Project 6 Discussion

Class Thread

```
fields
    regs: array [13] of int          -- Space for r2..r14
    stackTop: ptr to void            -- Current system stack top ptr
    name: ptr to array of char
    status: int         -- JUST_CREATED, READY,
                        -- RUNNING, BLOCKED, UNUSED
    initialFunction: ptr to function (int)
    initialArgument: int
    systemStack: array [SYSTEM_STACK_SIZE] of int
    isUserThread: bool
    userRegs: array [15] of int          -- Space for r1..r15
    myProcess: ptr to ProcessControlBlock
```

Class Thread

methods

```
    Init (n: ptr to array of char)
    Fork (fun: ptr to function (int), arg: int)
    Yield ()
    Sleep ()
    CheckOverflow ()
    Print ()
```

Class ProcessControlBlock

```
fields
    pid: int                      -- The process ID
    parentsPid: int               -- The pid of the parent of this process
    status: int                   -- ACTIVE, ZOMBIE, or FREE
    myThread: ptr to Thread       -- Each process has one thread
    exitStatus: int               -- The value passed to Sys_Exit
    addrSpace: AddrSpace          -- The logical address space
    fileDescriptor: array [MAX_FILES_PER_PROCESS]
                              of ptr to OpenFile
```

Class ProcessControlBlock

methods

```
    Init ()
    Print ()
```

Class ThreadManager

```
fields
    threadTable: array [MAX_NUMBER_OF_PROCESSES]
                                           of Thread
    freeList: List [Thread]
    threadManagerLock: Mutex
                           -- These synchronization objects
    aThreadBecameFree: Condition
                           --    apply to the "freeList"
```

Class ThreadManager

methods

```
    Init ()
    Print ()
    GetANewThread () returns ptr to Thread
    FreeThread (th: ptr to Thread)
```

Class ProcessManager

```
fields
    processTable: array [MAX_NUMBER_OF_PROCESSES]
                          of ProcessControlBlock
    processManagerLock: Mutex
                          -- These synchronization objects
    aProcessBecameFree: Condition
                          --    apply to the "freeList"
    freeList: List [ProcessControlBlock]
    aProcessDied: Condition
                          -- Signalled for new ZOMBIEs
    nextPid: int
```

Class ProcessManager

```
methods
    Init ()
    Print ()
    GetANewProcess () returns ptr to ProcessControlBlock
    -- FreeProcess (p: ptr to ProcessControlBlock)
    TurnIntoZombie (p: ptr to ProcessControlBlock)
    WaitForZombie (proc: ptr to ProcessControlBlock)
                                   returns int
```

Class FileControlBlock

```
fields
    fcbID: int
    numberOfUsers: int
                      -- count of OpenFiles pointing here
    startingSectorOfFile: int      -- or -1 if FCB not in use
    sizeOfFileInBytes: int
    bufferPtr: ptr to void         -- ptr to a page frame
    relativeSectorInBuffer: int    -- or -1 if none
    bufferIsDirty: bool    -- Set to true when buffer is modified
```

Class FileControlBlock

methods

- **Init** ()
- **Print** ()

Class OpenFile

```
fields
    currentPos: int                   -- 0 = first byte of file
    fcb: ptr to FileControlBlock   -- null = not open
    numberOfUsers: int       -- count of Processes pointing here
```

Class OpenFile

```
methods
    Print ()
    SynchRead (targetAddr, numBytes: int) returns bool
                        -- returns true if all okay
    ReadInt () returns int
    LoadExecutable (addrSpace: ptr to AddrSpace) returns int
                                                  -- returns -1 if problems
```

Class FileManager

```
fields
    fileManagerLock: Mutex

    fcbTable: array [MAX_NUM_FILE_CONTROL_BLKS]
                          of FileControlBlock
    anFCBBecameFree: Condition
    fcbFreeList: List [FileControlBlock]

    openFileTable: array [MAX_NUM_OPEN_FILES]
                          of OpenFile
    anOpenFileBecameFree: Condition
    openFileFreeList: List [OpenFile]

    directoryFrame: ptr to void
```

Class FileManager

```
methods
    Init ()
    Print ()
    FindFCB (filename: String) returns
            ptr to FileControlBlock       -- null if errors
    Open (filename: String) returns ptr to OpenFile
                                          -- null if errors
    Close (open: ptr to OpenFile)
    Flush (open: ptr to OpenFile)
    SynchRead (open: ptr to OpenFile,
      targetAddr, bytePos, numBytes: int) returns bool
    SynchWrite (open: ptr to OpenFile,
      targetAddr, bytePos, numBytes: int) returns bool
```

Class DiskDriver

fields

DISK_STATUS_WORD_ADDRESS: ptr to int
DISK_COMMAND_WORD_ADDRESS: ptr to int
DISK_MEMORY_ADDRESS_REGISTER: ptr to int
DISK_SECTOR_NUMBER_REGISTER: ptr to int
DISK_SECTOR_COUNT_REGISTER: ptr to int
semToSignalOnCompletion: ptr to Semaphore
semUsedInSynchMethods: Semaphore
diskBusy: Mutex

Class DiskDriver

```
methods
    Init ()
    SynchReadSector
      (sectorAddr, numberOfSectors, memoryAddr: int)
    StartReadSector
      (sectorAddr, numberOfSectors, memoryAddr: int,
                      whoCares: ptr to Semaphore)
    SynchWriteSector
      (sectorAddr, numberOfSectors, memoryAddr: int)
    StartWriteSector
      (sectorAddr, numberOfSectors, memoryAddr: int,
              whoCares: ptr to Semaphore)
```

TestProgram3.c

```
function main
     -- SysExitTest ()
     -- BasicForkTest ()
     -- YieldTest ()
     -- ForkTest ()
     -- JoinTest1 ()
     -- JoinTest2 ()
     -- JoinTest3 ()
     -- ManyProcessesTest1 ()
     -- ManyProcessesTest2 ()
     -- ManyProcessesTest3 ()
     -- ErrorTest ()
...etc...
```

The “Exit Status” Problem

Problem:

A process exits, providing a return code (exit status)

The parent process may need that exit status

Parent calls “Sys_Join ()” ... But not until later

Need a place to store this number.

(Must keep exitStatus connected with its process pid)

Keep it in the PCB

Don’t free the PCB immediately

Zombies

Solution:

Processes become "zombies" before getting freed

Process Status:

- ACTIVE
  - normal (ready/running/blocked)
- ZOMBIE
  - all resources (except PCB) are freed
    - no thread!
  - some other thread will free the PCB
  - will have no zombie children
- FREE
  - iff the PCB is on the free list

Handle_Sys_Join

```
function Handle_Sys_Join (processID: int) returns int

    -- Set “child” to point to the right PCB…
    child = ... (Search through the processTable)

    -- Make sure we found something…
    if child == null...

    -- Make sure it really is a child of this process…
    if child.parentsPid != ...

    -- Check its status…
    if child.status == FREE then FatalError…

    -- Wait for it to terminate, get its exit code and return it…
    childsExitStatus = processManager.WaitForZombie (child)
    return childsExitStatus
```

ProcessManager.WaitForZombie

This method is passed a ptr to a process. It waits for that process to turn into a zombie. Then it saves its exitStatus and adds it back to the free list. It returns the exitStatus.

