Solution to the

Gaming Parlor

Programming Project

The Gaming Parlor - Solution

- ***Scenario:***
  - Front desk with dice (*resource units*)
  - Groups request (e.g., 5) dice (*They request resources*)
  - Groups must wait, if none available
  - A list of waiting groups... A "condition" variable
  - Dice are returned (*resources are released*)
  - The condition is signaled
  - The group checks and finds it needs to wait some more
  - The group (thread) waits...and goes to the end of the line
- ***Problem?***
-

The Gaming Parlor - Solution

- ***Scenario:***
  - Front desk with dice (*resource units*)
  - Groups request (e.g., 5) dice (*They request resources*)
  - Groups must wait, if none available
  - Dice are returned (*resources are released*)
  - A list of waiting groups… A "condition" variable
  - The condition is signalled
  - The group checks and finds it needs to wait some more
  - The group (thread) waits…and goes to the end of the line
- ***Problem?***
  - Starvation!

The Gaming Parlor - Solution

- ***Approach:***
  - Serve every group "first-come-first-served".

- ***Implementation:***
  - Keep the thread at the front of the line separate
  - "Leader" - the thread that is at the front of the line
  - Use 2 condition variables.
  - "Leader" will have at most one waiting thread
  - "RestOfLine" will have all other waiting threads

The Threads

```
function Group (numDice: int)
    var i: int
    for i = 1 to 5
      gameParlor.Acquire (numDice)
      currentThread.Yield ()
      gameParlor.Release (numDice)
      currentThread.Yield ()
    endFor
  endFunction


thA.Init ("A")
thA.Fork (Group, 4)
...
```

The Monitor

```
class GameParlor
   superclass Object
   fields
     monitorLock: Mutex
     leader: Condition
     restOfLine: Condition
     numberDiceAvail: int
     numberOfWaitingGroups: int
   methods
     Init ()
     Acquire (numNeeded: int)
     Release (numReturned: int)
     Print (str: String, count: int)
  endClass
```

The Release Method

```
method Release (numReturned: int)
   monitorLock.Lock ()

   -- Return the dice
   numberDiceAvail = numberDiceAvail + numReturned

   -- Print
   self.Print ("releases and adds back", numReturned)

   -- Wakeup the first group in line (if any)
   leader.Signal (&monitorLock)

   monitorLock.Unlock ()
endMethod
```

The Acquire Method

```
method Acquire (numNeeded: int)
   monitorLock.Lock ()
   -- Print
   self.Print ("requests", numNeeded)
   -- Indicate that we are waiting for dice.
   numberOfWaitingGroups = numberOfWaitingGroups + 1
   -- If there is a line, then get into it.
   if numberOfWaitingGroups > 1
        restOfLine.Wait (&monitorLock)
   endIf
   -- Now we're at the head of the line.  Wait until
                                      there are enough dice.
   while numberDiceAvail < numNeeded
        leader.Wait (&monitorLock)
   endWhile
   ...
```

The Acquire Method

```
   ...

   -- Take our dice.
   numberDiceAvail = numberDiceAvail - numNeeded

   -- Now we are no longer waiting; wakeup some other
                                  group and leave.
   numberOfWaitingGroups = numberOfWaitingGroups - 1
   restOfLine.Signal (&monitorLock)

   -- Print
   self.Print ("proceeds with", numNeeded)

   monitorLock.Unlock ()
endMethod
```

CS 333
Introduction to Operating Systems

Class 13 - Virtual Memory (3)

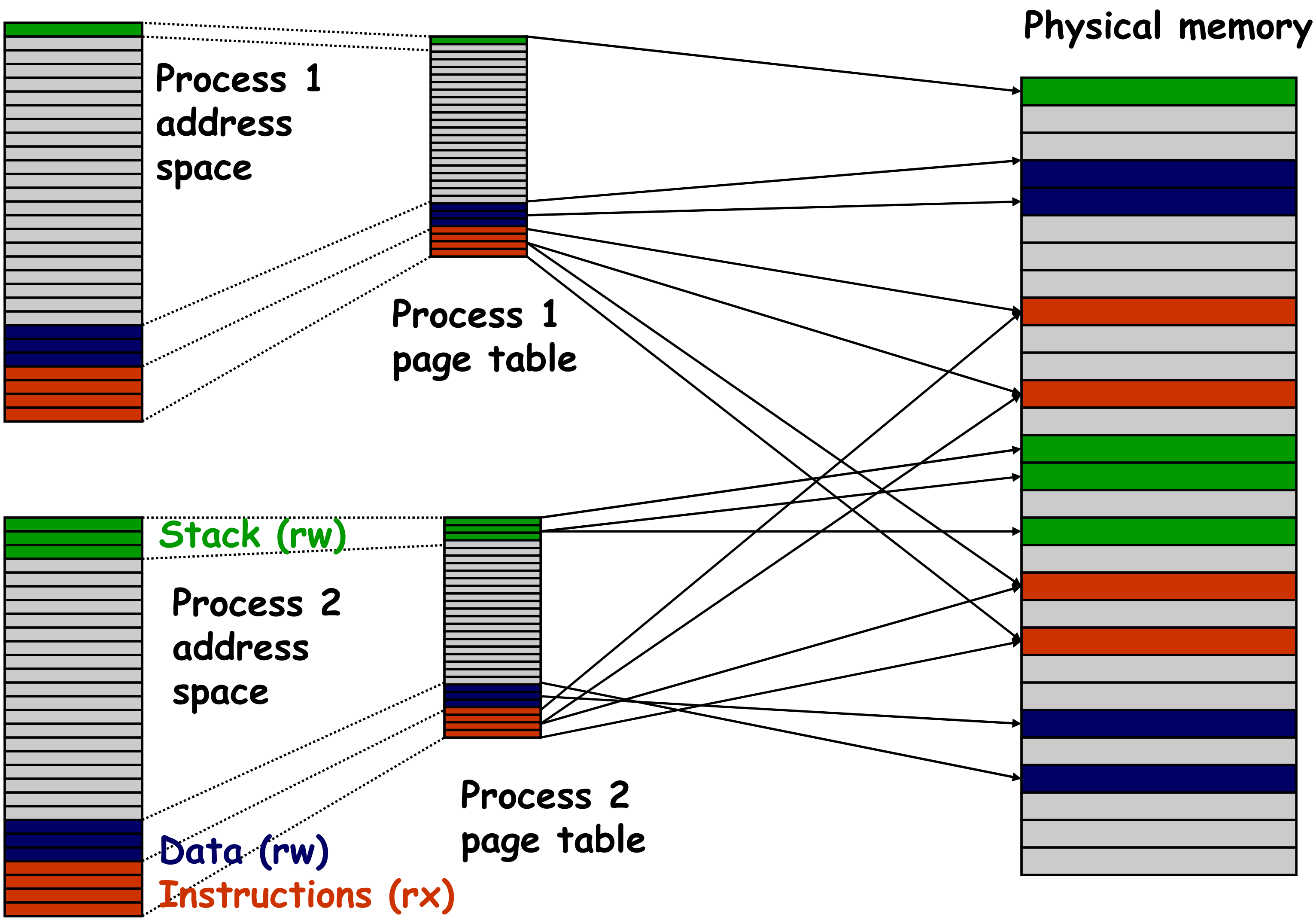
Jonathan Walpole
Computer Science
Portland State University

