresponding entry in the page table.
- **Calculating Physical Address:**
  The physical page number from the page table entry, combined with the offset, gives the actual physical address in memory.
- **Accessing Physical Memory:**
  The calculated physical address is used to read or modify data in physical memory.

- **Handling Page Faults:**
  If needed, Nachos loads the required page from secondary storage during a page fault.

6. **How Nachos initializes the machine status (register, etc) before running a thread (process)**
   - **Processor Registers:**
     Nachos initializes the general-purpose registers, program counter (PC), stack pointer (SP), and other processor registers to appropriate initial values. These values are often based on the context of the thread's execution.
   - **Program Counter (PC):**
     The program counter is set to the starting address of the thread's code segment. This is the address where the execution of the thread's instructions will begin.
   - **Stack Pointer (SP):**
     The stack pointer is set to the initial top of the thread's stack. The stack is used for managing function calls, local variables, and other runtime data.
   - **Other Processor State:**
     Nachos may set other processor state information, such as the status register, interrupt enable/disable status, and any other relevant control registers.
   - **Machine-Specific State:**
     If there is machine-specific state information stored in the thread's data structure (e.g., in the machineState array), Nachos initializes this state as needed.
   - **Memory Management Registers:**
     If applicable, Nachos sets up memory management registers, such as base and bounds registers, to establish the thread's memory boundaries.

7. **Which object in Nachos acts the role of process control block**
   - **Thread object:**

   The Thread class in Nachos encapsulates the necessary information and state associated with a thread or process. It includes attributes such as the thread's ID, name, status, stack information, program counter, register values. The Thread class, therefore, serves as the equivalent of a process control block, providing a data structure to manage and represent the state of individual threads within the operating system.

Part 2:Implementation

1. Detail of your implementation

- Due to the built-in RR schedule of Nachos (since RR is used for non-multi-programming and its effectiveness is similar to FCFS), modifying addrspace.cc and addrspace.h to support multi-programming will enable smooth execution of RR.
- Since Nachos' default settings allocate all physical memory pages, running 2 or more threads will result in complete duplication of pages assigned to each, causing errors during execution.
- Therefore, the modifications needed are:

  Add **static bool usedPhysPage[NumPhysPages];** to record which physical pages have already been used.

  Add **static int usedPhysPageNum;** to record the total number of used physical pages.

  Modify AddrSpace() to allocate only the necessary number of physical pages for a thread, rather than allocating all at once.

  Modify ~AddrSpace() to reset the contents of the used static variables.

  Modify bool AddrSpace::Load(char *fileName)

Addrspace.h

Add variable `usedPhysPage` and `usedPhysPageNum`

```
    //Initialize page
bool AddrSpace::usedPhysPage[NumPhysPages];
int AddrSpace::usedPhysPageNum=0;
```

usedPhysPage: used to store the information of which pages are used or not.
usedPageNum: used to save how many pages are being used currently.

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[numPages]; // create an enthough big pagetable
    for (unsigned int i = 0, j = 0; i < numPages; i++){

        pageTable[i].virtualPage = i; // for now, virt page # = phys page #

        while(j < NumPhysPages && usedPhysPage[j++]){} //If usedPhysPage[j++] is True
        // It's means this page is occupy by other threads
        // so j++ to find next page

        usedPhysPage[j-1]=TRUE; //turn it to occupy

        pageTable[i].physicalPage = j-1;
        // pageTable[i].physicalPage = i;

        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

    // pageTable = new TranslationEntry[NumPhysPages];
    // for (int i = 0; i < NumPhysPages; i++) {
    // pageTable[i].virtualPage = i;    // for now, virt page # = phys page #
    // pageTable[i].physicalPage = i;
    // pageTable[i].valid = TRUE;
    // pageTable[i].use = FALSE;
    // pageTable[i].dirty = FALSE;
    // pageTable[i].readOnly = FALSE;
    // }

    // // zero out the entire address space
    // bzero(kernel->machine->mainMemory, MemorySize);
}
```

At **AddrApace::AddrSpace()** constructor we comment out the previous code because we only

need specific size of address space size for multi-programing

We iterate through the usedPhysPage array, increments the index j until it finds a page that is not yet used (i.e., FALSE in usedPhysPage). Once it finds an unused page, it sets its value to TRUE and proceeds to store data on that page.

