Nachos 2 (Task Summary)

Task – 01 (Address Space : Virtual Memory Support)

Provide an interface **PageTable** to do the following tasks

- Keep track of free pages
- Allocate from free pages
- De-allocate pages

You also need to do the followings:

1. Edit constructor of AddrSpace to link virtual address/ virtual page number(vpn) and physical address/physical page number(ppn) properly
2. Write functions for reading from and writing to virtual memory.
   - First, convert virtual address to its physical address
   - Then read/write in physical address using Machine::ReadMem() and Machine::WriteMem()

Save the <processId, PageTable> pair inside kernel.

Task – 02 (System Call: Exec and Exit)

Exec

- Implement both system calls with standard arguments provided in **syscall.h**
- Carefully review the code inside **userprog/progtest.cc (**Startprocess() function to be precise)
- A new pageTable must be created when Exec is called. For now, you donot need to implement any **Page Replacement Policy**. If there are not enough number of free pages, throw an exception.
- If you find that there are not enough pages to run the new process and you have already allocated some pages for it, **de-allocate** them. Anyway, it is suggested to keep track of **Number of free pages** inside the **PageTable** data structure.
- Properly copy filename (you may need to use virtual memory read/write here)
- Check if the filename is too long to fit in free pages. Handle the case where a filename can be in non-contiguous pages.

Exit

- Implement both system calls with standard arguments provided in **syscall.h**
- Print status value (argument of Exit) for debugging purpose.
- You must **free the allocated pages** once a process is done executing.

Task – 03 (System Call: Read and Write)

- Read **console.cc** to see the implementation of a console. Also see **userprog/progtest.cc (consoleTest()** function) to see how a console object is used.

- Create a **console** object as global inside **Initialize()** in **threads/system.cc**
- Implement a thread safe, synchronized console interface (take help of **consoleTest**) to control and support read/write in console.
- Now, implement **Read** and **Write** system calls as in **syscall.h** (You don't have to worry about file read/write. Just support these system calls for console i/o)

Task – 04 (Bonus: User Exception Handling)

- Make the kernel bulletproof so that any exception in user program doesn't terminate the kernel or affect any other process. Go through **exception.cc** and try to understand how to do it. **As this is a bonus task, you won't get any detailed documentation** ☹ .

Advice regarding System Call Implementation

- Create a separate function for each system call inside **exception.cc .** Call this function from the exception handler.
- Increment **PC** properly. (You will get the instructions inside the slide provided)

Running User Programs

Those who are having hard time running nachos –x command, use the following after you are in **userprog** directory and **make** is complete

```
./nachos –x ../test/matmult
```

Here, **matmult** is a test program provided with nachos. Use other programs if you wish.

Nachos

Not Another Completely Heuristic Operating System

Nachos: Present Status

- Currently Nachos does not support multiprogramming.
- Currently Nachos only implements ***halt*** system call.
- No support for reading and writing from user program.
- No support for running multiple thread spawning from user program (multithreaded user program).

Nachos Lab 2

Step 1: User program

User programs

- All user level processes execute as a kernel-level thread in Nachos.
- Nachos can run user programs as long as they make only those system calls that are understood by Nachos.
- User program must be converted into appropriate format to run on MIPS architecture.

User programs (contd.)

- Written in ANSI C, lack of many libraries like printf (examples in nachos-3.4/code/test/).
- Reside in user address space (can see only data structures declared/created inside the program).
- Communication with the kernel through system calls (open file, start a new process, etc.) defined in userprog/syscall.h.

Difference between Nachos executable and Linux executable

- Linux executables are in coff format.
- Nachos executables are in noff format.
- There must be a way to convert the coff format into noff format.
- **See /code/bin/coff.h and /code/bin/noff.h**

noff

- Noff header
  - Resides at the beginning of the file and describes the contents of rest of the file, giving info about the program's instruction, initialized variables
  - *Noffmagic,* header maintains a reserved magic number indicating that the file is in noff format.
  - Before attempting to execute a file, nachos checks that number.

