NACHOS OVERVIEW

CIS657 Principles of Operating Systems

NACHOS

- Not Another Completely Heuristic Operating System
- A watered-down version of Unix
- You have complete access to source code
- Runs as a Unix process: easy to debug
- It’s a real OS, although a clipped one.

OUTLINE

1. Installing and Building NachOS
2. NachOS Directory and File Structure
3. NachOS Architecture
4. Additional information
    1. Simulated Components of MIPS
    2. Files related to Threads
    3. Debugging Tips
    4. System Utilities
    5. Multiprogramming & Virtual Memory in Nachos

INSTALLING AND BUILDING NACHOS

BUILDING NACHOS

1. Go to the build directory:

cd nachos/code/build.linux

2. Build NachOS:

First: make clean

Second: make depend

Third: make

3. Use **ls** command to check the executable file **nachos** (if you find, the compilation is successful)

4. Run NachOS:

./nachos –K

- You should see the output loop.

NACHOS DIRECTORY AND FILE STRUCTURE

NACHOS DIRECTORY AND FILE STRUCTURE

- code/threads: Heart of the kernel -- scheduler, synch primitives, etc.
- code/filesys: Filesystem
- code/lib: Library routines
- code/machine: MIPS simulator and simulated hardware
- code/network: Networking
- code/test: test user programs in C. Need a MIPS cross-compiler
- code/userprog: Support for user-level processes

NACHOS ARCHITECTURE

NACHOS ARCHITECTURE (1)

- An instance of Nachos is a regular Unix process
- Processes (Processes = “Threads” in Nachos) on Nachos are run by the MIPS emulator in code/machine/mipssim.cc and others
- “Kernel-level processes (Nachos threads)” refer to processes running under Nachos OS
- “User-level processes (Nachos threads)” refer to processes running on Nachos. These programs are compiled in the code/test directory
- Each user-level process has a corresponding kernel-level process
- This means the user-level processes in Nachos has two-sets of registers and stacks (one under kernel-space and the other under user-level)

NACHOS ARCHITECTURE (2)

- MIPS Simulator runs as a main event loop, invoked with Machine::Run
- Kernel code gets called from the simulator through (simulated) exceptions and interrupts
- Interrupts cause the simulator to call the appropriate interrupt handler
- Exceptions and System Calls cause the simulator to call the interrupt handler (userprog/exception.cc)
- Returning from the interrupt handler or exception handler returns control to the simulator

ADDITIONAL INFORMATION

SIMULATED COMPONENTS OF MIPS

Simulation code in code/machine

TIMER

- Acts like a periodic alarm clock
- Runs in simulation time
- timer.[cc|h], interrupt.[cc|h]

DISK

- One recording surface
  - divided into tracks, and each track is divided into sectors
- simulator stores the contents of the simulated disk in a Unix file called DISK_X, where X is the identifier of the simulated machine
- disk.[cc|h]