```
//----------------------------------------------------------------------
// AddrSpace::~AddrSpace
//  Dealloate an address space.
//----------------------------------------------------------------------

AddrSpace::~AddrSpace()
{
    for(int i=0;i<numPages;i++){
        usedPhysPage[pageTable[i].physicalPage] = FALSE; //clear the page table to unused state
    }
    usedPhysPageNum-=numPages;
    delete pageTable;
}
```

At deconstructor **AddrSpace::~AddrSpace()** we set every used page to false to determine the state of pages are not being used now. Also delete the Table.

```
//----------------------------------------------------------------------
// AddrSpace::Load
//  Load a user program into memory from a file.
//
//  Assumes that the page table has been initialized, and that
//  the object code file is in NOFF format.
//
//  "fileName" is the file containing the object code to load into memory
//----------------------------------------------------------------------

bool
AddrSpace::Load(char *fileName)
{
OpenFile *executable = kernel->fileSystem->Open(fileName);
NoffHeader noffH;
unsigned int size;

if (executable == NULL) {
cerr << "Unable to open file " << fileName << "\n";
return FALSE;
}
```

```
executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
if ((noffH.noffMagic != NOFFMAGIC) &&
(WordToHost(noffH.noffMagic) == NOFFMAGIC))
SwapHeader(&noffH);
ASSERT(noffH.noffMagic == NOFFMAGIC);

// how big is address space?
size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
+ UserStackSize; // we need to increase the size
// to leave room for the stack
numPages = divRoundUp(size, PageSize);
// cout << "number of pages of " << fileName<< " is "<<numPages<<endl;

size = numPages * PageSize;

ASSERT(usedPhysPageNum+numPages <= NumPhysPages);
//numpage is the page size of this threads
//now we're not trying vm so used page + numpage must smaller then total page
// check we're not trying
// to run anything too big --
// at least until we have
// virtual memory
usedPhysPageNum+=numPages;

//----------------------------------------------------
//
pageTable = new TranslationEntry[numPages]; // create an enthough big pagetable
for (unsigned int i = 0, j = 0; i < numPages; i++)
{
pageTable[i].virtualPage = i; // for now, virt page # = phys page #

while(usedPhysPage[j++]){} //If usedPhysPage[j++] is True
// It's means this page is occupy by other threads
// so j++ to find next page

usedPhysPage[j-1]=TRUE; //turn it to occupy

pageTable[i].physicalPage = j-1;
// pageTable[i].physicalPage = i;

pageTable[i].valid = TRUE;
pageTable[i].use = FALSE;
pageTable[i].dirty = FALSE;
pageTable[i].readOnly = FALSE;
}
```

```
//----------------------------------------------------

DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

// then, copy in the code and data segments into memory
if (noffH.code.size > 0) {
DEBUG(dbgAddr, "Initializing code segment.");
DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
/*
executable->ReadAt(
&(kernel->machine->mainMemory[noffH.code.virtualAddr]),
noffH.code.size, noffH.code.inFileAddr);
*/
executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage*Page
Size+(noffH.code.virtualAddr%PageSize)]),
noffH.code.size, noffH.code.inFileAddr);

}
if (noffH.initData.size > 0) {
DEBUG(dbgAddr, "Initializing data segment.");
DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
/*
executable->ReadAt(
&(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
noffH.initData.size, noffH.initData.inFileAddr);
*/
executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage*
PageSize+(noffH.initData.virtualAddr%PageSize)]),
noffH.initData.size, noffH.initData.inFileAddr);

}

delete executable; // close file
return TRUE; // success
}
```

At **load()** function we fix the readAt() function to construct the mapping relationship of virtual address to physical address, we are trying to load the program into memory for execution:

Load code into memory

```
/*
executable->ReadAt(
&(kernel->machine->mainMemory[noffH.code.virtualAddr]),
```

```
noffH.code.size, noffH.code.inFileAddr);
*/
executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage*
PageSize+(noffH.code.virtualAddr%PageSize)]),
noffH.code.size, noffH.code.inFileAddr);
```

load program data into memory

```
/*
executable->ReadAt(
&(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
noffH.initData.size, noffH.initData.inFileAddr);
*/
executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalP
age*PageSize+(noffH.initData.virtualAddr%PageSize)]),
noffH.initData.size, noffH.initData.inFileAddr);
```

result:

without multi-prgraming

```
../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
16
15
18
19
1return value:0
7
return value:0
```

With multi-programing

```
9
8
7
6
1consoleIO_test1
consoleIO_test2
return value:0
5
16
17
18
19
return value:0
done
OK (1.030 sec real, 1.031 sec wall)
Triggering a new build of os_group11_ta
Finished: SUCCESS
```

Part 3:Contribution

1. Describe details and percentage of each member's contribution.

100%楊士賢