Page sharing

- **In a large multiprogramming system...**
  - Some users run the same program at the same time
    - **Why have more than one copy of the same page in memory???**

- ***Goal:***
  - Share pages among "processes" (not just threads!)
    - **Cannot share writable pages**
    - **If writable pages were shared processes would notice each other's effects**
    - **Text segment can be shared**

Page sharing



Page sharing

- **"Fork" system call**
  - Copy the parent's virtual address space
    - **... and immediately do an "Exec" system call**
    - **Exec overwrites the calling address space with the contents of an executable file (ie a new program)**
  - Desired Semantics:
    - **pages are copied, not shared**
  - Observations
    - **Copying every page in an address space is expensive!**
    - **processes can't notice the difference between copying and sharing unless pages are modified!**

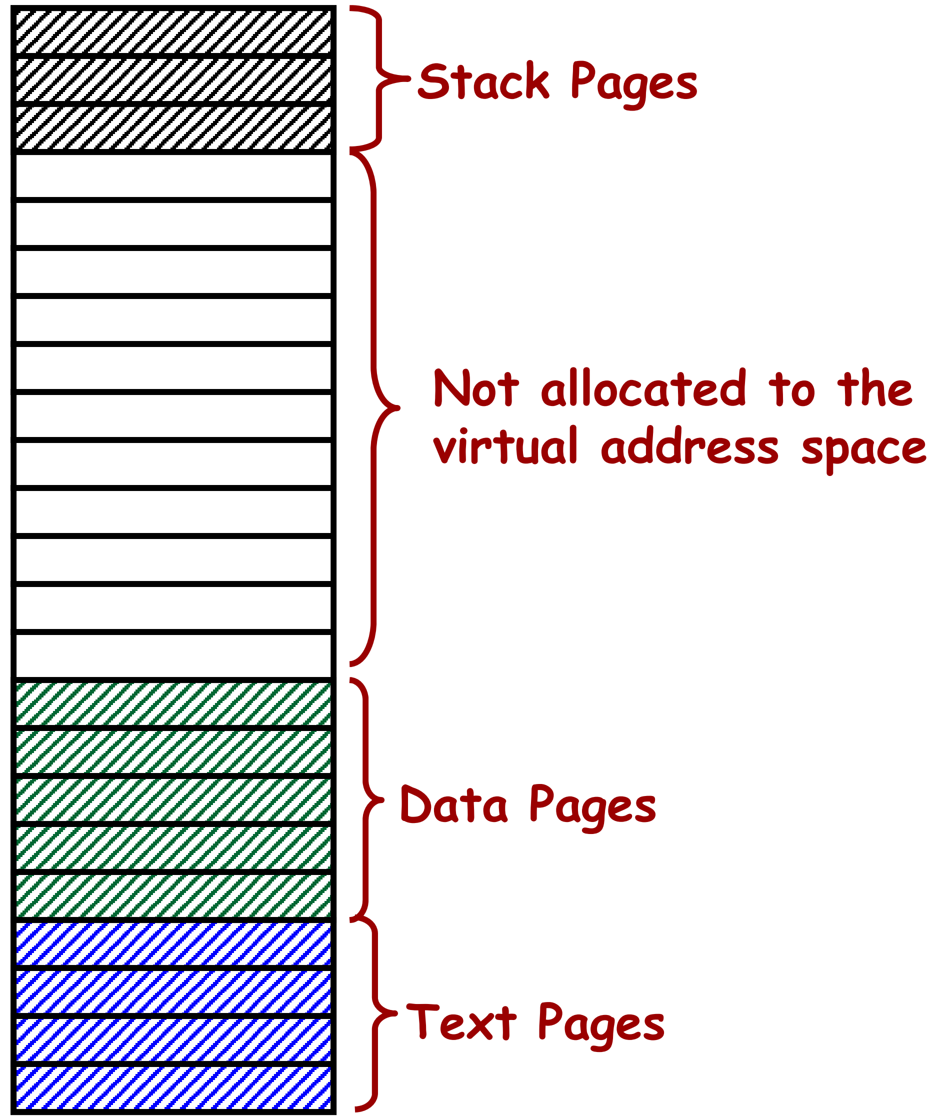
Page sharing

- **Idea: Copy-On-Write**
  - Initialize new page table, but point entries to existing page frames of parent
    - **Share pages**
  - Temporarily mark all pages "read-only"
    - **Share all pages until a protection fault occurs**
  - Protection fault (copy-on-write fault):
    - **Is this page really read only or is it writable but temporarily protected for copy-on-write?**
    - **If it is writable**
      - **copy the page**
      - **mark both copies "writable"**
      - **resume execution as if no fault occurred**