Makefile (~/test/makefile)

- **halt.o: halt.c**
  $(CC) $(CFLAGS) -c halt.c

  halt: halt.o start.o
  $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
  ../bin/coff2noff halt.coff halt

- The above compiles a user program called *halt.c* .
- resulting object file is then linked with another file called *start.o* to produce a binary called *halt.coff* .
- Since we want to run *halt.c* in Nachos, we then invoke the program *coff2noff* to produce a binary file called *halt* in the *Noff* format.

Makefile contd.

```
1 all: halt shell matmult sort
2 start.o: start.s ../userprog/syscall.h
3 $(CPP) $(CPPFLAGS) start.s > strt.s
4 $(AS) $(ASFLAGS) -o start.o strt.s
5 rm strt.s
6 shell.o: shell.c
7 $(CC) $(CFLAGS) -c shell.c
8 shell: shell.o start.o
9 $(LD) $(LDFLAGS) start.o shell.o -o shell.coff
10 ../bin/coff2noff shell.coff shell
```

Makefile to compile and run shell.c

Makefile contd.

- Lines 2-5 generate object code for *start.s*.
- Lines 6 and 7 generate object code for *shell.c*.
- Lines 8 and 9 produce an executable binary.
  - Note: listing *start.o* before *shell.o* ensures that the code for *start.s* resides before that of the main program.
- Line 10 translates the binary into the Noff format. *shell*, is thus, an executable ready to execute under Nachos.

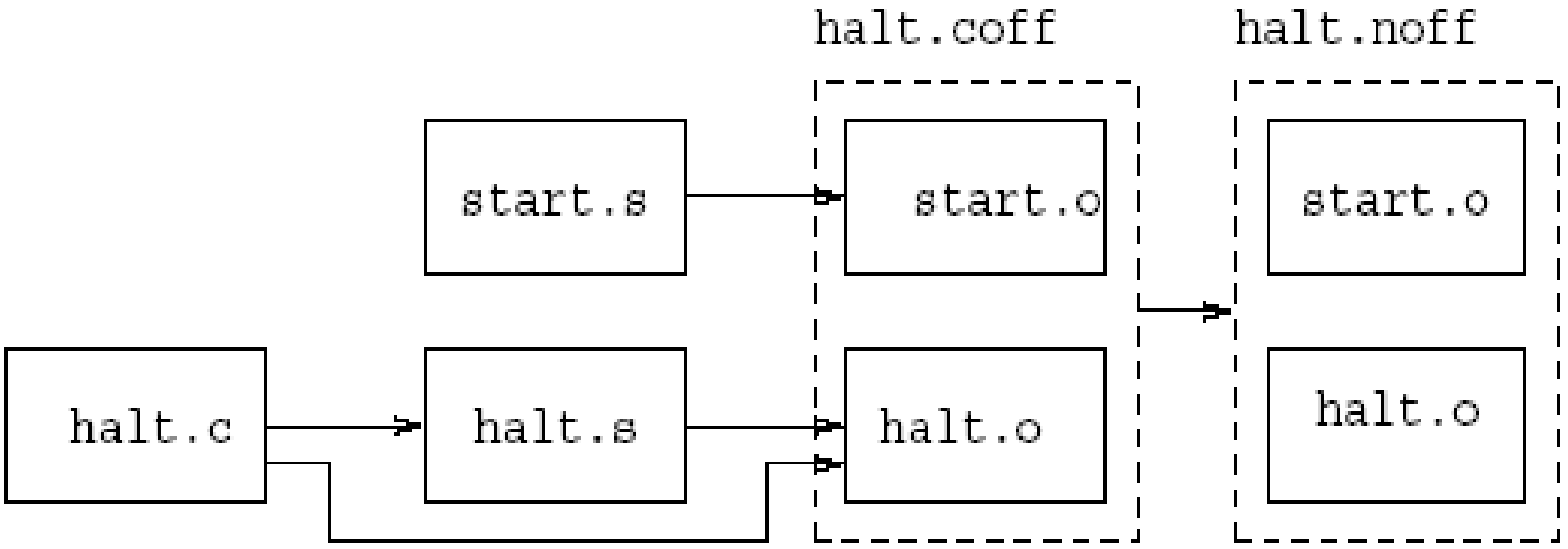
What is start.s?