ProcessManager.WaitForZombie

```
processManagerLock.Lock ()

while proc.status != ZOMBIE
    aProcessDied.Wait (& processManagerLock)
endWhile

i = proc.exitStatus
proc.status = FREE
freeList.AddToEnd (proc)
aProcessBecameFree.Signal (& processManagerLock)

processManagerLock.Unlock ()
return i
```

You must implement ProcessFinish

function ProcessFinish (exitStatus: int)

The implentation of Handle_Sys_Exit...

```
function Handle_Sys_Exit (returnStatus: int)
    ProcessFinish (returnStatus)
endFunction
```

NOTE: Kernel also calls **ProcessFinish** *whenever it needs to kill a UserProcess, e.g., AddressException, etc.*

ProcessFinish - Part 1

```
-- Save the exit code...
currentThread.myProcess.exitStatus = exitStatus

-- Disconnent the PCB from the Thread ...
ignore = SetInterruptsTo (DISABLED)
currentThread.myProcess = null
proc.myThread = null
currentThread.isUserThread = false
ignore = SetInterruptsTo (ENABLED)
```

(continued)

ProcessFinish - Part 2

```
-- Close any open files...
for i = 0 to MAX_FILES_PER_PROCESS-1
    open = proc.fileDescriptor [i]
    if open != null
        fileManager.Close (open)
    endIf
endFor
-- Return all frames to the pool...
frameManager.ReturnAllFrames (& proc.addrSpace)
-- Turn this Process into a ZOMBIE...
processManager.TurnIntoZombie (proc)
-- Terminate this thread; parent will deal with the ZOMBIE...
ThreadFinish ()
```

ProcessManager.TurnIntoZombie

This method is passed a ptr to a process; It turns it into a zombie - dead but not gone! - so that its exitStatus can be retrieved if needed by its parent.

ProcessManager.TurnIntoZombie

```
processManagerLock.Lock ()

-- Get rid of any ZOMBIE children...
 for i = 0 to MAX_NUMBER_OF_PROCESSES-1
    child = & processTable[i]
    if child.parentsPid == p.pid && child.status == ZOMBIE
       child.status = FREE
       freeList.AddToEnd (child)
       aProcessBecameFree.Signal (& processManagerLock)
    endIf
endFor
```

(continued)

ProcessManager.TurnIntoZombie

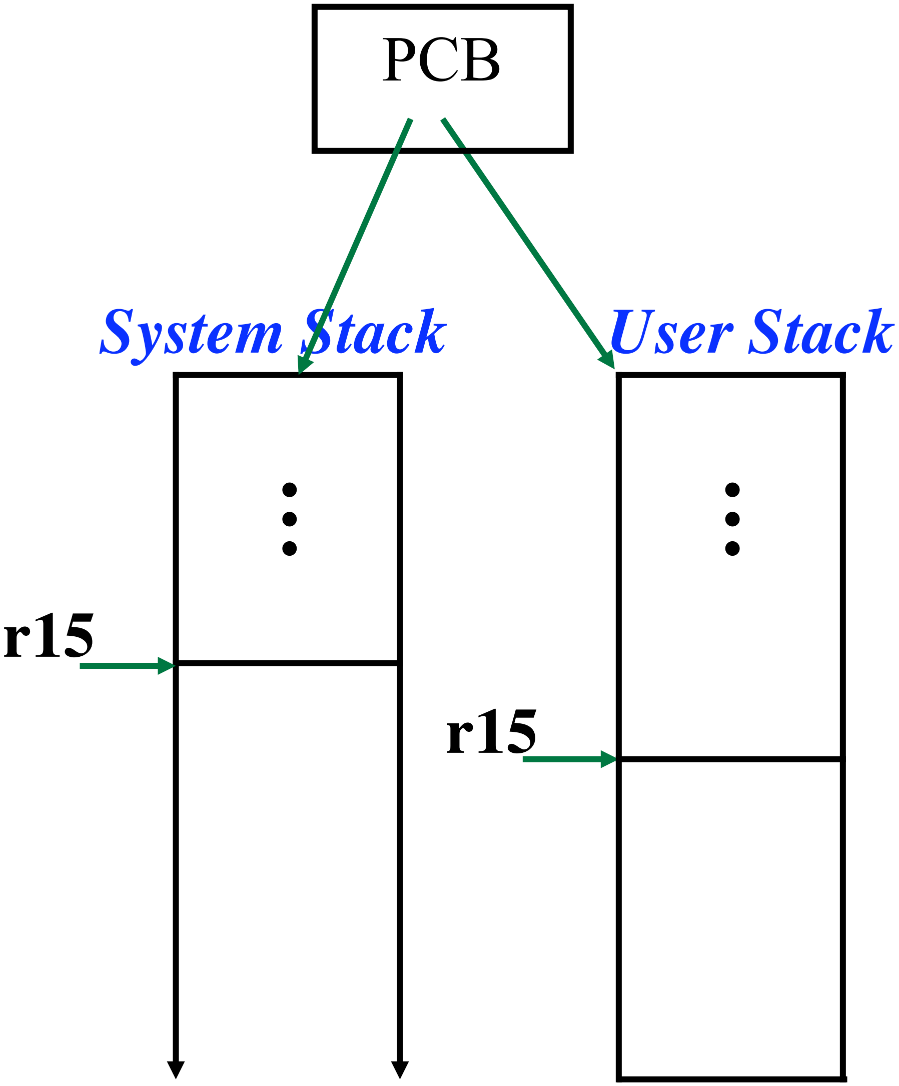
```
-- Set "parent" to point to our parent (or null if none)...
parent = null
for i = 0 to MAX_NUMBER_OF_PROCESSES-1
    if processTable[i].pid == p.parentsPid
            parent = & processTable[i]
    endIf
endFor
```

(continued)

ProcessManager.TurnIntoZombie

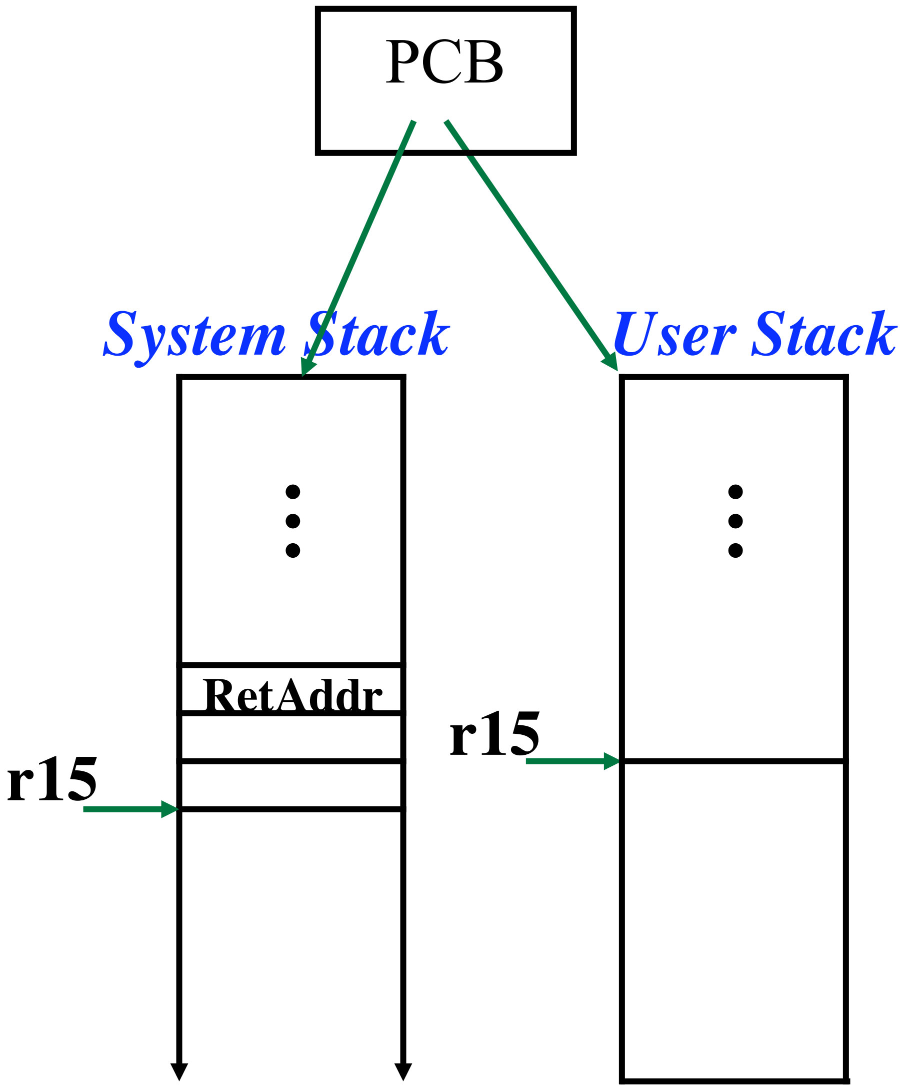
```
-- If our parent is ACTIVE, we must become a ZOMBIE!!!
if parent && parent.status == ACTIVE
    -- Turn into a ZOMBIE and let everyone who cares know it
    p.status = ZOMBIE
    aProcessDied.Broadcast (& processManagerLock)
else
    -- Go straight to grave; no one will wait for our exitStatus
    p.status = FREE
    freeList.AddToEnd (p)
    aProcessBecameFree.Signal (& processManagerLock)
endIf

processManagerLock.Unlock ()
```

