

# Nachos in CPS110

Labs 1-3 use Nachos as a thread library.

- Lab 1: understanding races.
- Lab 2: implement and use synchronization primitives.
- Lab 3: complex synchronization.

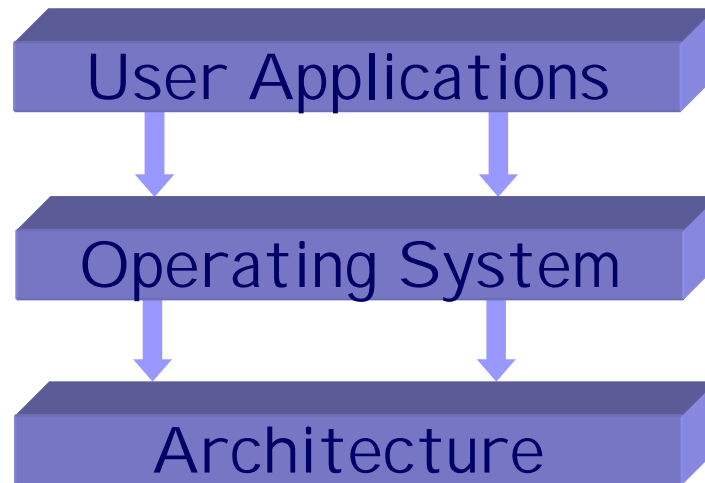
Labs 4-6 use Nachos as an operating system.

- Lab 4: concurrent user programs.
- Lab 5: I/O with files and pipes.
- Lab 6: virtual memory.

# What is Nachos?

What is an operating system?

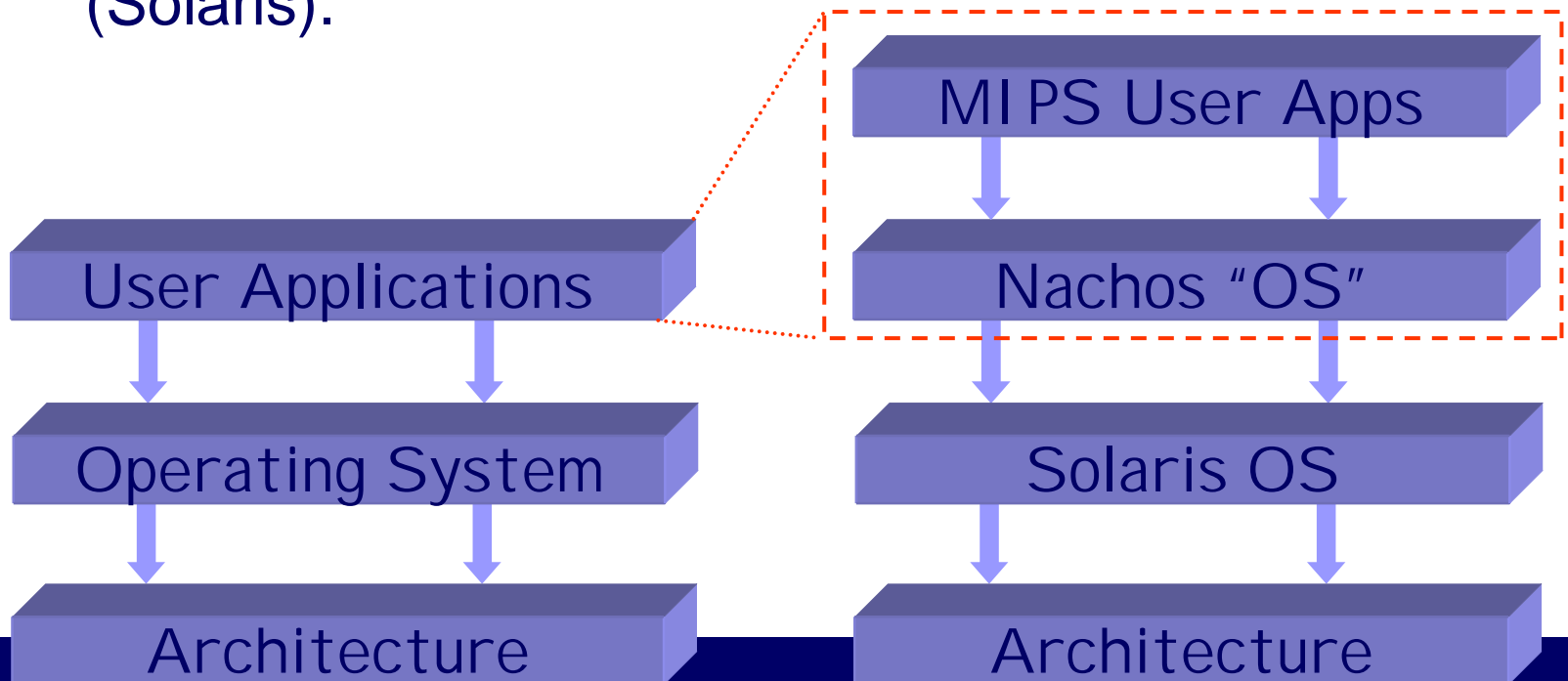
- “friendly” interface between user programs (Powerpoint) and hardware (x86, IDE disk, video card, etc).