- The file *start.s* contains assembly language code that needs to be executed before the code of each user program's main routine.
- For proper execution of *shell.c* , we need to make sure that the code in *start.s* resides before that of its main program.
- In addition to containing this initialization code, *start.s* also contains stub modules for invoking system calls .

What is start.s?

- Specifically, the very first instruction in *start.s* calls the user-supplied *main* routine, whereas the second instruction invokes the Nachos *Exit* system call, insuring that user processes terminate properly when their main program returns.

Construction of Nachos User Program



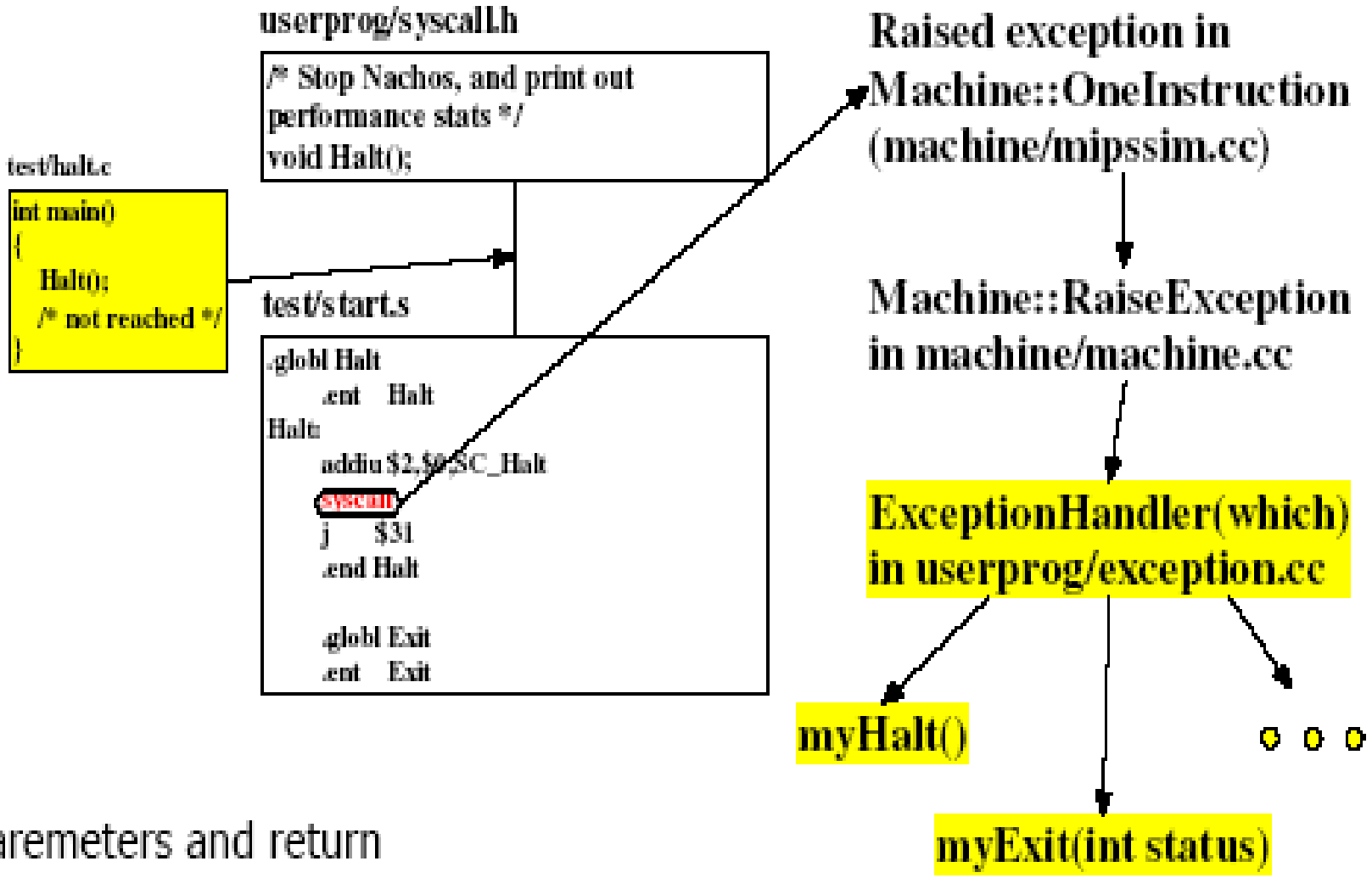
Construction of Nachos User Program

Nachos Lab 2

Step 2: System Call

System Call in Nachos

- Nachos kernel serves user programs via system calls.
- System call causes an exception (interrupt) and changes from user mode to kernel mode.
- As a result the exception handler is called eventually (userprog/exception.cc).
- Arguments are placed in registers r4 - r7.
- Return value – in r2 (see userprog/exception.cc)



Paremeters and return values are in registers!

System Call in Nachos

Need to perform following steps -

- get the parameters for the system call from registers.
- convert any pointers (ie, char* pointers) from virtual to physical addresses.
- pass the parameters to corresponding system call handler.
- place the return value in the appropriate register.