The Fork Syscall



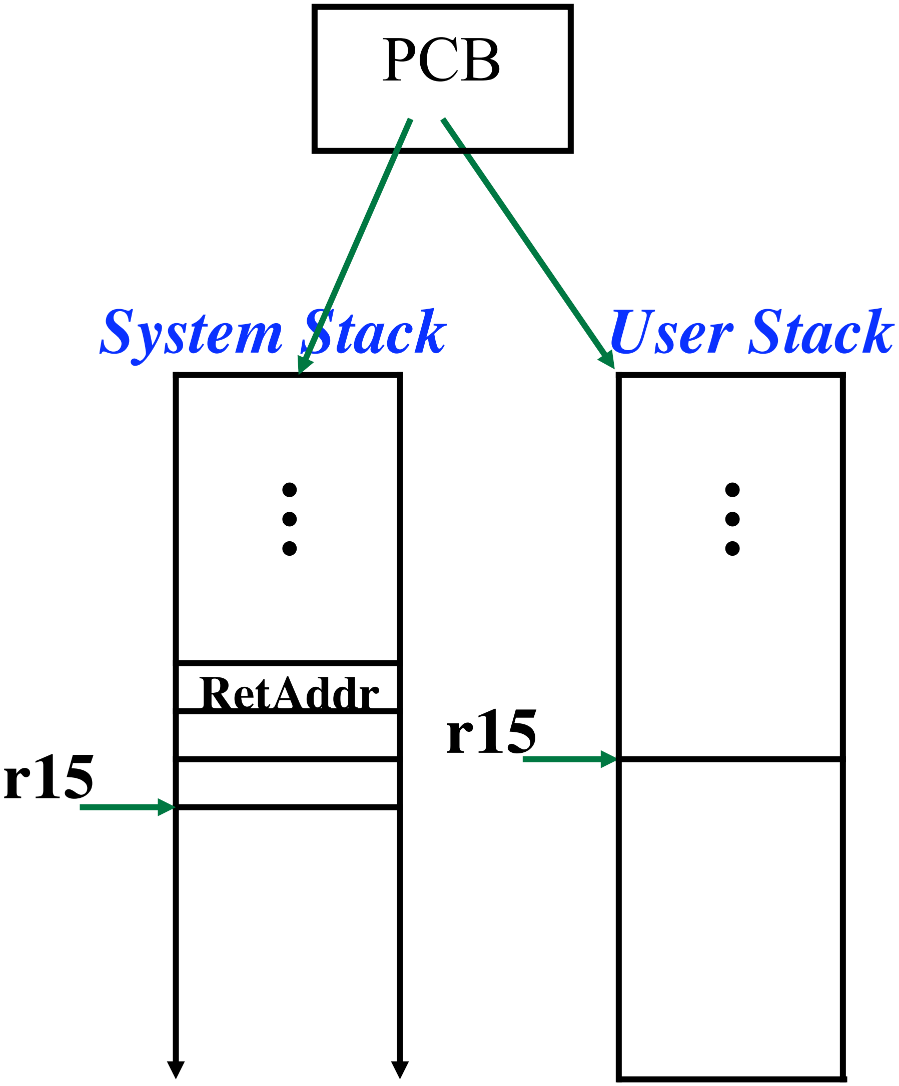
Syscall interupt occurs

The Fork Syscall



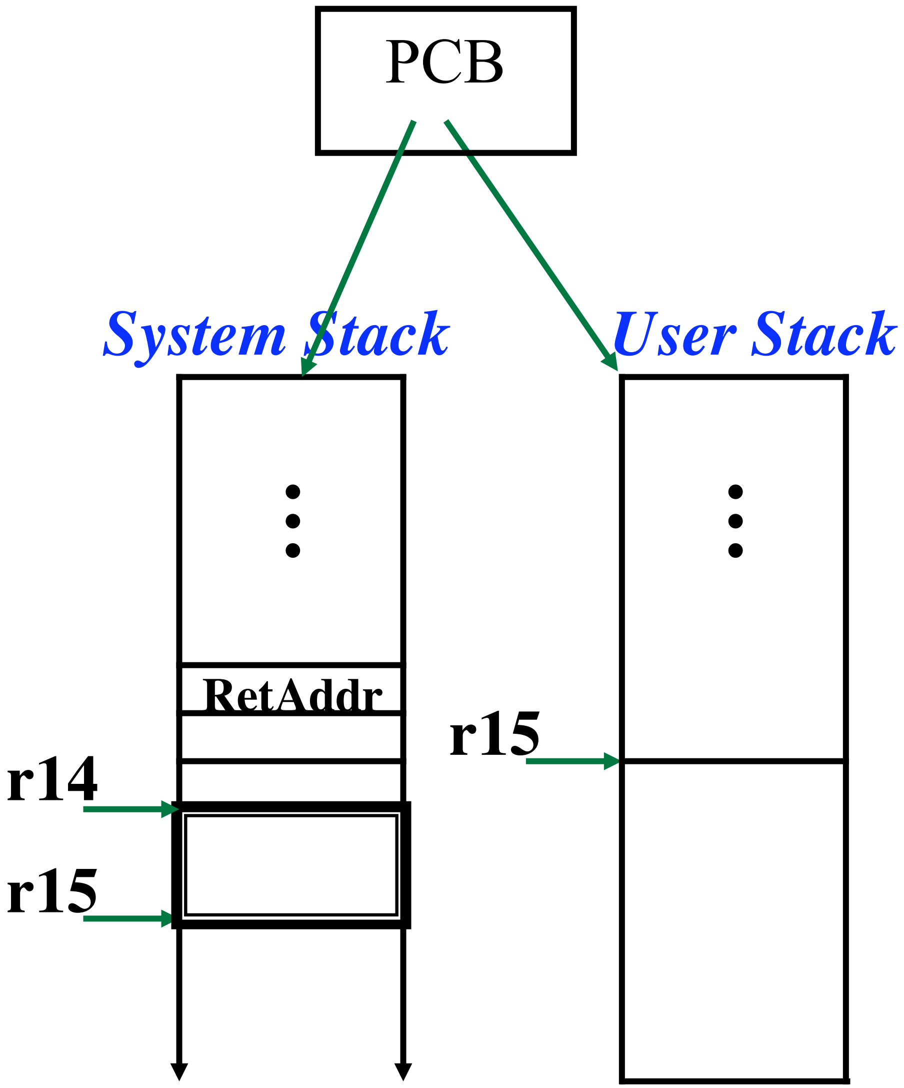
Syscall interupt occurs

The Fork Syscall