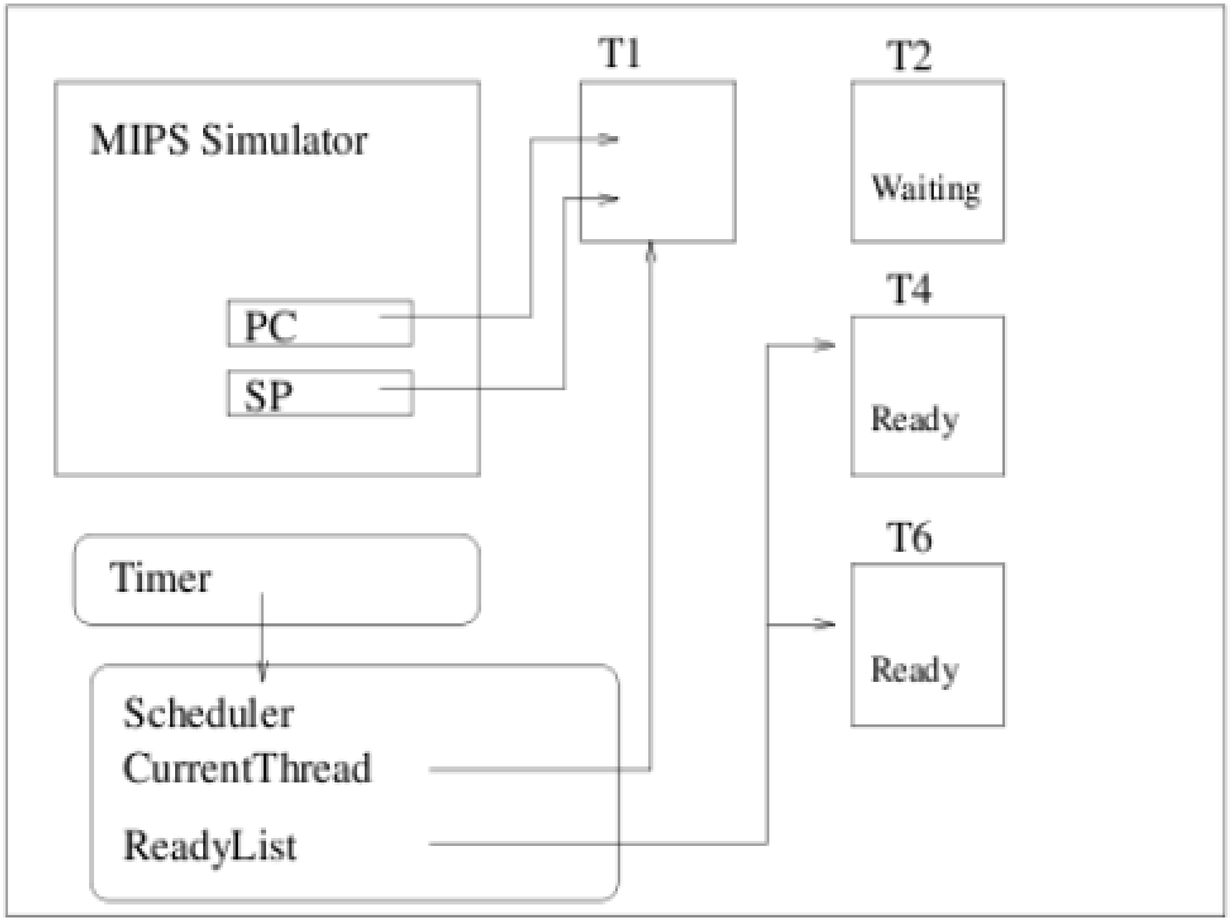
MIPS INSTRUCTION PROCESSOR

- responsible for simulating the execution of user programs
- a set of registers and a small (simulated) physical memory for storing user programs and their data
- translating virtual memory addresses to physical addresses
- When the user program executes a system call instruction, an exception is generated
- machine.[cc|h], mipssim.[cc|h], and translate.[cc|h]

INTERRUPTS

- simulate the interrupt and exception mechanism
- Interrupt.[cc|h], exception.cc

FILES RELATED TO THREADS


MIPS Simulator
PC
SP
T1
T2
Waiting
T4
Ready
T6
Ready
Timer
Scheduler
CurrentThread
ReadyList

FILES RELATED TO THREADS

- Main.cc
- System.cc
- Thread.[cc|h]
- Threadtest.cc
- Scheduler.[cc|h]
- Switch.[cc|h]

MAIN.CC

- Nachos main entry
- Various command line flags.
  - You should get familiar to these
  - Let’s see the source (You can download from the Blackboard (located under content menu)).

SYSTEM.CC

- Initializes most of the system stuff
- H/W devices
- Creates objects

THREAD.[CC | H]

```
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
  int machineState[MachineStateSize];  // all registers except for stackTop

 public:
  Thread(char* debugName);          // initialize a Thread
  ~Thread();                        // deallocate a Thread
                                    // NOTE -- thread being deleted
                                    // must not be running when delete
                                    // is called

  // basic thread operations
```

THREAD.H (CONTINUED)

```
  void Fork(VoidFunctionPtr func, int arg);  // Make thread run (*func)(arg)
  void Yield();                              // Relinquish the CPU if any
                                             // other thread is runnable
  void Sleep();                              // Put the thread to sleep and
                                             // relinquish the processor
  void Finish();                             // The thread is done executing

  void CheckOverflow();                      // Check if thread has
                                             // overflowed its stack
  void setStatus(ThreadStatus st) { status = st; }
  char* getName() { return (name); }
  void Print() { printf("%s, ", name); }
   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.
```

THREADTEST.CC

- Contains procedure SimpleThread(int which) and ThreadTest()

```
void
SimpleThread(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();
    }
}
```

THREADTEST.CC

- ThreadTest(): called by main()

```
void
ThreadTest()
{
    DEBUG('t', "Entering SimpleTest.\n");

    Thread *t = new Thread("forked thread");

    t->Fork(SimpleThread, 1);
    SimpleThread(0);
}
```

WHY IN NACHOS, THEY ARE CALLED "THREADS?"