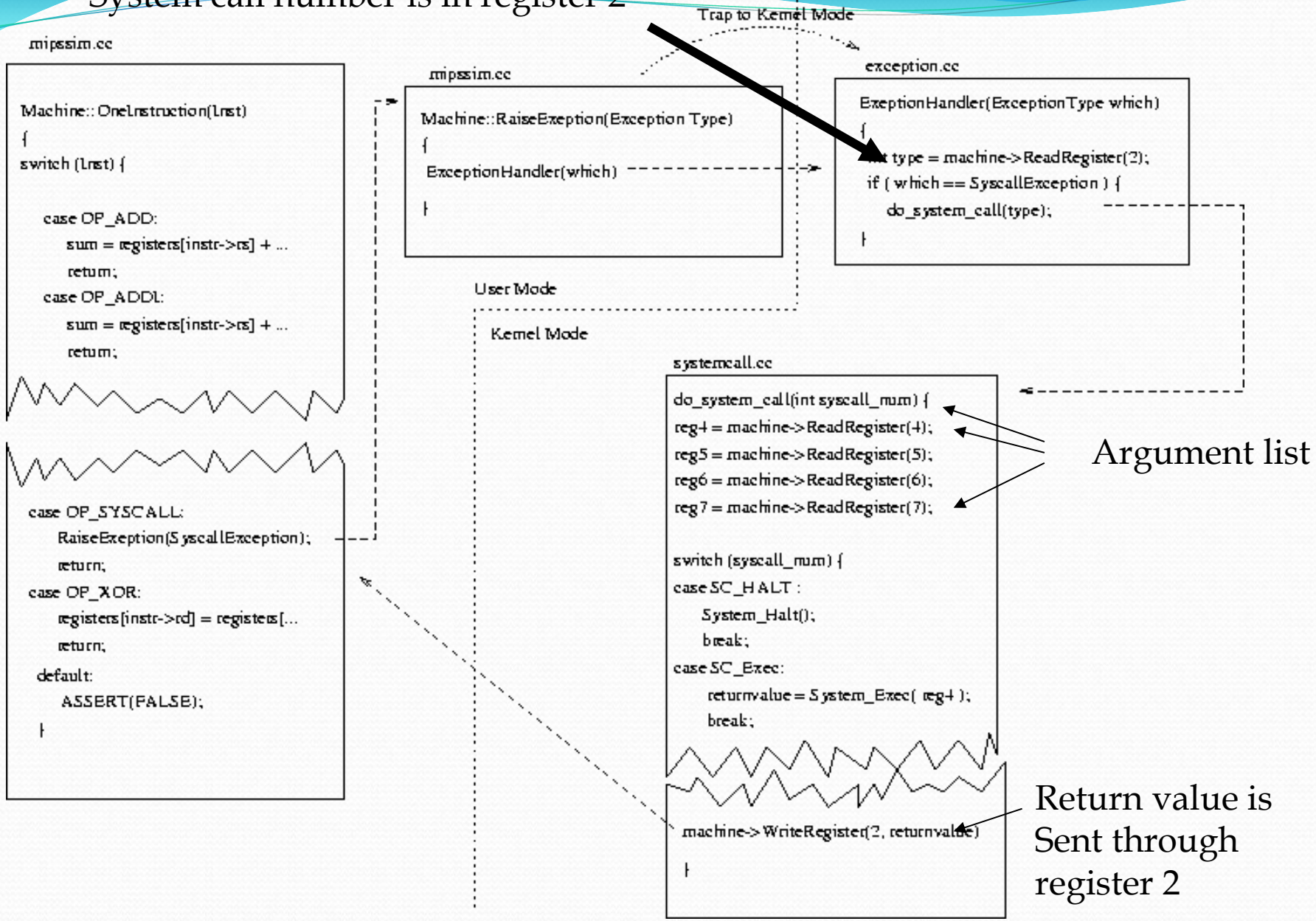
System Call in Nachos

- ExceptionHandler parameter identifies exception
- If it's a system call:
  - R2: system call code (defined in syscall.h)
  - R4: arg1
  - R5: arg2
  - R6: arg3
  - R7: arg4
  - Return any result in R2.

System Call in Nachos

- **Void Write(char* buf, int size, openFileID id);**
- **R2 = SC_Write**
- **R4 = &buf**
- **R5 = size**
- **R6 = id**

System call number is in register 2



Code Example

```
void ExceptionHandler(ExceptionType which) {
int type = machine->ReadRegister(2);
if (which == SyscallException) {
switch(type) {
case SC_Halt: {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
}
case SC_Create: {
        // your code;
        break;
}
}
```

Code Tips for system call

- infinite system call loop : Re-executing the syscall instruction after returning from the system call.
- Solution: update PCReg to point to the instruction following the syscall instruction.
- The following code properly updates the program counter.

Code Tips [must be added]