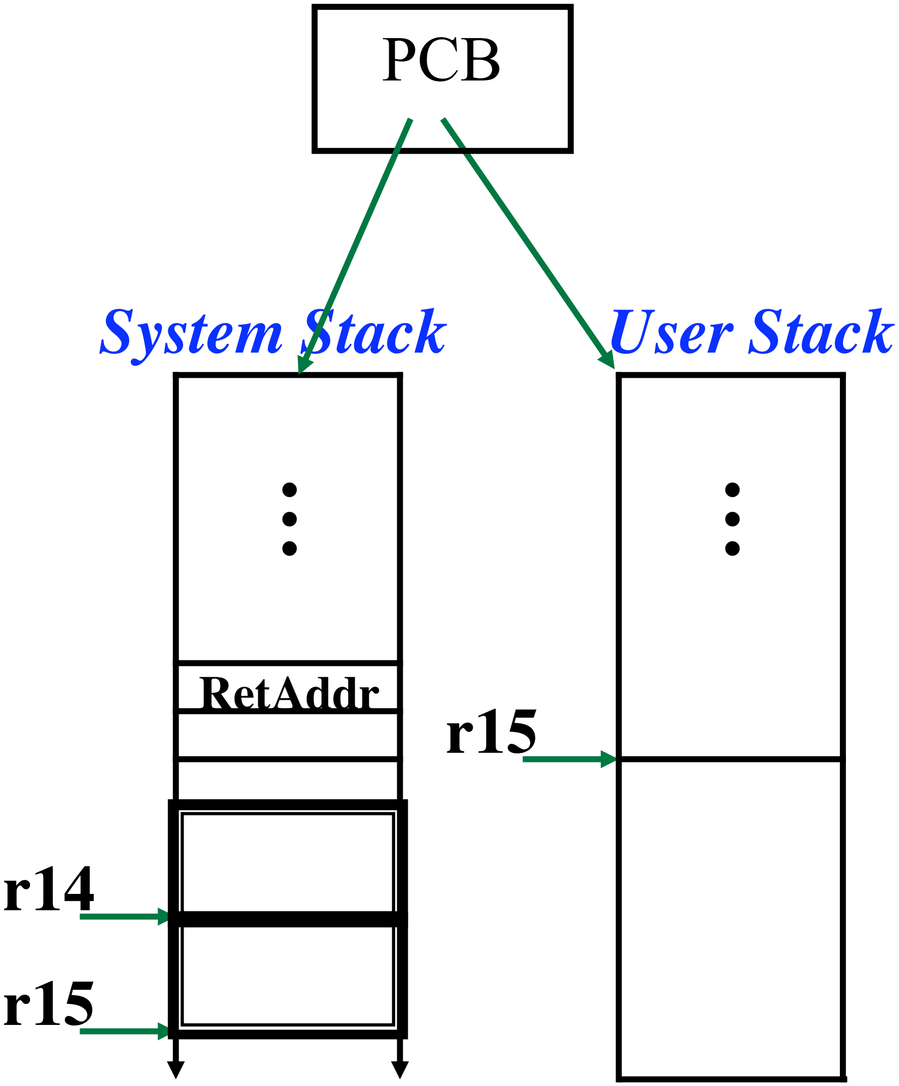
Jump to assembly routine "SyscallTrapHandler"

The Fork Syscall



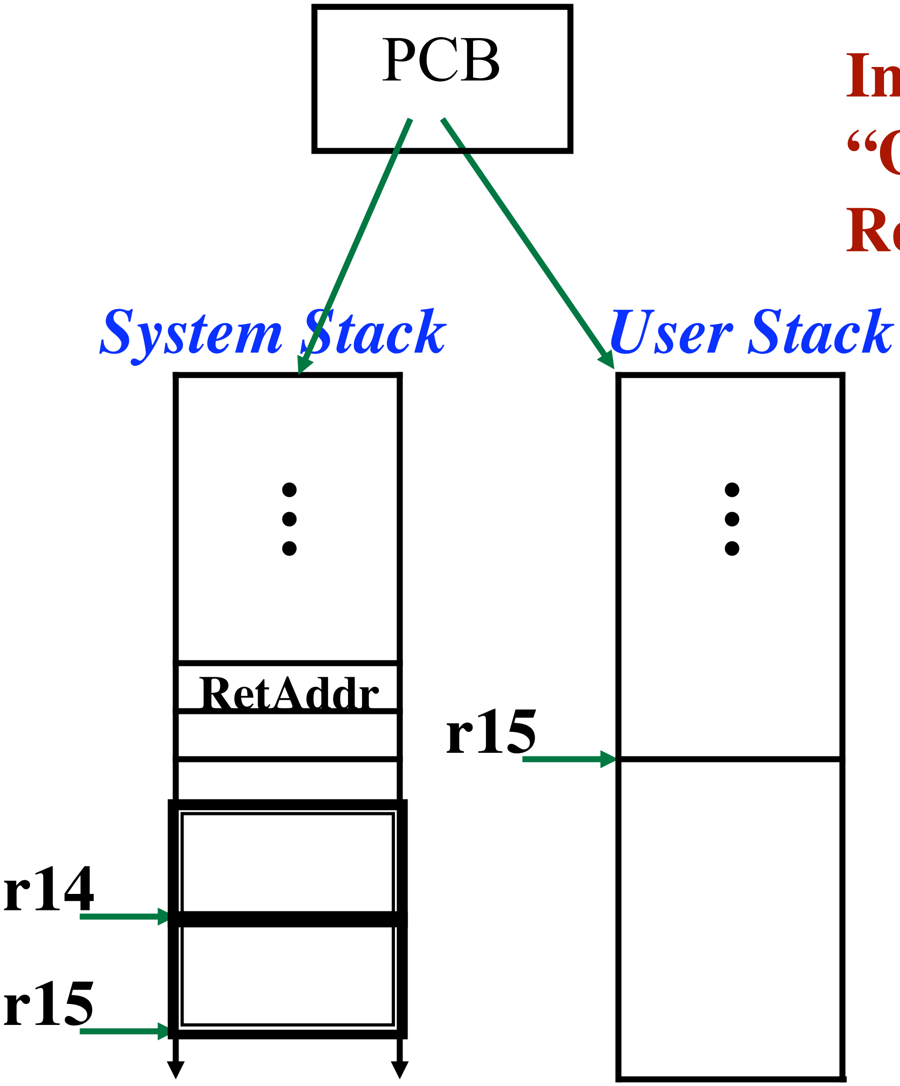
Calls kernel routine "SyscallTrapHandler"