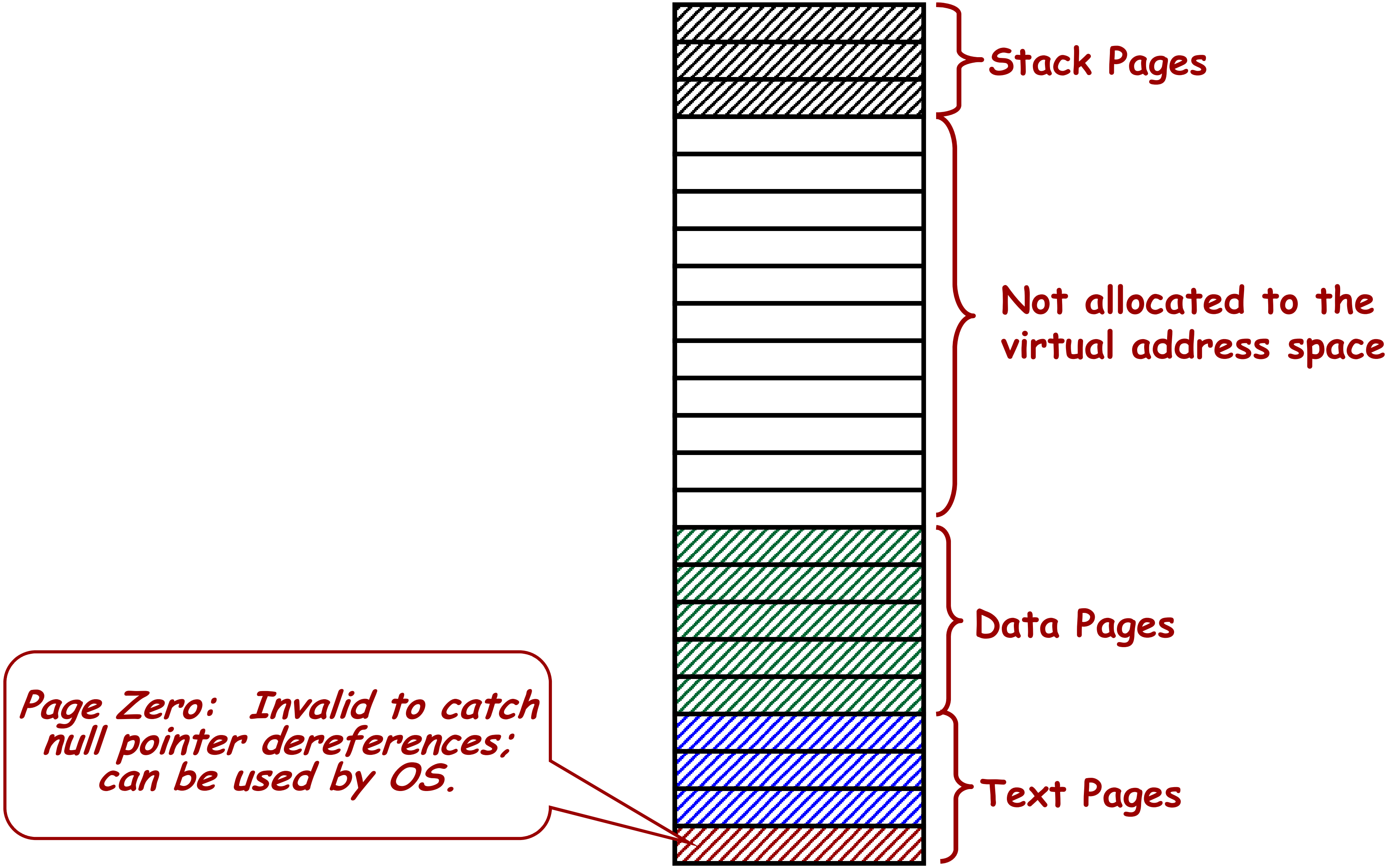
New System Calls for Page Management

- **Goal:**
  - Allow some processes more control over paging!
- **System calls added to the kernel**
  - A process can request a page before it is needed
    - **Allows processes to grow (heap, stack etc)**
  - Processes can share pages
    - **Allows fast communication of data between processes**
    - **Similar to how threads share memory**
      - **… so what is the difference?**