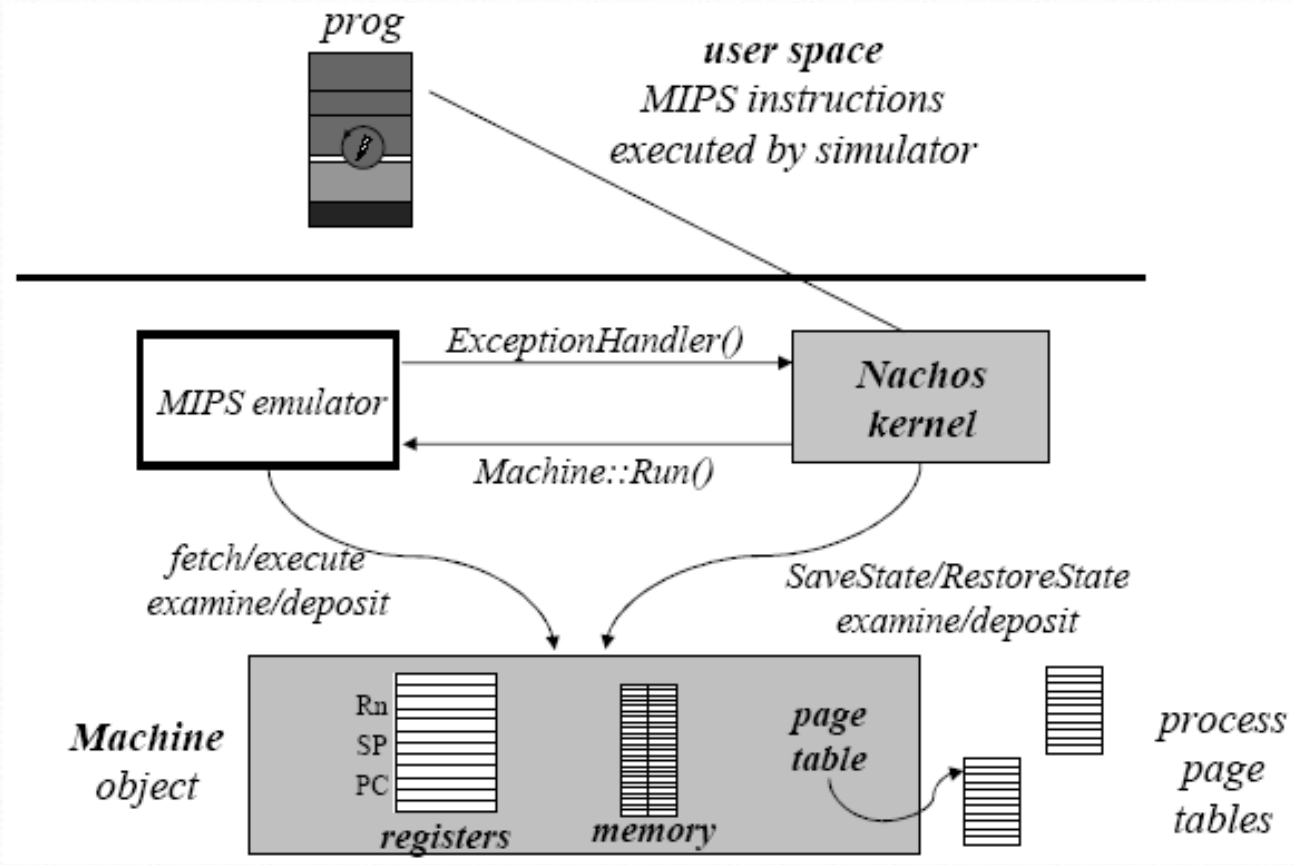
- pc = machine->ReadRegister(PCReg)
- Machine->WriteRegister(PrevPCReg,pc)
- Pc=machine->ReadRegister(NextPCReg)
- Machine->WriteRegister(PCReg,pc)
- pc += 4
- Machine->WriteRegister(NextPCReg,pc)

Nachos Lab 2

Step 3: Multiprogramming

How a user program is loaded in main memory?

- Remember that when you start Nachos, it automatically creates a “main” kernel thread that takes care of initializing Nachos and running the user program, turning on debugging, etc.
- It then calls *StartProcess*() [see /userprog/progtest.cc].