The Fork Syscall



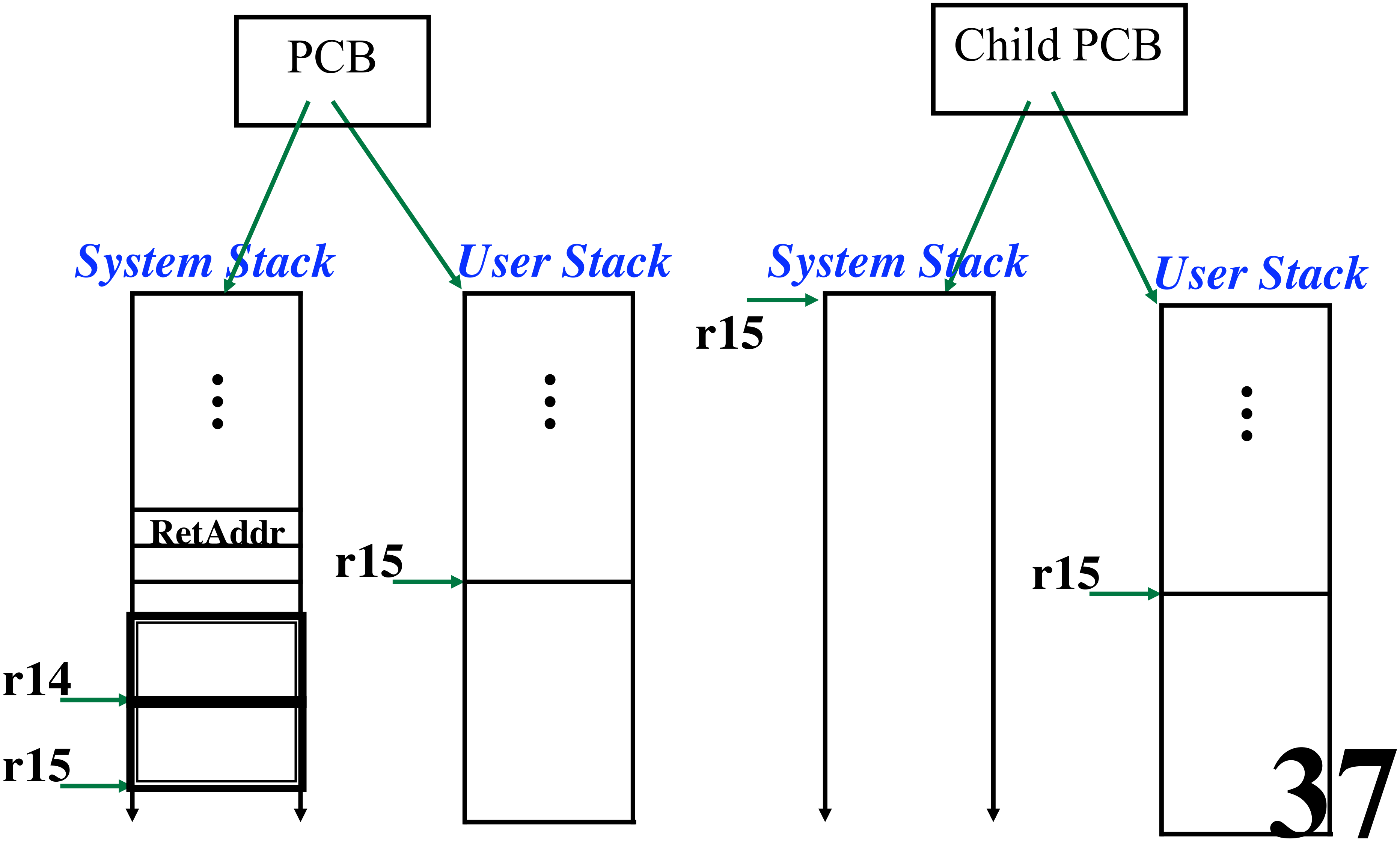
Invoke
“Handle_Sys_Fork”

The Fork Syscall



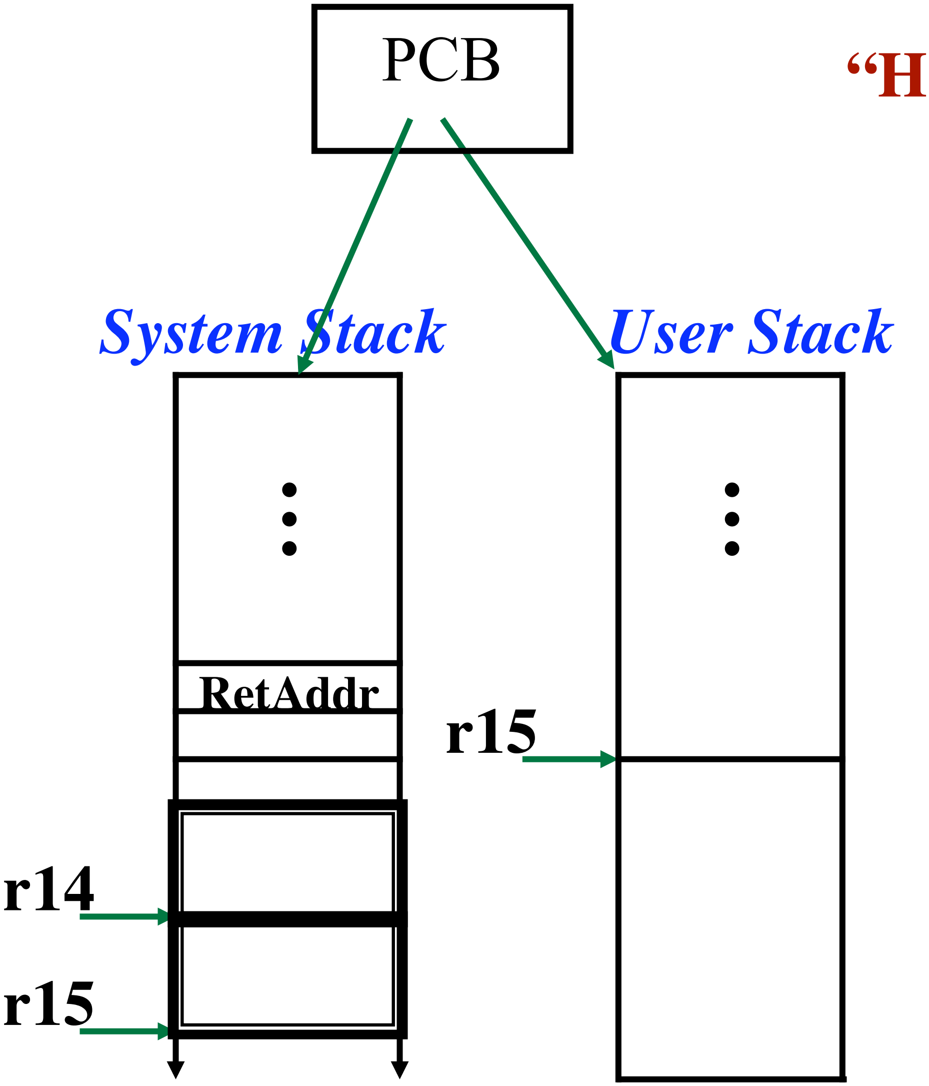
Invoke "GetOldUserPCFromSystemStack"
Retrieves "RetAddr" from stack