Unix processes

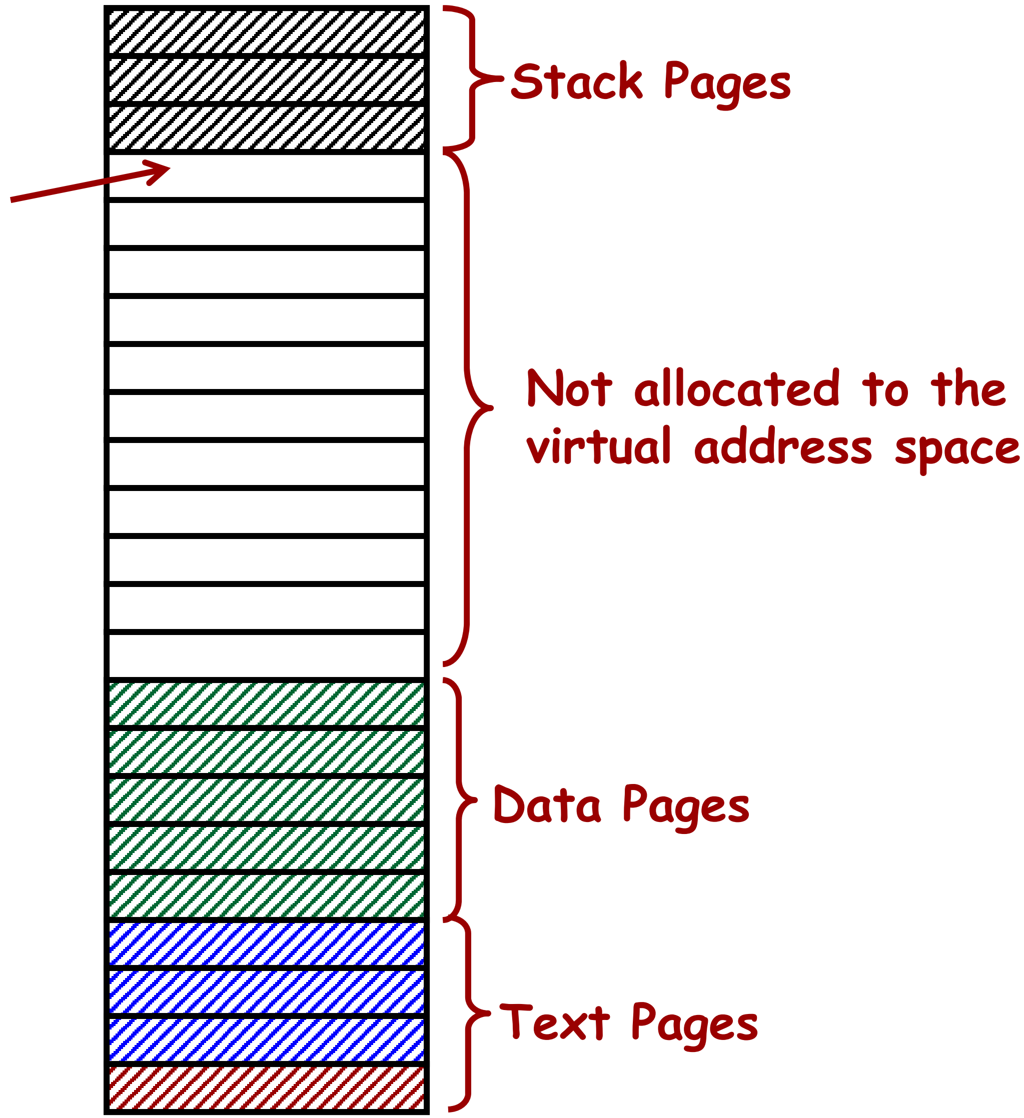


Unix processes



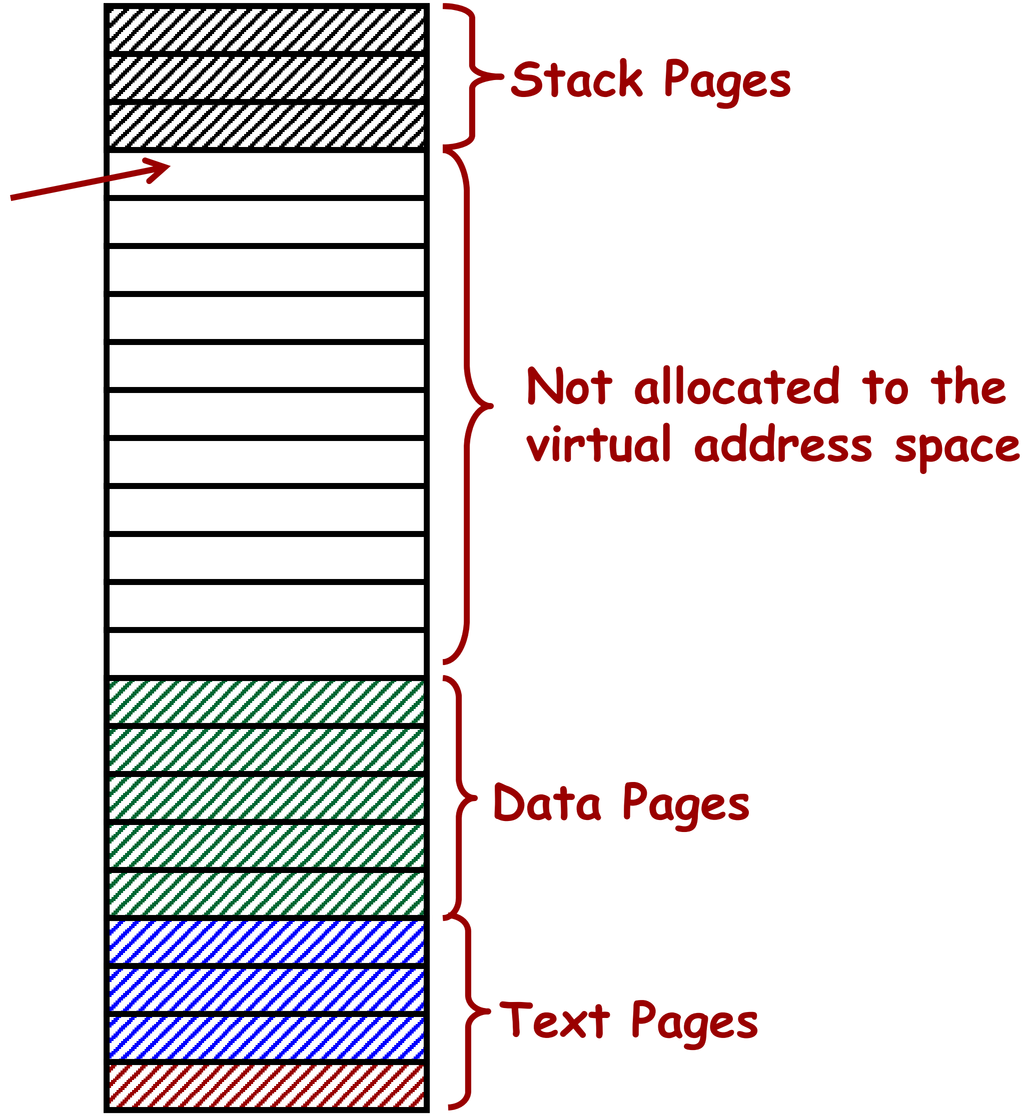
Unix processes

The stack grows;
Page requested here



Unix processes

The stack grows;
Page requested here
A new page is allocated
    and process continues

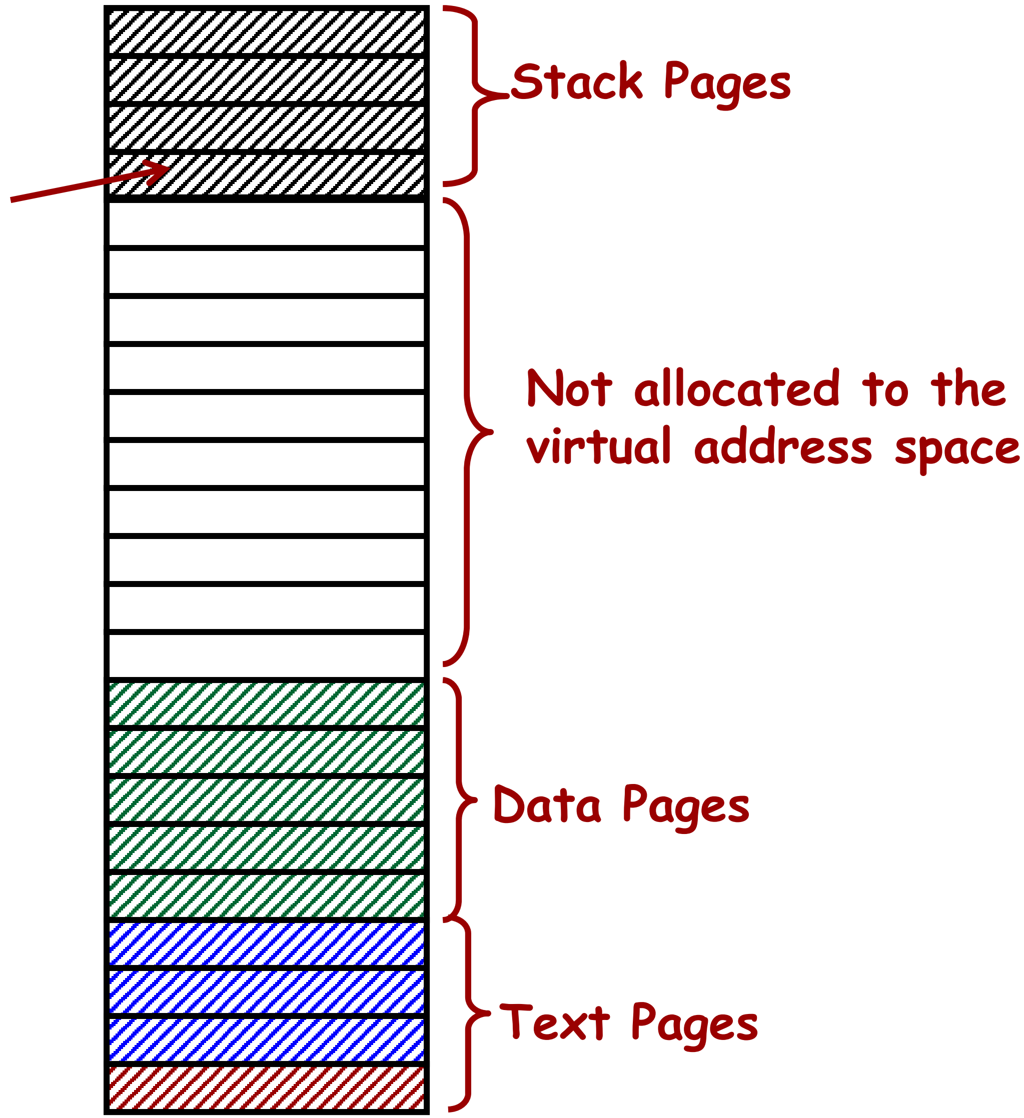


Unix processes

The stack grows;