- From the Unix's point of view, these are threads. In fact, these are threads implemented in the user-level. Therefore, the kernel is not aware of them.
- From the Nachos' point of view, these are really heavy weight processes.
- Q: Can we implement a thread system for Nachos?

  A: Sure.

DEBUGGING TIPS

DEBUGGING TIPS

- Most segmentation faults and errors are the result of memory allocation problems
- Be aware of the Memory Model

INTERNAL DEBUGGING IN NACHOS

- Trace Facility with **–d** flag
  - Example: nachos –d td
    - Print debugging messages related to threads(“t”) and the disk (“d”) emulation
- Single Stepping by **–s** command-line argument
- Assertions
  - assertion is tested and if it is found to be false, the program is terminated and a message is printed indicating which assertion failed

GDB CAPABILITIES

- run a program
- stop it on any line or the start of a function
- examine various types of information like values of variables, sequence of function calls
- stop execution when a variable changes
- change values of variables (during execution)
- call a function at any point in execution

EXTERNAL DEBUGGER (GDB)

- All of your Nachos code should be compiled with the -g option to g++ (Makefile)
- You are in the same directory as the NachOS executable (nachos), start gdb nachos
  - Run NachOS under control of GDB
- GDB's prompt, enter a GDB command
  - break (halt execution at a specified function or line number)
  - run (pass any parameter to nachos)
  - list (print out source code between lines xxx and yyy)
  - next (execute one line at a time, step over)
  - step (execute one line at a time, step into)
  - print (display the values of program variables)
  - continue (with breakpoint, loop)
  - bt (produce a stack backtrace)

SYSTEM UTILITIES ON UNIX

EDITORS

- VIM
  - Based on vi, standard on UNIX machines
  - Smaller and faster editor
  - Start a session on Terminal
    - vim source.c
  - Press ‘i’ to enter insert mode
  - Press ‘Esc’ to enter command mode
    - ‘:wq’ => save and quit the session
  - https://www.engadget.com/2012/07/10/vim-how-to/

USEFUL UNIX UTILITIES

- grep – search for strings in a file
- find – find a particular file
- du – determine the disk usage of files and directories
- ls – list files and their permissions
- man grep (manual)
- mkdir – create a directory
- pwd – show the current directory that you are in
- cp – copy a file
- mv – move and/or rename file(s)
- rm – delete a file from the filesystem
- rmdir – delete empty directories
- chmod – change permissions of files