The Fork Syscall

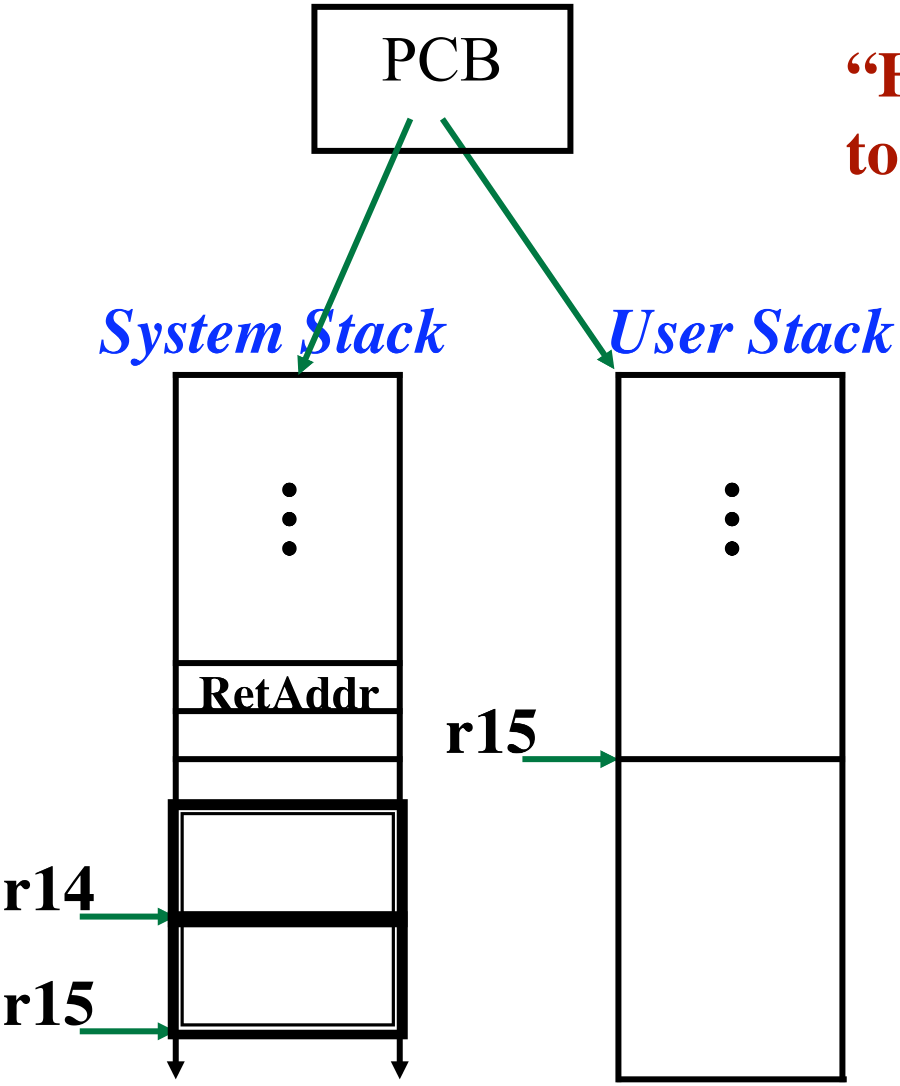


The Fork Syscall

“Handle_Sys_Fork” continues

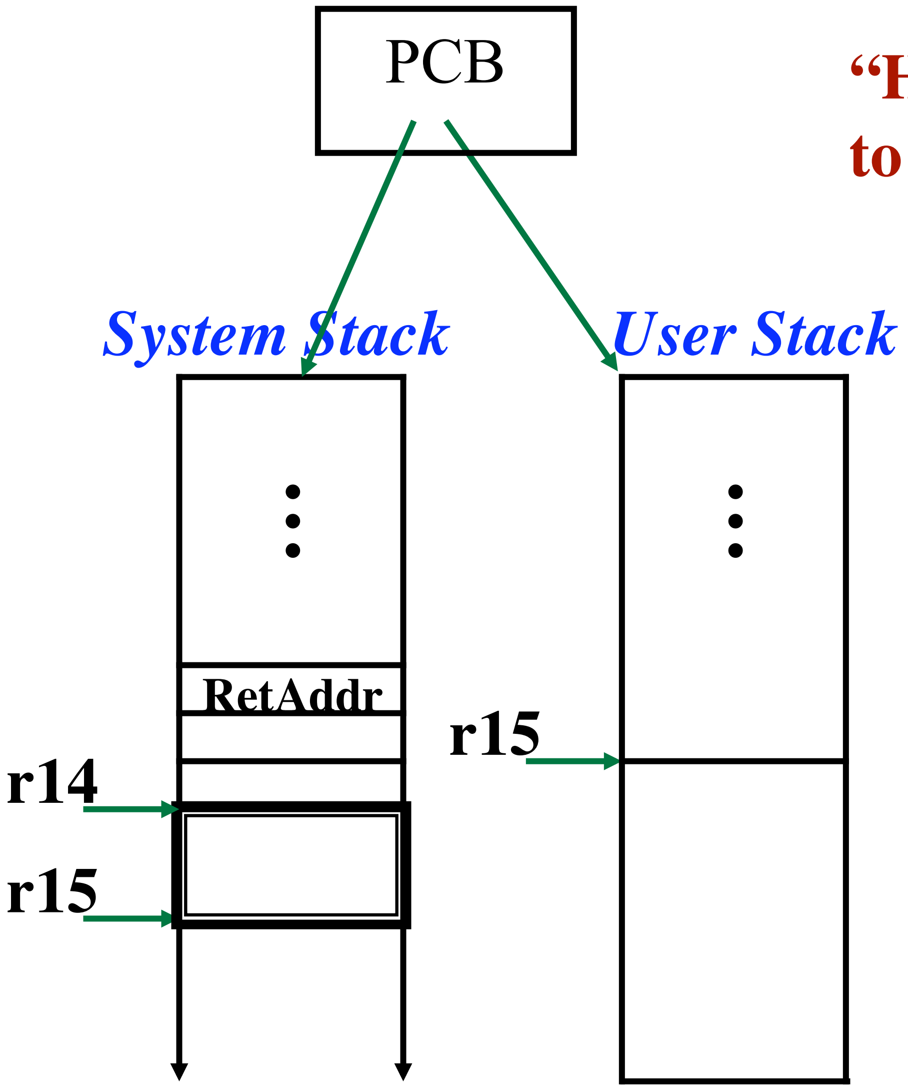


The Fork Syscall



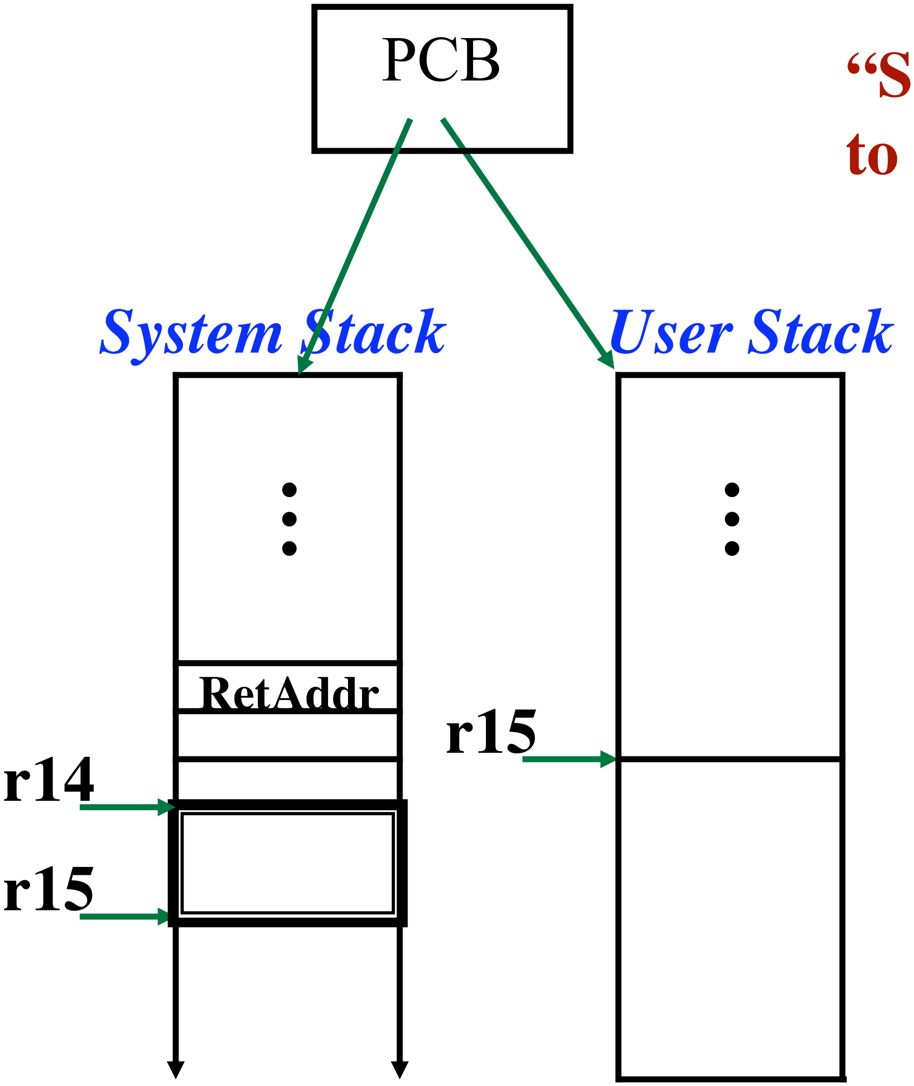
“Handle_Sys_Fork” returns to “SyscallTrapHandler”