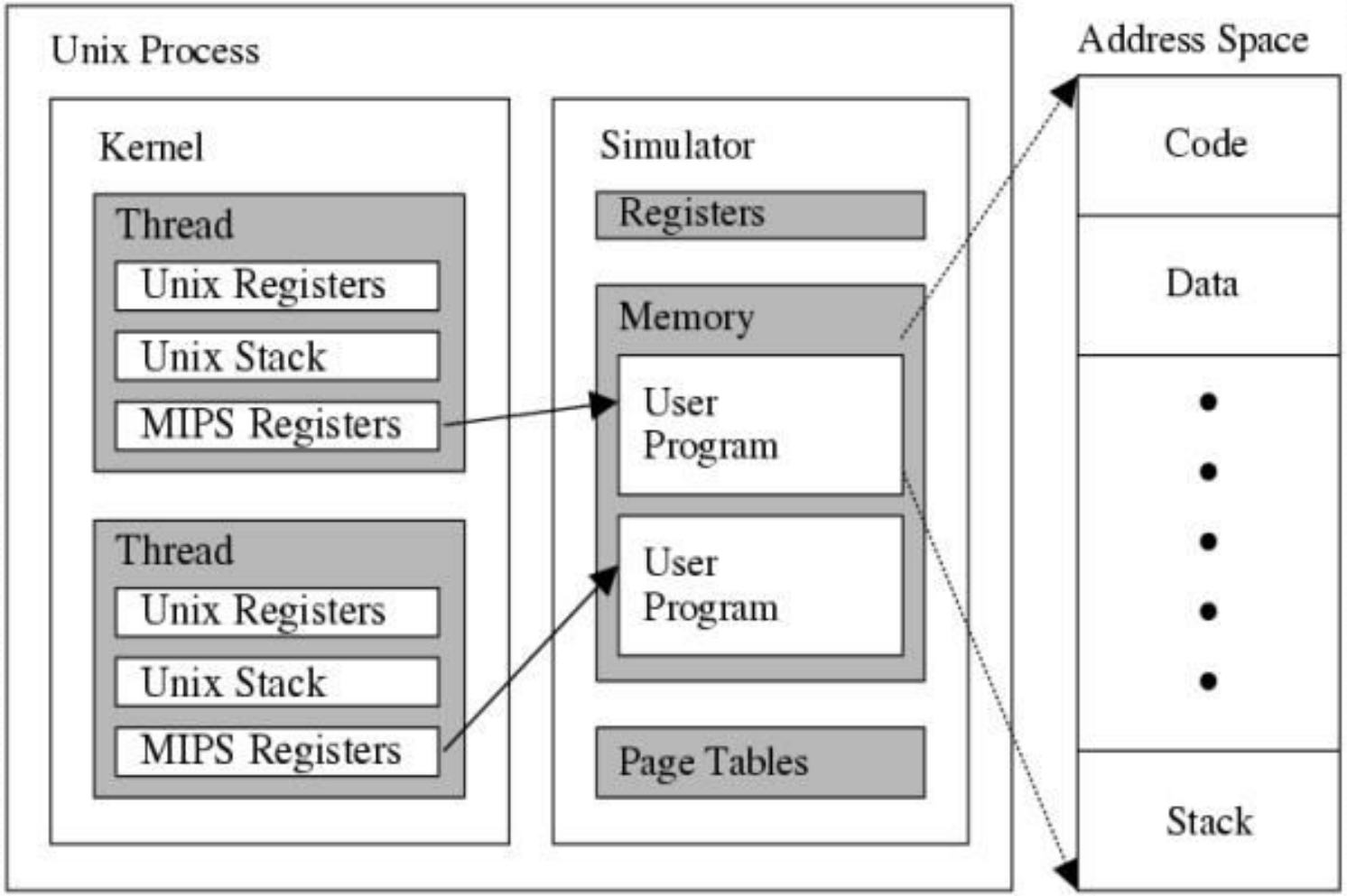

prog
user space
MIPS instructions
executed by simulator
MIPS emulator
ExceptionHandler()
Nachos
kernel
Machine::Run()
fetch/execute
examine/deposit
SaveState/RestoreState
examine/deposit
Machine
object
Rn
SP
PC
registers
memory
page
table
process
page
tables

How a user program is loaded in main memory?

- Inside *StartProcess()*, nachos initializes a new AddrSpace for the executable that
  - Allocates necessary memory for loading the user program.
  - Maintains a page table for virtual to physical memory mapping.

Implementation in Address Space

- See /userprog/addrspace.cc.
- AddrSpace has two private variables – pageTable and numPages.
- Closely look the constructor.
  - How address space is initialized.
  - How page table info is populated.
  - How the user program is loaded into main memory.



Inside Nachos Process: Nachos Kernel, Machine Simulator, User Programs

Implementation in Address Space

```
pageTable = new TranslationEntr [numPages];
  for (i = 0; i < numPages; i++){
     pageTable[i].virtualPage = i;
     pageTable[i].physicalPage = i;
     pageTable[i].valid = TRUE;
     pageTable[i].use = FALSE;
     pageTable[i].dirty = FALSE;
     pageTable[i].readOnly = FALSE;
      }
```

- You must change it to support multiprogramming.

Implementation in Address Space

- *ReaAt*() is used to read data from a specified memory location.
- Remember, the address must be physical memory location.
- As nachos only supports one user program, there is one to one mapping between virtual and physical memory address.

Implementation in Address Space

- Can you tell why nachos works although there is no conversion from virtual to physical address?
- Because….
- See previous slides (One to one mapping).
- In case of multiprogramming, that will not work.
- You may need a virtual to physical memory address converter.