# What is Nachos? (reality)

Nachos looks, feels, and crashes like a “real” OS.

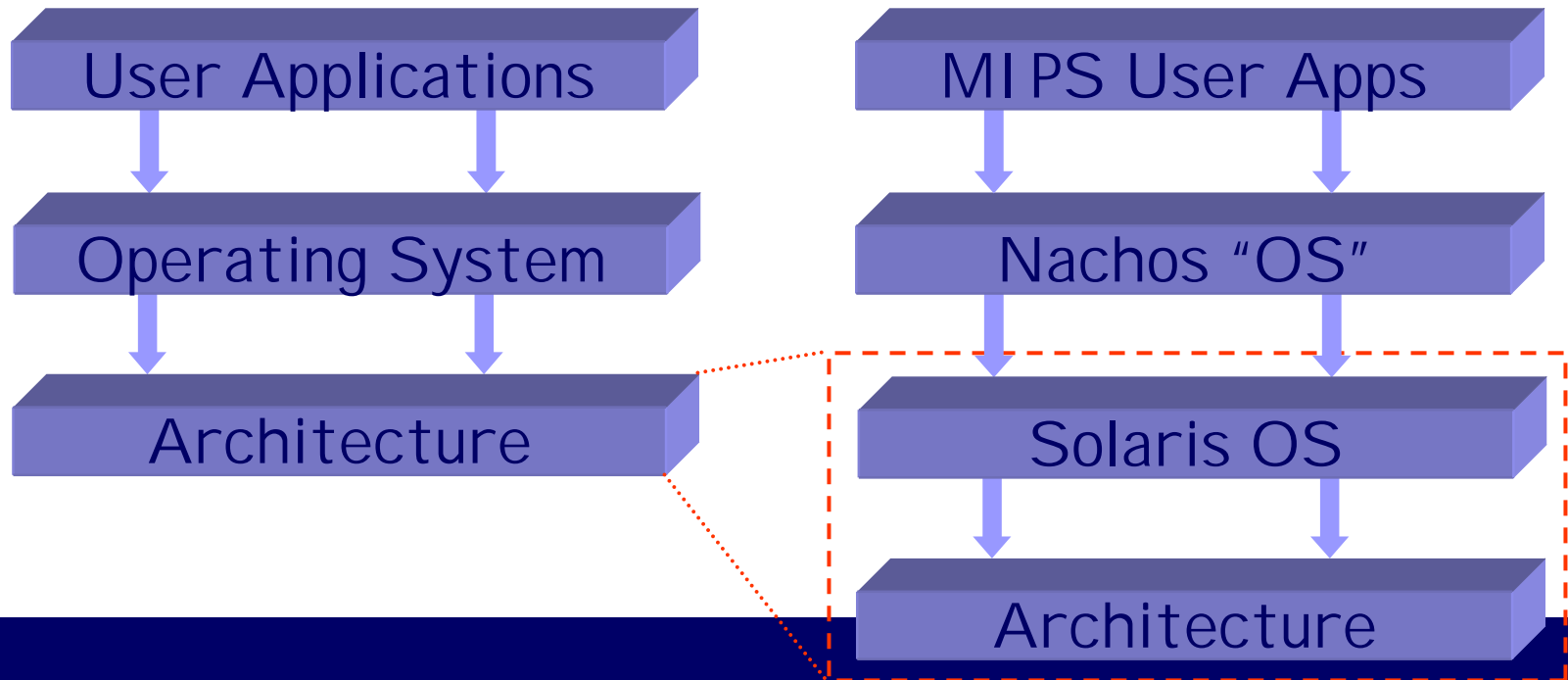
- Both the Nachos “OS” and test programs run together as an ordinary process on an ordinary Unix system (Solaris).