The Fork Syscall



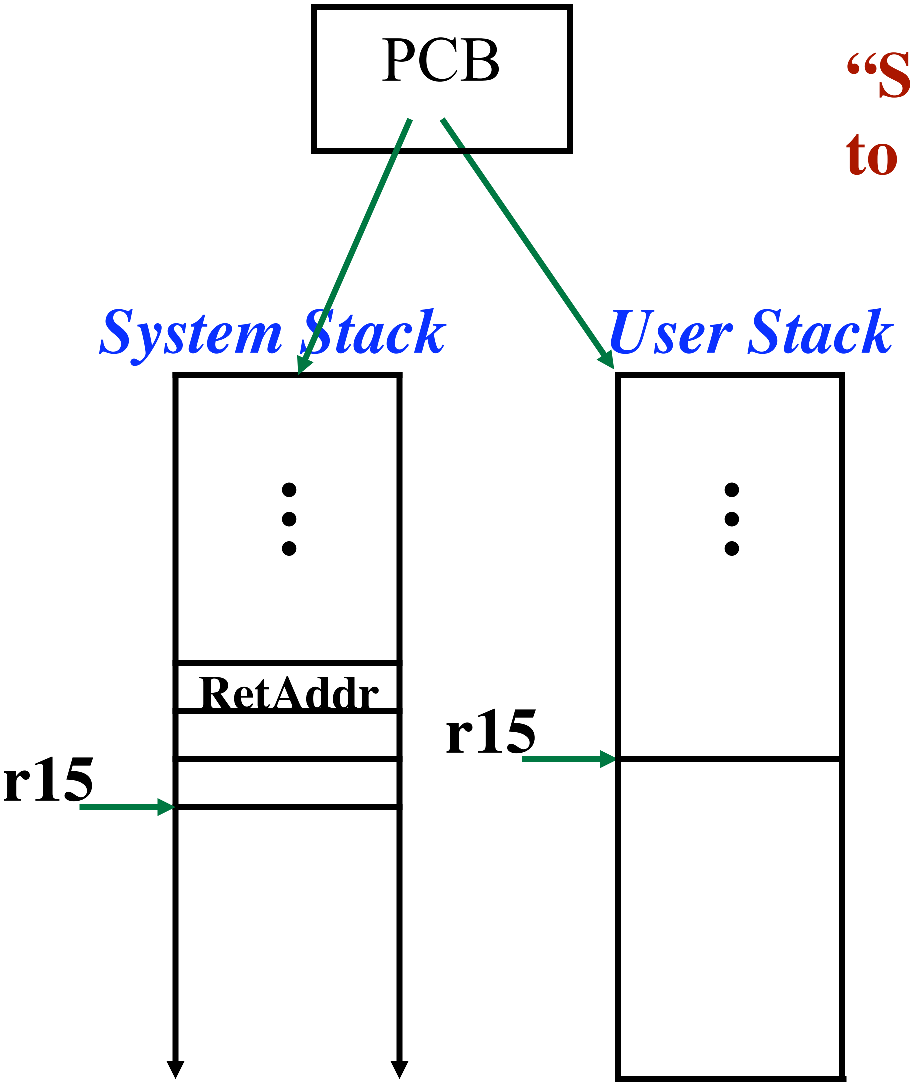
“Handle_Sys_Fork” returns to “SyscallTrapHandler”

The Fork Syscall



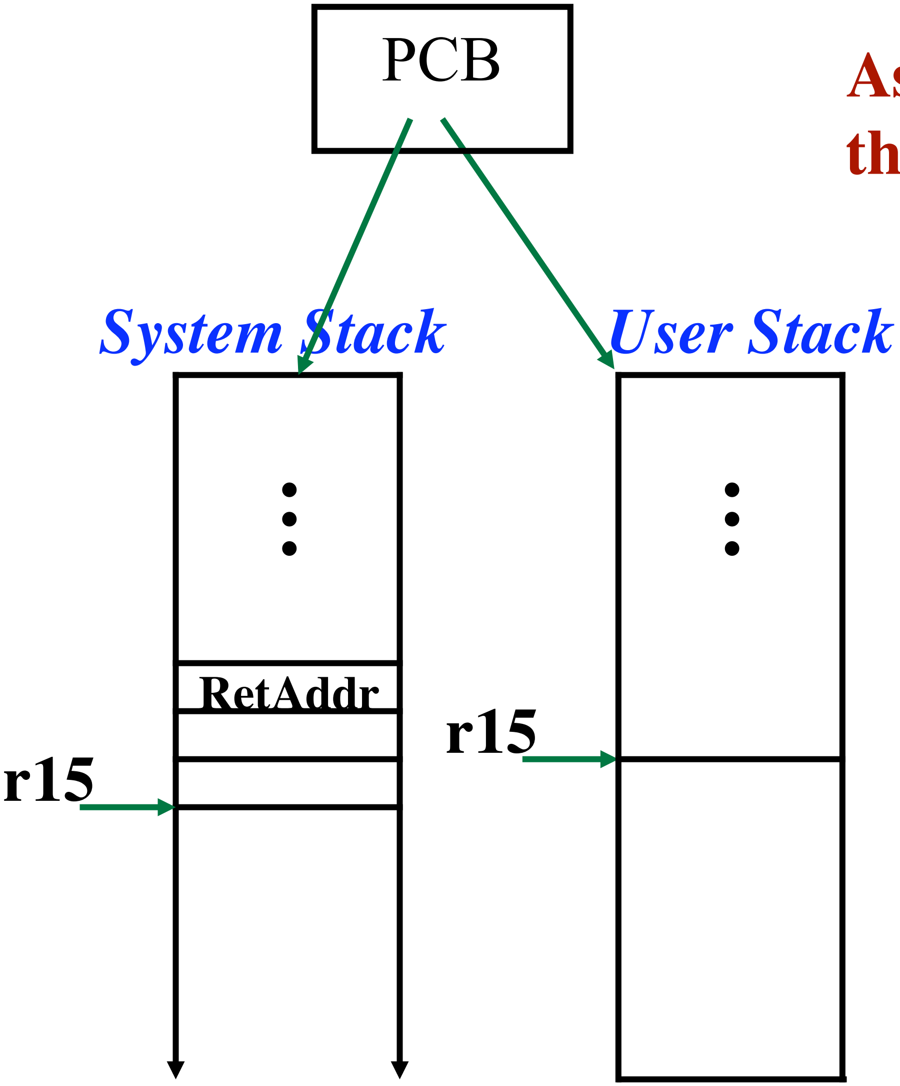
“SyscallTrapHandler” returns to assembly routine