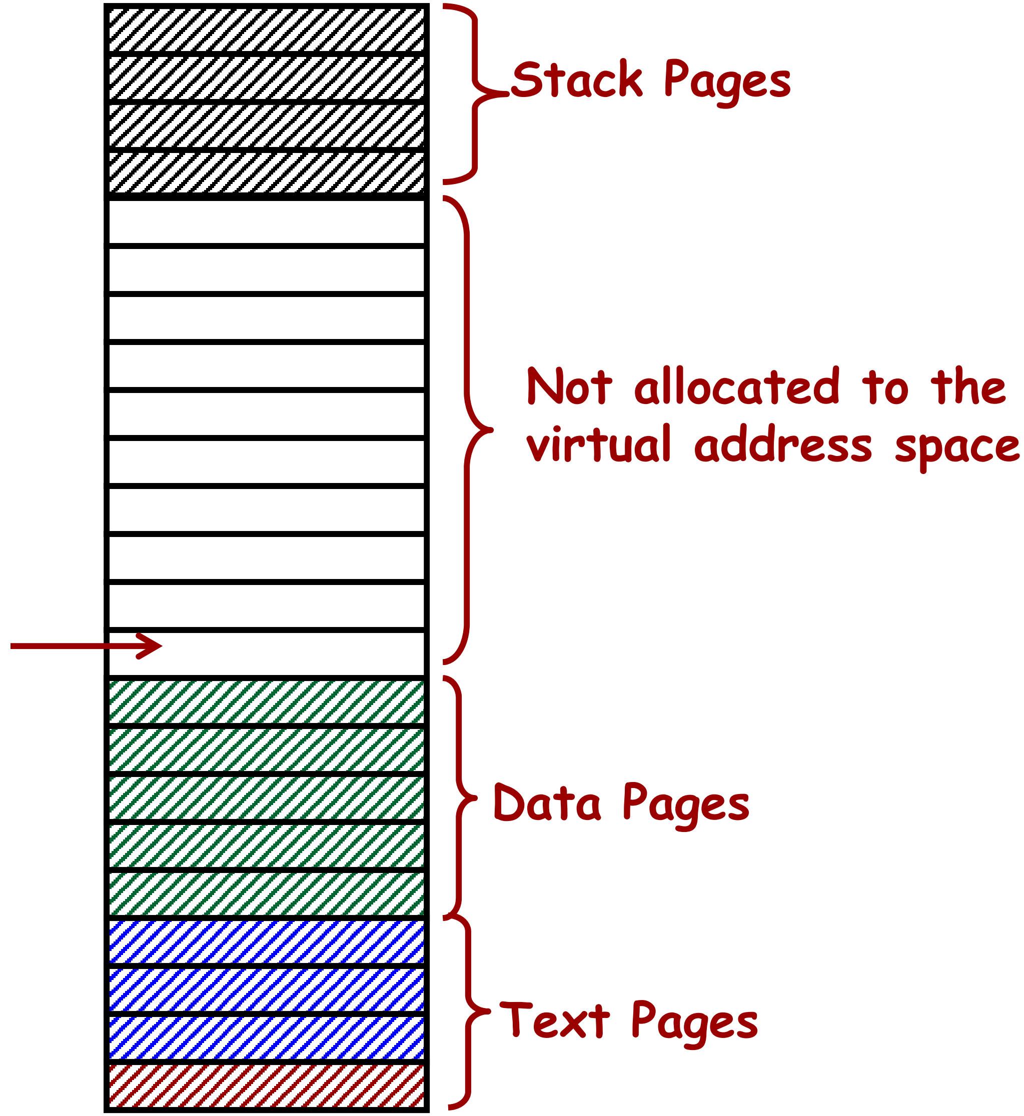
Page requested here

A new page is allocated
and process continues



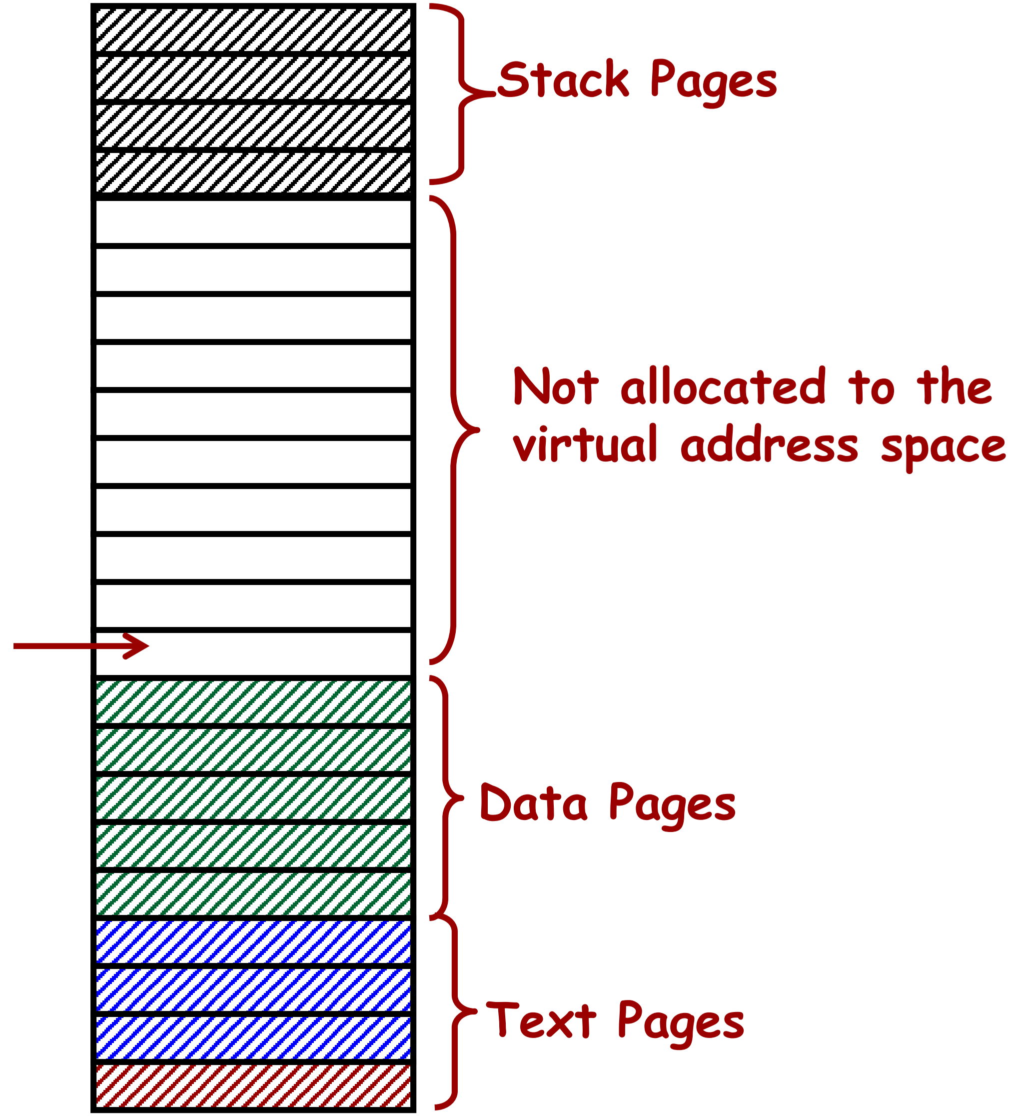
Unix processes

The heap grows;
Page requested here

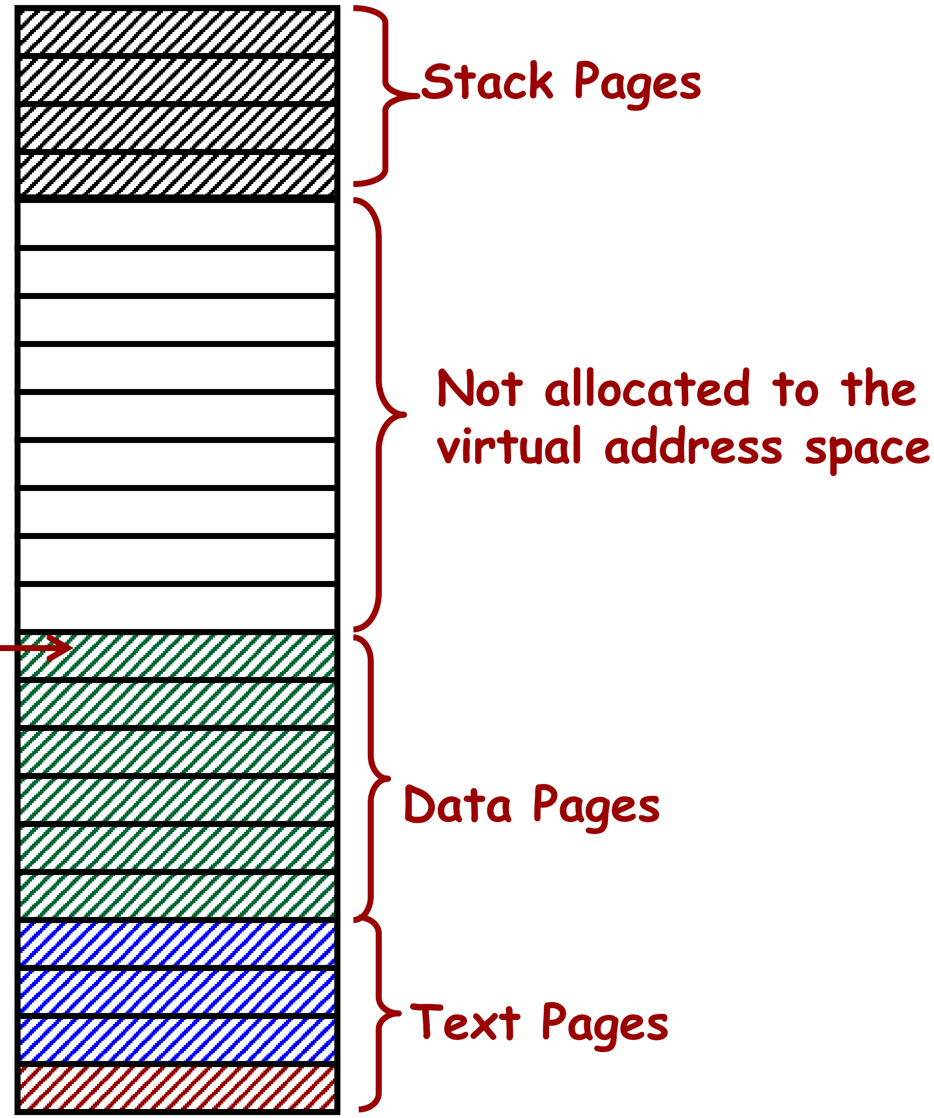


Unix processes

The heap grows;
Page requested here
A new page is allocated
and process continues



Unix processes



The heap grows;
Page requested here
A new page is allocated
and process continues

Virtual memory implementation

- **When is the kernel involved?**

Virtual memory implementation

- **When is the kernel involved?**
  - *Process creation*
  - *Process is scheduled to run*
  - *A fault occurs*
  - *Process termination*

Virtual memory implementation

- ***Process creation***
  - Determine the process size
  - Create new page table

Virtual memory implementation

- ***Process is scheduled to run***
  - MMU is initialized to point to new page table
  - TLB is flushed (unless it's a tagged TLB)

Virtual memory implementation

- ***A fault occurs***
  - Could be a TLB-miss fault, segmentation fault, protection fault, copy-on-write fault …
  - Determine the virtual address causing the problem
  - Determine whether access is allowed, if not terminate the process
  - Refill TLB (TLB-miss fault)
  - Copy page and reset protections (copy-on-write fault)
  - Swap an evicted page out & read in the desired page (page fault)

Virtual memory implementation

- ***Process termination***
  - Release / free all frames (if reference count is zero)
  - Release / free the page table