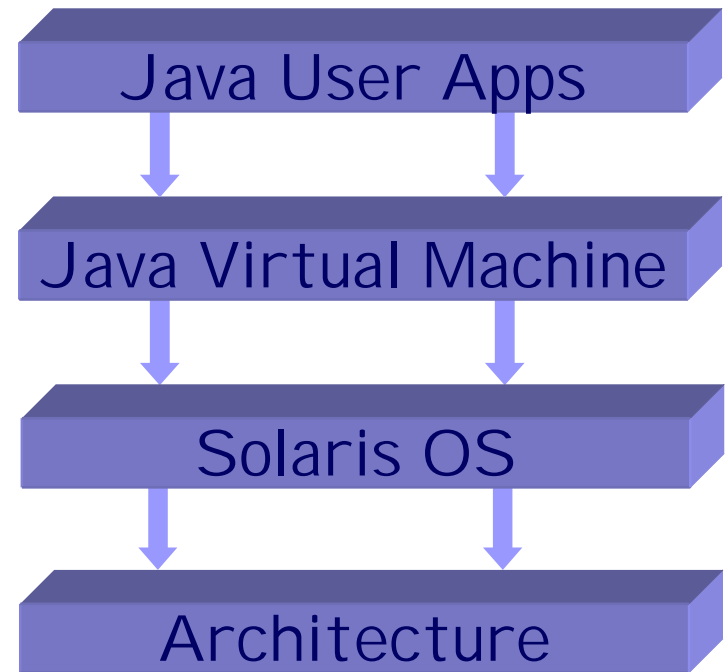
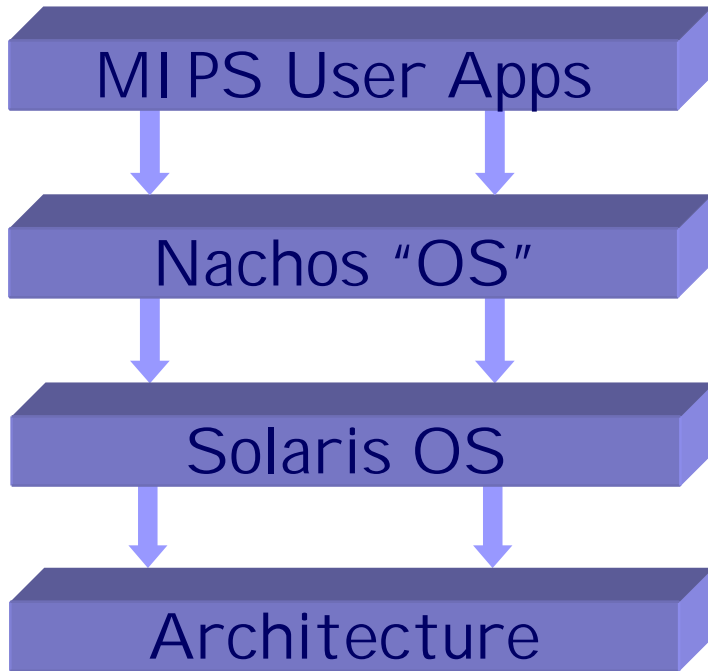
# What is Nachos? (for us)

Nachos runs *real* user programs on a *simulated* machine.