The Fork Syscall



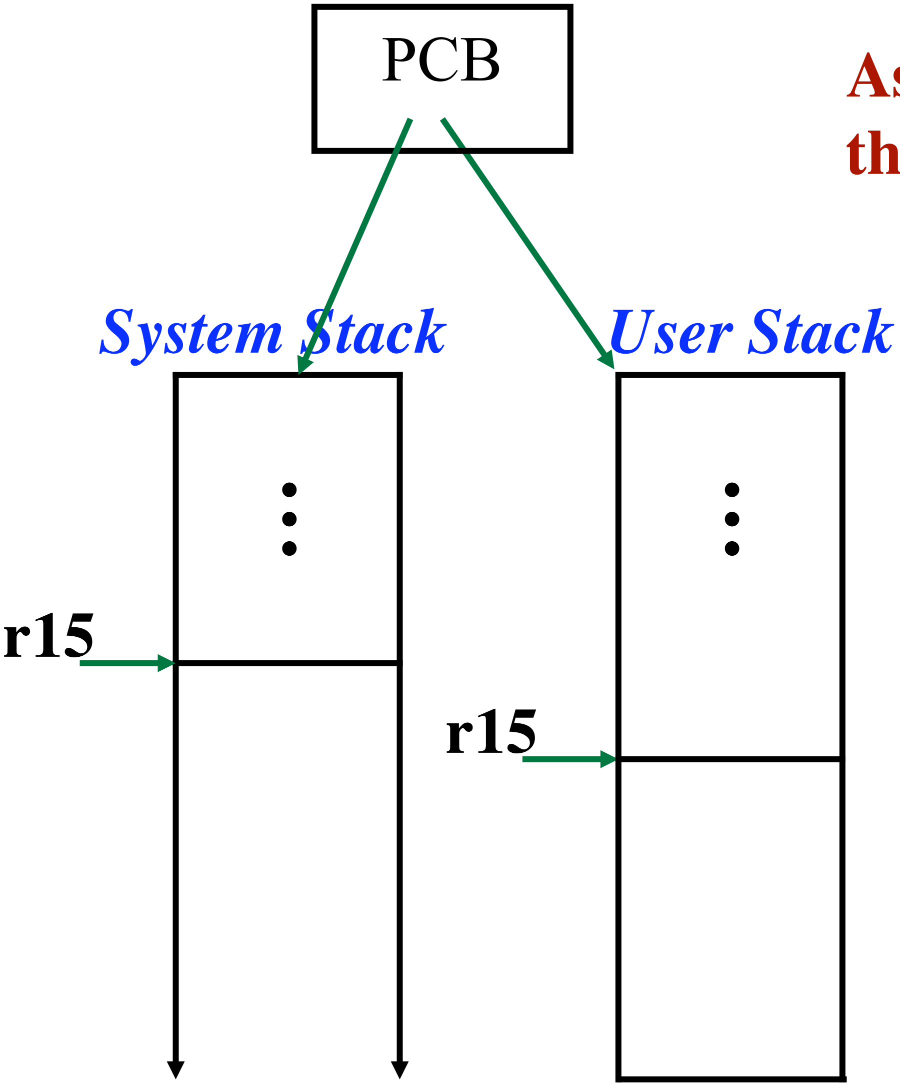
“SyscallTrapHandler” returns to assembly routine

The Fork Syscall



Assembly routine executes the “reti” instruction

The Fork Syscall



Assembly routine executes the “reti” instruction

Handle_Sys_Fork

```
-- Get a new PCB and Thread
newPCB = ...
newThread = ...
-- Initialize PCB
newPCB.myThread = ...
newPCB.parentsPID = ...
-- Initialize Thread
newThread.name = ...
newThread.status = ...
newThread.myProcess = ...
```

(continued)

Handle_Sys_Fork

```
-- Must share OpenFiles with parent...
fileManager.fileManagerLock.Lock ()

for i = 0 to ...
    newPCB.fileDescriptor [i] = oldPCB.fileDescriptor [i]
    if not null, then increment OpenFile.numberOfUsers
endFor

fileManager.fileManagerLock.Unlock ()
```

(continued)

Handle_Sys_Fork

```
-- This is a running user thread, which means its user-register
-- data is in the registers (not saved); grab these values...
SaveUserRegs (&newThread.userRegs[0])

-- Reset the system stack top...
newThread.stackTop =
    & (newThread.systemStack[SYSTEM_STACK_SIZE-1])

-- No other threads will touch our user stack or the new stack,
-- so okay to let other threads run...
ignore = SetInterruptsTo (ENABLED)
```

(continued)

Handle_Sys_Fork

```
-- Allocate new frames for this address space...
frameManager.GetNewFrames (& newPCB.addrSpace,
    oldPCB.addrSpace.numberOfPages)
```

(continued)

Handle_Sys_Fork

```
-- Copy all pages...
for i = 0 to oldPCB.addrSpace.numberOfPages-1
    if oldPCB.addrSpace.IsWritable (i)
       newPCB.addrSpace.SetWritable (i)
    else
       newPCB.addrSpace.ClearWritable (i)
    endIf
    MemoryCopy (
             newPCB.addrSpace.ExtractFrameAddr (i),
             oldPCB.addrSpace.ExtractFrameAddr (i),
             PAGE_SIZE)
endFor
```

(continued)

Handle_Sys_Fork

```
-- Get the User PC which is buried in the system stack
-- of the current process.  This value points to the instruction
-- following the syscall.  This is the place to which we
-- must return in the child.
oldUserPC = GetOldUserPCFromSystemStack ()

-- Fork a new thread and have it “resume” execution
-- in user-land.
newThread.Fork (ResumeChildAfterFork, oldUserPC)

-- Return child’s pid.
return newPCB.pid
```

ResumeChildAfterFork

Initial function of new thread.
Executes in system mode.
Completes the work of “fork”ing the child.
Assumes user stack and registers are already set up.

ResumeChildAfterFork

```
function ResumeChildAfterFork (initPC: int)

    -- Disable interrupts...
    ignore = SetInterruptsTo (DISABLED)

    -- Set the page table registers to point to this process's page table
    currentThread.myProcess.addrSpace.SetToThisPageTable ()

    -- Set the user registers
    RestoreUserRegs (&currentThread.userRegs[0])

    -- Any future interrupts will save the user regs to the Thread object
    currentThread.isUserThread = true
```

(continued)

ResumeChildAfterFork

```
      -- Reset system stack top
      initSystemStackTop =
         (& currentThread.systemStack[SYSTEM_STACK_SIZE-1])
                                                      asInteger

      -- Invoke BecomeUserThread to clear the "System Mode" bit
      -- set the user stack pointer and jump to 'initPC'
      BecomeUserThread (initUserStackTop,         -- Initial User Stack
                        initPC,                   -- Initial PC
                        initSystemStackTop)       -- Initial System Stack

   endFunction
```