MULTIPROGRAMMING & VIRTUAL MEMORY IN NACHOS

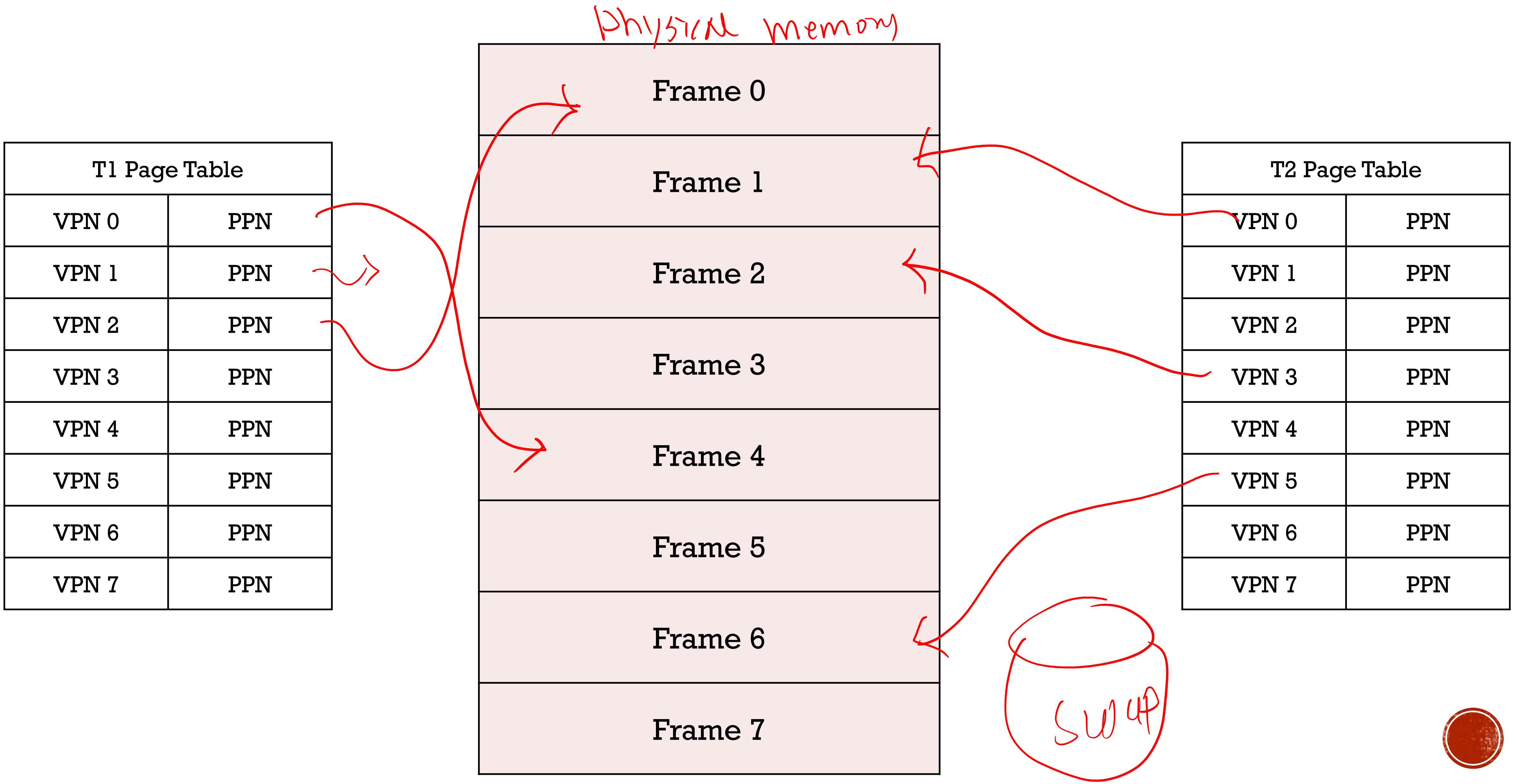
MAIN MEMORY IN NACHOS

- Implemented as an array of bytes
- Byte-addressable, 128-byte frames (physical pages)
  - 128-byte frame == size of disk sector
  - PageSize = 128
- 128 frames of main memory (default)
  - NumPhysPages = 128
  - MemorySize = 128 * 128 = 16 KB
- TLBSize = 4 if used
- See machine/machine.h
- Address space for each Nachos process is 128 KB

MEMORY MANAGEMENT IN NACHOS

- Address space for each Nachos thread is 128 KB
  - Contiguous logical address space
  - AddrSpace object per Nachos thread
    - page table
- Virtual memory is used to minimize the number of physical pages used at any given time
- Swap in/out
- Need to keep track of which physical pages are free or used, and the owners (threads) of each page

NON-CONTIGUOUS MEMORY ALLOCATION EXAMPLE IN NACHOS



T1 Page Table	
VPN 0	PPN
VPN 1	PPN
VPN 2	PPN
VPN 3	PPN
VPN 4	PPN
VPN 5	PPN
VPN 6	PPN
VPN 7	PPN

T2 Page Table	
VPN 0	PPN
VPN 1	PPN
VPN 2	PPN
VPN 3	PPN
VPN 4	PPN
VPN 5	PPN
VPN 6	PPN
VPN 7	PPN

TRANSLATION ENTRIES

- Each Nachos process' page table: array of TranslationEntry object
  - See /machine/translate.h

```
class TranslationEntry {
  public:
    int virtualPage;        // The page number in virtual memory.
    int physicalPage;       // The page number in real memory (relative to the
                            //  start of "mainMemory"
    bool valid;         // If this bit is set, the translation is ignored.
                            // (In other words, the entry hasn't been initialized.)
    bool readOnly; // If this bit is set, the user program is not allowed
                            // to modify the contents of the page.
    bool use;           // This bit is set by the hardware every time the
                            // page is referenced or modified.
    bool dirty;         // This bit is set by the hardware every time the
                            // page is modified.
};
```