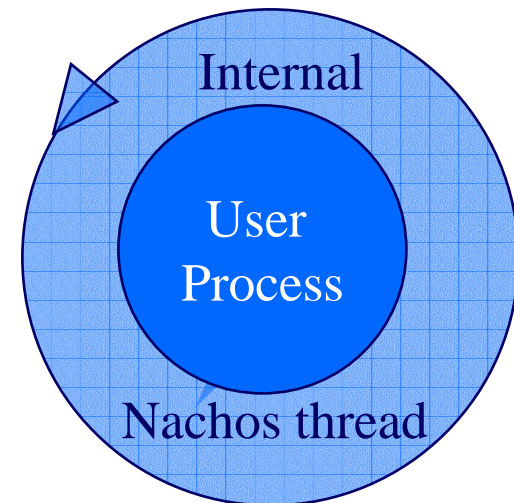
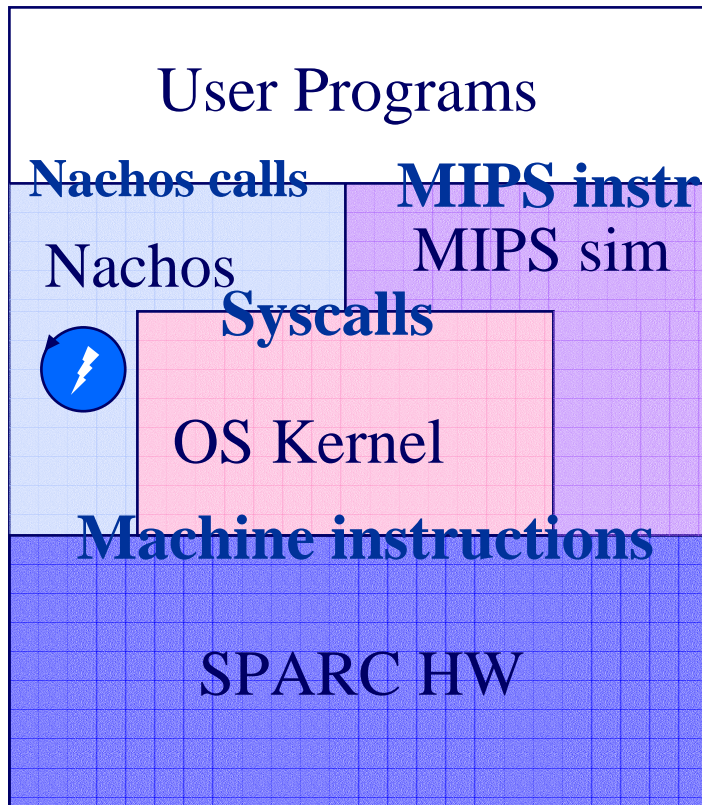
- Nachos MIPS simulator executes *real* user programs.
- The real OS is treated as part of the hardware.



# Look familiar?



# Introducing User Programs into Nachos



Conceptually:  
Nachos thread encapsulates  
user program, remains the  
schedulable entity

# Nachos Systems Call (Process)

userprog/syscall.h

**Spaceid Exec** (`char *name, int argc, char** argv, int pipectrl`) - Creates a user process by creating a new address space, reading the executable file into it, and creating a new internal thread (via `Thread::Fork`) to run it. To start execution of the child process, the kernel sets up the CPU state for the new process and then calls `Machine::Run` to start the machine simulator executing the specified program's instructions in the context of the newly created child process.

**Exit** (`int status`) - user process quits with `status` returned. The kernel handles an `Exit` system call by destroying the process data structures and thread(s), reclaiming any memory assigned to the process, and arranging to return the exit status value as the result of the `Join` on this process, if any.

**Join** (`Spaceid pid`) - called by a process (the joiner) to wait for the termination of the process (the joinee) whose `Spaceid` is given by the `pid` argument. If the joinee is still active, then `Join` blocks until the joinee exits. When the joinee has exited, `Join` returns the joinee's exit status to the joiner.

## StartProcess(char \*filename)

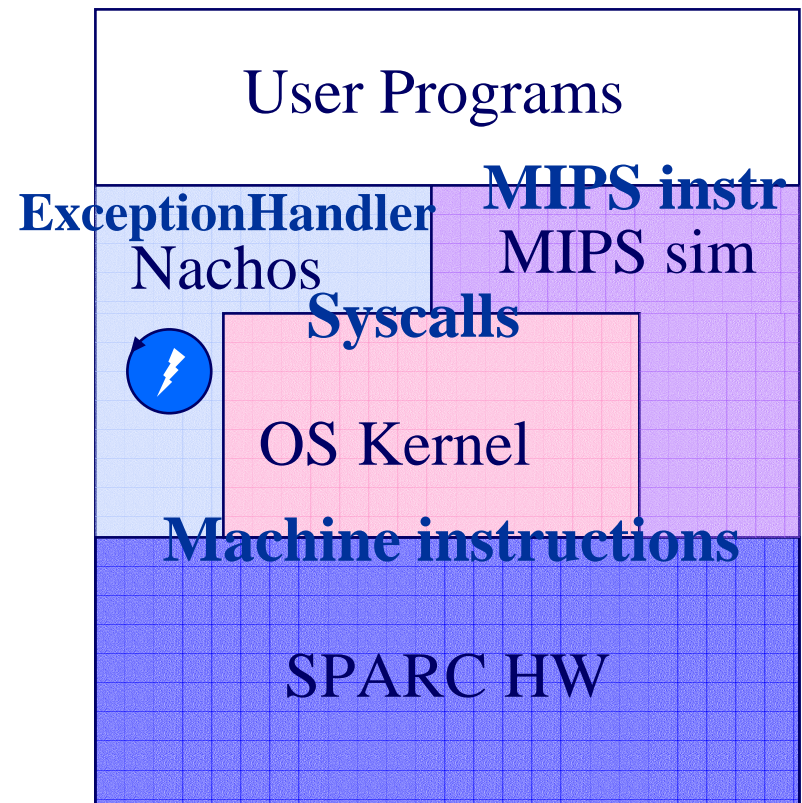
```
{
  OpenFile *executable = fileSystem->Open(filename);
  AddrSpace *space;
  if (executable == NULL) {
    printf("Unable to open file %s\n", filename);
    return;
  }
  space = new AddrSpace(executable);
  currentThread->space = space;
  delete executable;          // close file
  space->InitRegisters();     // set the initial register values
  space->RestoreState();      // load page table register
  machine->Run();             // jump to the user program
  ASSERT(FALSE);            // machine->Run never returns;
                             // the address space exits
                             // by doing the syscall "exit"
}
```