Handling a page fault

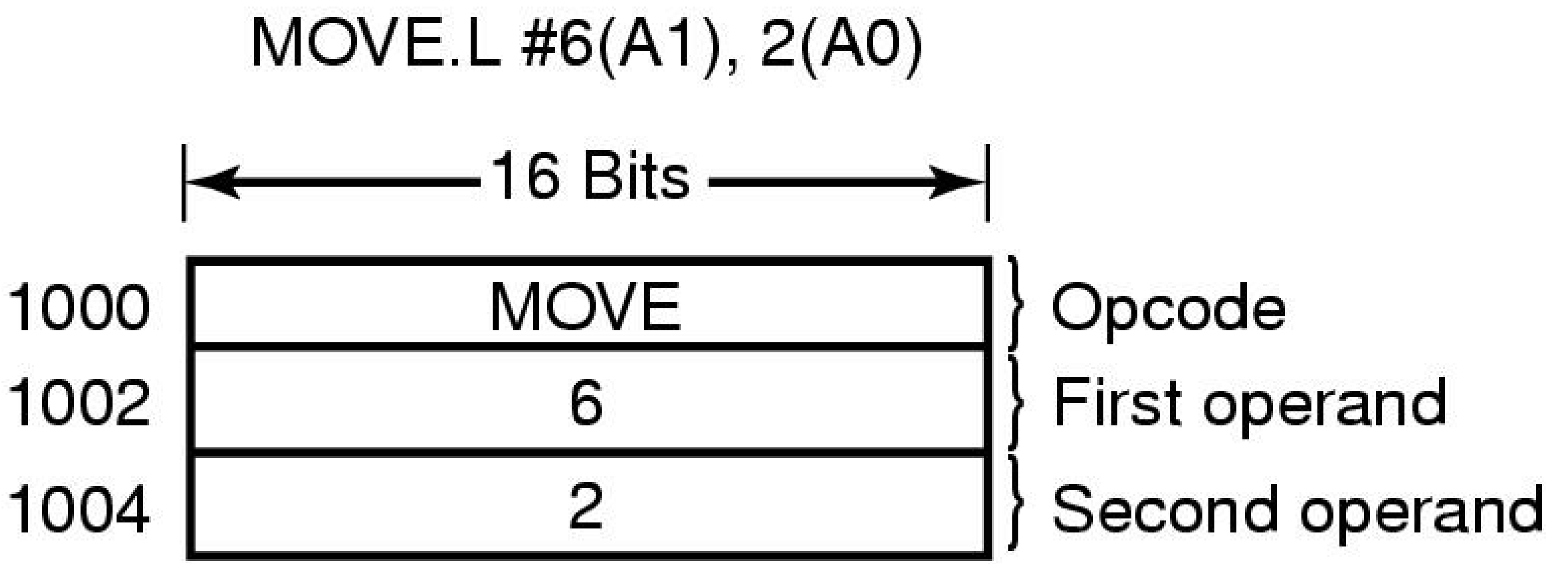
- **Hardware traps to kernel**
  - PC and SR are saved on stack
- **Save the other registers**
- **Determine the virtual address causing the problem**
- **Check validity of the address**
  - determine which page is needed
  - may need to kill the process if address is invalid
- **Find the frame to use (page replacement algorithm)**
- **Is the page in the target frame dirty?**
  - If so, write it out (& schedule other processes)
- **Read in the desired frame from swapping file**
- **Update the page tables**
- ***(continued)***

Handling a page fault

- **Back up the current instruction**
  - The "faulting instruction"
- **Schedule the faulting process to run again**
- **Return to scheduler**
- ...
- **Reload registers**
- **Resume execution**

Backing the PC up to restart an instruction

- Consider a multi-word instruction.
- The instruction makes several memory accesses.
- One of them faults.
- The value of the PC depends on when the fault occurred.
- How can you know what instruction was executing???



Solutions

- **Lot's of clever code in the kernel**
- **Hardware support (precise interrupts)**
  - Dump internal CPU state into special registers
  - Make "hidden" registers accessible to kernel
- **What if you swapped out the page containing the first operand in order to bring in the second?**

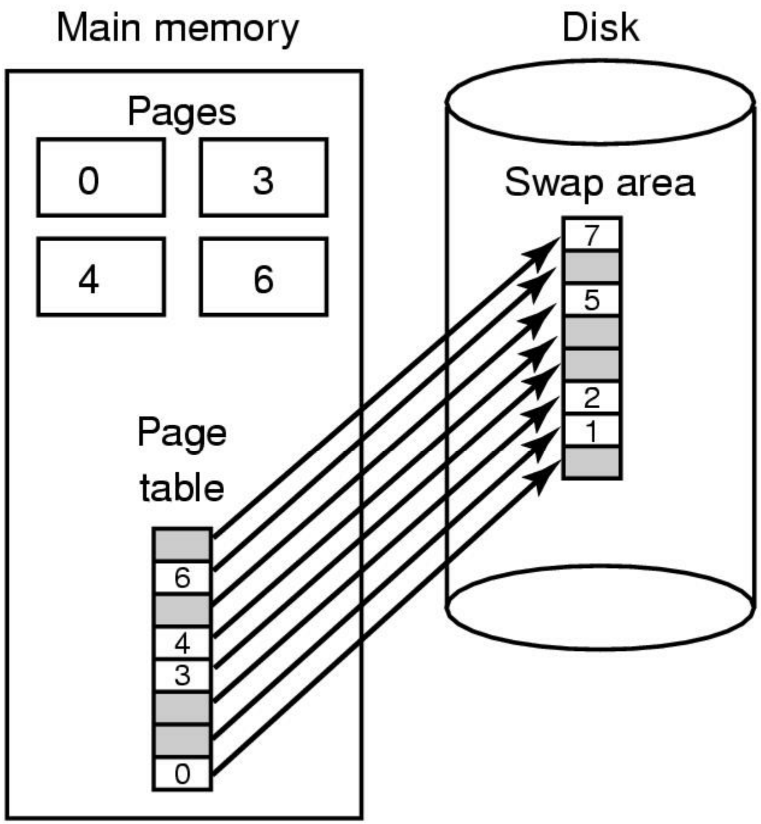
Locking pages in memory

- **Virtual memory and I/O interact**
  - Requires “Pinning” pages
- ***Example:***
  - One process does a read system call
    - **(This process suspends during I/O)**
  - Another process runs
    - **It has a page fault**
    - **Some page is selected for eviction**
    - **The frame selected contains the page involved in the read**
- ***Solution:***
  - Each frame has a flag: “Do not evict me”.
  - Must always remember to un-pin the page!

Managing the swap area on disk

- ***Approach #1:***
  - A process starts up
    - **Assume it has N pages in its virtual address space**
  - A region of the swap area is set aside for the pages
  - There are N pages in the swap region
  - The pages are kept in order
  - For each process, we need to know:
    - **Disk address of page 0**
    - **Number of pages in address space**
  - Each page is either...
    - **In a memory frame**
    - **Stored on disk**