DOING TRANSLATION/PAGE FAULT HANDLING

- Fetching/Executing next_inst = read_mem(PC, …)

  read_mem(…) // machine/translate.cc

  Translate() : logical addr -> physical addr

  RaiseException() // machine/machine.cc

  ExceptionHandler() // userprog/exception.cc

- Nachos supports either linear page table or software managed TLB, not both

DEMAND PAGING/PAGE REPLACEMENT

- demand paging using page faults to dynamically load process virtual pages on demand, rather than initializing page frames for each process in advance
- page replacement enabling the kernel to evict any virtual page from memory in order to free up a physical page frame to satisfy a page fault
  - Three special bits in each PTE
    - valid bit to tell the machine which virtual pages are resident in memory (a valid translation) and which are not resident (an invalid translation)
      - If a user process references an address for which the PTE is marked invalid, then the machine raises a page fault exception and transfers control to the kernel's exception handler
    - use bit (reference bit) to pass information to the kernel about page access patterns. If a virtual page is referenced by a process, the machine sets the corresponding PTE reference bit to inform the kernel that the page is active. Once set, the reference bit remains set until the kernel clears it.
    - dirty bit in the PTE whenever a process executes a store (write) to the corresponding virtual page. This informs the kernel that the page is dirty; if the kernel evicts the page from memory, then it must first "clean" the page by preserving its contents on disk. Once set, the dirty bit remains set until the kernel clears it.

SWAPPING: PAGE IN/OUT

- When a page fault occurs, take the correct page into main memory (physical page)
- If there is no free page in the physical memory, Nachos must remove a page from physical memory to create a space for the faulted page
  - Swapped out
  - Select a victim page

MULTIPLE USER PROGRAMS

- Up until now, you have executed all threads within the Nachos kernel

- Nachos runs user programs in their own private address space
  - MIPS binary – so we use MIPS cross-compiler
  - Check Makefile in test directory

- Nachos creates an address space and copies the contents of the instruction and initialized variable segments into the address space

- Now execute user programs to invoke kernel routines via system calls
  - You need to modify Nachos to support multiple user processes (Nachos threads) using system calls to request services from the kernel
  - Trap into the Nachos Kernel

SYSTEM CALLS AND EXCEPTION HANDLING

- Traps by invoking *RaiseException*()
  - Passing arguments that indicate the exact cause of the trap
  - Calls *ExceptionHandler (in userprog/exception.cc)*

- *"syscall"* is passed
  - Syscall code in r2 register
  - Additional arguments in registers r4-r7
  - Result of the system call - register r2 on return

IN PROGRAMMING ASSIGNMENT

- Syscalls
  - Exit
  - Fork
  - Read/Write
  - Exec
- Multiprogrammed Kernel (-x prog, -quantum timeslice)
  - Run multiple user programs implemented with the Syscalls
  - Each user program runs as a Nachos Thread
    - Syscalls
    - Memory Manager
- Memory Management (Virtual Memory)
  - Demand paging
  - Page Replacement
  - Swapping
  - Hints:
    - Frame Table
    - Page Table per process
    - Swap Table

HOW TO TEST

- Write user programs using the Syscalls
- Test user programs for multiprogrammed kernel
- Test with larger memory needed programs
  - ~/test/matmul
  - ~/test/sort
  - Requires lots of memory (much larger than the physical memory)
- Run user programs concurrently and get the correct result
  - Use RR scheduling
  - -quantum to set timeslice
- **YOU are NOT allowed to modify the memory size in machine**

VIRTUAL MEMORY HINTS

- Swap space
  - Use file system
  - Read files in ~/filesys directory
- Think HOW TO LOAD pages correctly
- DEAL WITH Page Fault Exception

MAINTAIN TABLES

- Page Table
  - One per process
- Frame Table
  - Record every frame's info
  - Frame represent a physical page
  - FrameTableEntry
- Swap Table
  - Record every sector's info in swap
  - SwapTableEntry
  - Sector size == frame size

DESIGN YOUR OWN MEMORY MANAGER

- You can define your own class for memory manager
- Design your own page replacement method
  - Or you can choose one of replacement algorithms discussed in the class
  - NOT FIFO

USEFUL FILES

- machine.cc, addrspace.cc, translate.cc
- READ header files first