→ Exec

## ExceptionHandler(ExceptionType which)

```
{  
    int type = machine->ReadRegister(2);  
  
    if ((which == SyscallException) && (type == SC_Halt)) {  
        DEBUG('a', "Shutdown, initiated  
        by user program.\n");  
        interrupt->Halt();  
    } else {  
        printf("Unexpected user mode  
        exception %d %d\n", which, type);  
        ASSERT(FALSE);  
    }  
}
```

Note: system call code must convert user-space addresses to Nachos machine addresses or kernel addresses before they can be dereferenced



# Lab 4: Fun with AddrSpace

Lab 4: concurrent processes with Exec, Exit, Join.

- Nachos already provides:

Process abstraction (Thread + AddrSpace).

Context switching (swap registers, page tables).

Timeslicing.

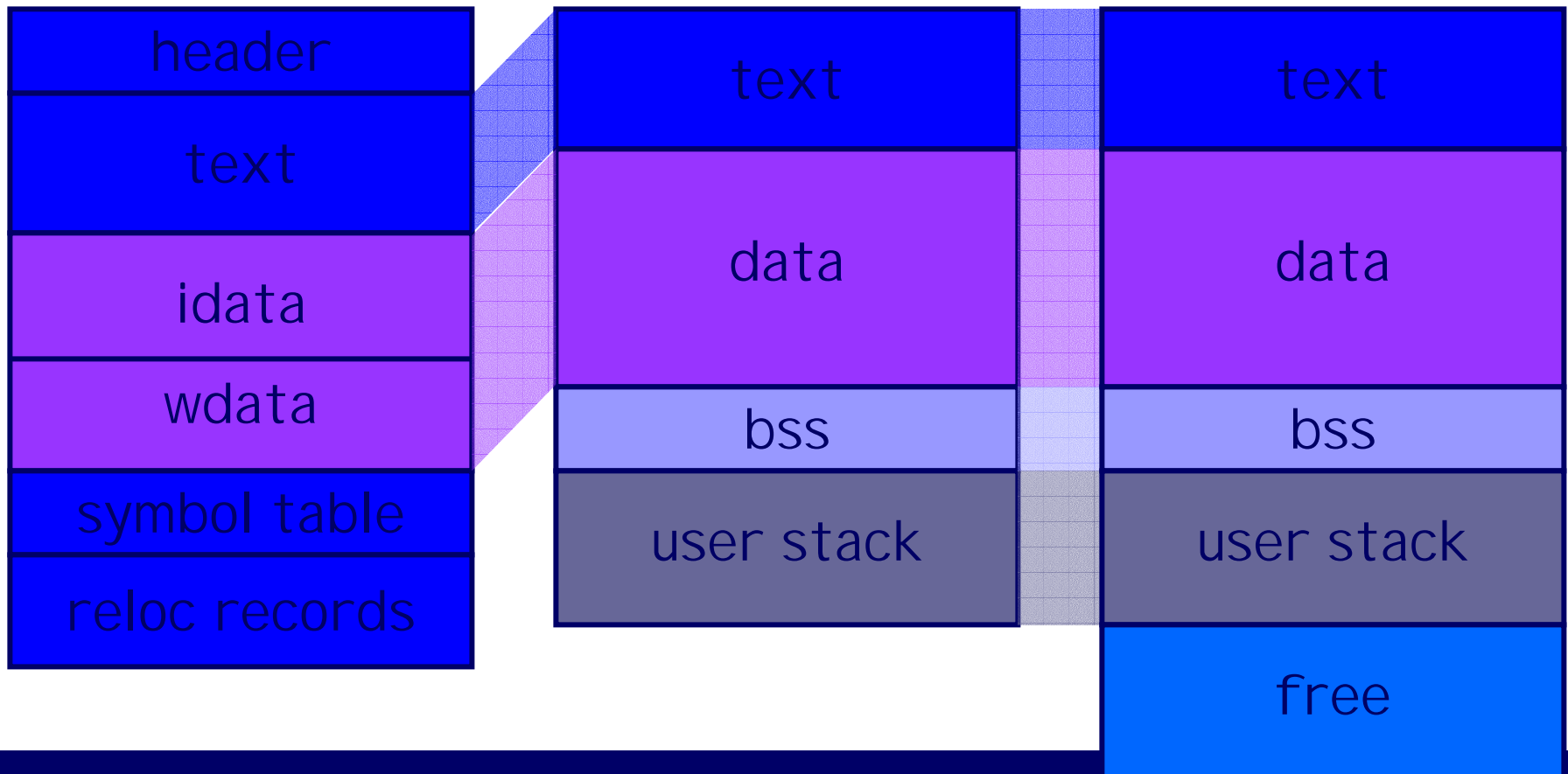
- But:

AddrSpace (as provided) allows only one process.

Up next:

- executable, address space review.
- “Fixing” AddrSpace.

# Executable into AddrSpace



## AddrSpace::AddrSpace(OpenFile \*executable)

```
{ ...
    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);
    size = numPages * PageSize;
    ASSERT(numPages <= NumPhysPages);    // check we're not trying
    // to run anything too big --

    // at least until we have virtual memory
```

```

// first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE; // if the code segment was entirely on
                                   // a separate page, we could set its
                                   // pages to be read-only
}

// zero out the entire address space, to zero the uninitialized data segment
// and the stack segment
// bzero(machine->mainMemory, size); rm for Solaris

memset(machine->mainMemory, 0, size);

```

```
// then, copy in the code and data segments into memory
if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
          noffH.code.virtualAddr, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
                      noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
          noffH.initData.virtualAddr, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
                      noffH.initData.size, noffH.initData.inFileAddr);
}
}
```

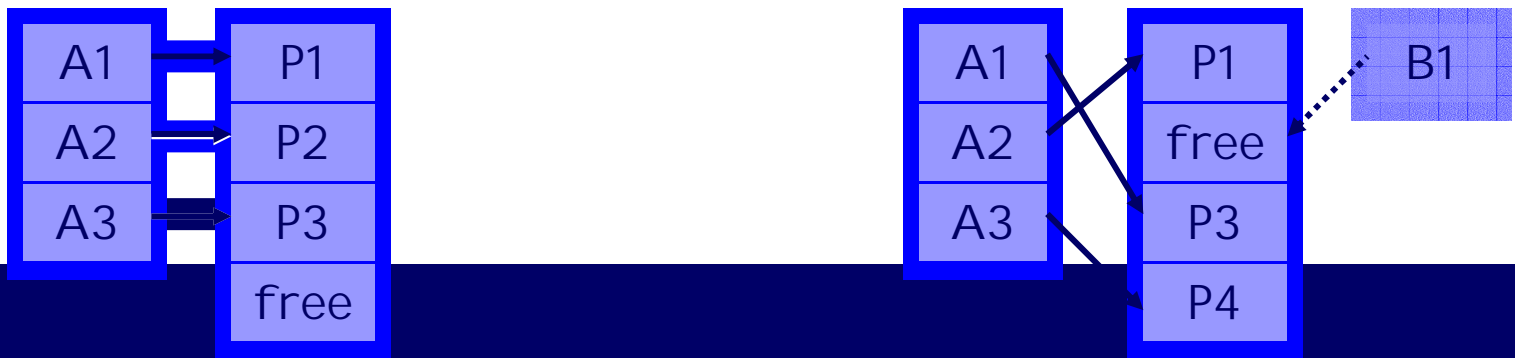
# “Fixing” AddrSpace

AddrSpace::AddrSpace(OpenFile \*executable)