Approach #1



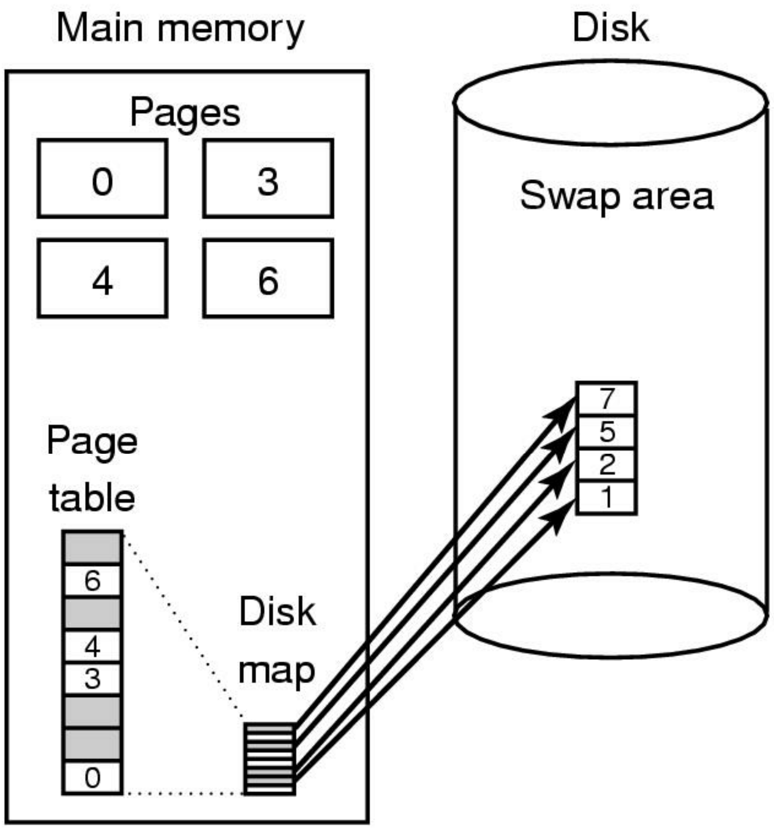
Problem

- ***What if the virtual address space grows during execution? i.e. more pages are allocated.***
- ***Approach #2***
  - Store the pages in the swap in a random order.
  - View the swap file as a collection of free "swap frames".
  - Need to evict a frame from memory?
    - **Find a free "swap frame".**
    - **Write the page to this place on the disk.**
    - **Make a note of where the page is.**
    - **Use the page table entry.**
      - **Just make sure the valid bit is still zero!**
  - Next time the page is swapped out, it may be written somewhere else.

Approach #2

This picture uses a separate data structure to tell where pages are stored on disk rather than using the page table

Some information, such as protection status, could be stored at segment granularity



Approach #3

- **Swap to a file**
  - Each process has its own swap file
  - File system manages disk layout of files

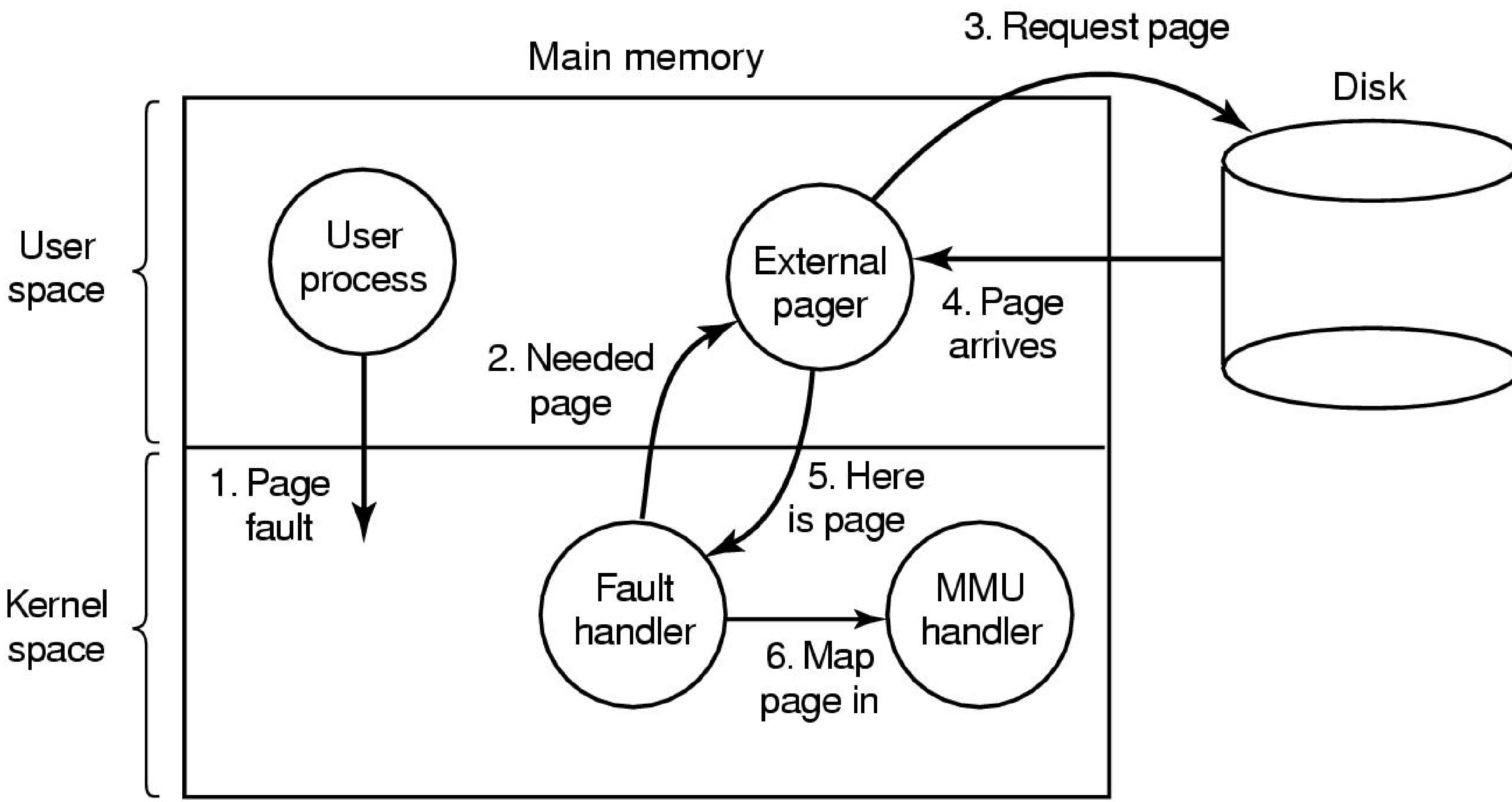
Approach #4

- Swap to an external pager process (object)
- A user-level "External Pager" process can determine policy
  - Which page to evict
  - When to perform disk I/O
  - How to manage the swap file
- When the OS needs to read in or write out a page it sends a message to the external pager
  - Which may even reside on a different machine
- Examples: Mach, Minix

Separation of Policy and Mechanism

- **Kernel contains**
  - Code to interact with the MMU
    - **This code tends to be *machine dependent***
  - Code to handle page faults
    - **This code tends to be *machine independent***

Separation of Policy and Mechanism



Paging performance

- **Paging works best if there are plenty of free frames.**
- **If all pages are full of dirty pages...**
  - Must perform 2 disk operations for each page fault
- **It's a good idea to periodically write out dirty pages in order to speed up page fault handling delay**

Paging daemon

- **Page Daemon**
  - A kernel process
  - Wakes up periodically
  - Counts the number of free page frames
  - If too few, run the page replacement algorithm...
    - **Select a page & write it to disk**
    - **Mark the page as clean**
  - If this page is needed later... then it is still there.
  - If an empty frame is needed later... this page is evicted.