Loading user program

- After all initialization, machine->Run() is called for each process.
- It actually invokes the MIPS simulator.
- MIPS simulator then fetches one instruction at a time and executes it.
- **How does it work?**

Running Multiple User Programs

Program_A.c

```
int main() {
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Write("Program A\n",10,1);
Halt();
}
```

Program_B.c

```
int main() {
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Write("Program B\n",10,1);
Halt();
}
```

Program_C.c

```
int main() {
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Write("Program C\n",10,1);
Halt();
}
```

Running Multiple User Programs

- Sample Execution
- > ./nachos –x a b c

  Loading program: a

  Number of pages: 12

  Size of code segment: 656

  Virtual address of code segment: 0

  Size of read only data segment: 96

  Virtual address of read only data segment: 656

  Program A

  Program A

  .....

Running Multiple User Programs

Context Switch: current = Program_A, selected = Program_B

Program B

Program B

.....

Context Switch: current = Program_B, selected = Program_C

Program C

Program C

.....

Nachos Lab 2

Step 5: Console I/O

Console Implementation in Nachos

- User program can't directly read from or write to console.
- Multiple program may compete for console access.
- So, there must be some synchronization.
- See **/machine/console.h** & **/machine/console.cc**.
- Also see **ConsoleTest**() in /userprog/progtest.cc.

Console: Creation

- Create an object of the Console class during Nachos startup (Initialize() function in threads/system.cc).
- Provide global access for that object (global variable in threads/system.h).

Console Interrupts

- Q: How console interrupts are handled in Nachos?
- A: The functions are called, which are provided as arguments during console creation in the constructor (may be written by you):
- **Console::Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, int callArg);**
- Check example in userprog/progtest.cc.

Console: Output

- Write one character at a time – Console::PutChar()
- *TRANSMIT COMPLETE* interrupt when finished.
- User-supplied handler is invoked (the function provided as an arguments during console creation).
- Implement ability to print any string, not only chars for Write() system call.

Console: Input

- One character at a time
- Console::GetChar() always return immediately: a char or EOF if nothing in the buffer.
- Interrupt generated when a key is stricken.
- Use synchronization primitives to avoid active waiting in the loop.
- Unix issue: in the lab environment a string is send from Unix to nachos program only after “Enter” is pressed.

Read/Write from user program

- Implement a new class SynchConsole with synchronization facilities similar to ConsoleTest().
- **SynchConsole:Write(char *)** – writes provided string to console synchronously.
- **SynchConsole:Read(char*)** – reads from console and copies to string.

How does user program read/write?

- Through system call – read and write.
- Tips: You have to read the string from the user space memory to kernel space using **Machine->readMem()** and then write using synchConsole.
- Similarly, you have to write the string to user space from kernel space using **machine->writeMem()** after reading it using synchConsole.

Nachos Lab 2

Coding Tips

What Files Do I Change?

- The following files may require changes or close attention to get a feel for what is going on.
  - Any code wrapped in #ifdef USER_PROGRAM … #endif will be included when you compile Nachos.
  - **exception.cc - This is where the actual code for your syscalls will go. So your Fork syscall could be a function called ForkSyscall() in this file.**
  - machine.h – change *numPhysPage* 32 to 128.

What Files Do I Change?

- **system.cc**
  - You should need to make minimal changes to this file. You may need to allocate any new operating system structures here.
- **addrspace.cc and addrspace.h**
  - **This is where code related to managing user-space memory should go.**
- Examining how stack space is allocated for a new process (see InitRegisters) might repay some extra attention.
- **syscall.h**
  - You will find the syscall numbers and the prototypes (not the actual syscall function) of the syscalls here.
- So the syscall function call in your user program must follow exactly this format.

User program Tips

- **Upon console I/O, the *console* device, schedules interrupts into the future to poll the console for characters. This may cause Nachos to continue execution even after all user process have finished and there are no more threads on the ready list. To get around this, user programs should call Halt() when they have finished execution.**