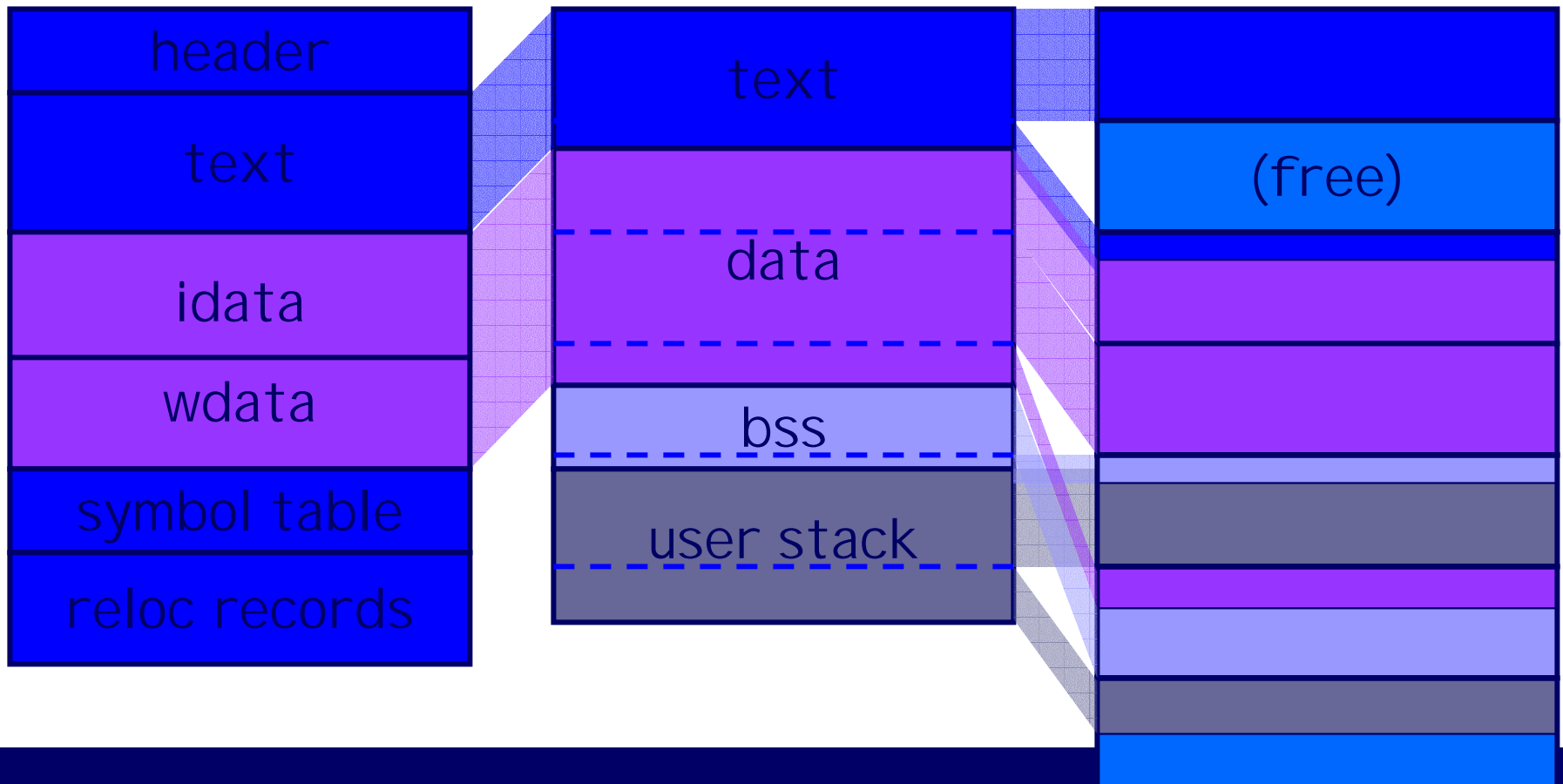
1. Computes user address space size, in pages.
2. Creates pageTable, **maps first N physical pages.**
3. Loads executable into address space **in one swoop.**

Lab 4 executive summary:

- Use available pages instead of first N pages.
- Executable->ReadAt must not cross page boundary.



# Executable into AddrSpace



# Where are we now?

Old (provided) Nachos:

- Only one AddrSpace, therefore only one process.

New (post lab 4) Nachos:

- Supports concurrent AddrSpace instances.

With new AddrSpace, multiprocessing “just works.”

Up next:

- Nachos MIPS machine simulator.
- Nachos Syscalls.

# The machine simulator

Getting started:

- StartProcess sets up address space, process state.
- Machine::Run starts the machine simulator.

The machine simulator:

1. Fetch instruction pointed to by program counter.
2. Execute instruction, increment program counter.\*  
    Raise exceptions (no auto PC++).  
    Occasionally switch between active processes.
3. Repeat.

# System calls

The machine simulator emulates MIPS instructions.  
One instruction, `syscall`, invokes `ExceptionHandler()`.

- On entry:

Register 2 identifies `syscall` (`Exec`, `Exit`, etc.).

Registers 5, 6, and 7 contain the argument(s).

- On exit:

Register 2 contains the `syscall` return value.

Must increment the program counter.

*What happens if it doesn't? When might that make sense?*

# The Exec system call

From above (user program):

```
SpaceID id = Exec("foo");
```

```
.rdata          // immutable data
$LC0:           // "foo" marker
.ascii "foo\0"
.text           // text segment
addiu $2,$0,SC_Exec // Reg2=type
la $4, $LC0     // Reg4 = VA
syscall         // trap to kernel
move $2, $0     // Reg2 = result
```

From below (nachos kernel):

```
Machine calls ExceptionHandler
ExceptionHandler(which=Syscall)
type = ReadRegister(2); // SC_Exec
vaddr = ReadRegister(4); // user VA
str = ReadStr(vaddr);   //~ read str
result = MyExec(str);   // do Exec
WriteRegister(2, result); // set rval
PrevPC = PC;           //~ inc PC
PC = NextPC;           //~ inc PC
NextPC = NextPC + 4;   //~ inc PC
return;                // resume
```

# Lab 5: I/O

Lab 5: I/O with console, files, and pipes.

- I/O system calls: Create, Open, Close, Read, Write.
- Bind fileids 0 (stdin) and 1 (stdout) to console.
- Pipe one process' stdout to another's stdin.

# Create, Open, Close

The “easy” system calls:

- `void Create(char *name);`
- `OpenFileID Open(char *name);`
- `void Close(OpenFileID id);`

Kernel support is already there! ([Lab 5 spoiler](#))

- `FileSystem::Create`
- `OpenFile::OpenFile + FileSystem::Open`
- `OpenFile::~~OpenFile`

Your job:

- Manage, translate `OpenFileIDs` to `OpenFiles`.

# Read, Write

I/O system calls:

- `void Write(char *buffer, int size, OpenFileID id);`
- `int Read(char *buffer, int size, OpenFileID id);`

Again, leverage existing kernel support :

- `OpenFile::Read, OpenFile::Write`

Keep in mind:

- Buffer may cross page boundary (you've seen this).
- Special handling for stdin/stdout (id 0/1).

These may use the console, or a pipe!

# Console

Console I/O:

- Special OpenFileIDs 0 and 1 read/write the console.

Again, some Nachos support:

- Console::GetChar, Console::PutChar

Your job:

- SynchConsole for atomic console Reads/Writes.



# Pipes

Pipe I/O:

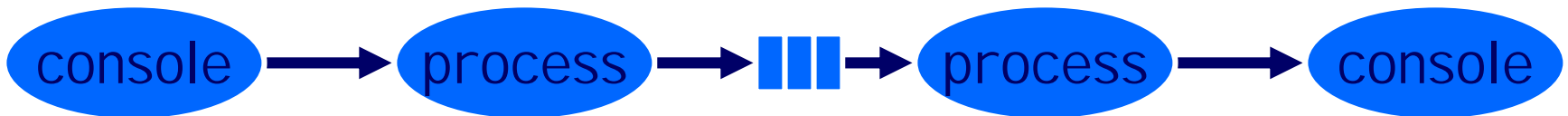
- Bind one process' stdout to another's stdin.

No Nachos support, but you've got a bag of tricks:

- Lab 3 BoundedBuffer.

Your job:

- Connect stdout/stdin to BoundedBuffer.



# Lab 6: Virtual Memory

- demand paging: use page faults to dynamically load process virtual pages on demand
- page replacement: enabling your kernel to evict any virtual page from memory