Implementing Processes

Dr. Daniel Andresen

CIS520 – Operating Systems

Review: Threads vs. Processes

1. The *process* is a *kernel abstraction* for an independent executing program.

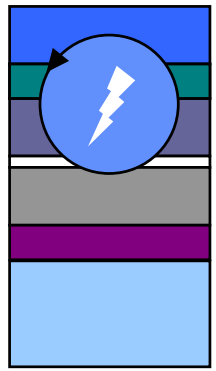
   includes at least one “thread of control”

   also includes a private address space (VAS)

   - VAS requires OS kernel support

   often the unit of resource ownership in kernel

   - e.g., memory, open files, CPU usage



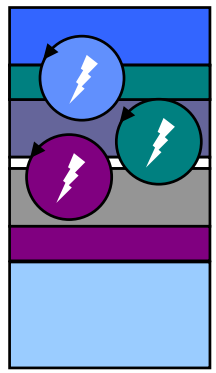
2. Threads may share an address space.

   Threads have “context” just like vanilla processes.

   - *thread context switch* vs. *process context switch*

   Every thread must exist within some process VAS.

   Processes may be “multithreaded” with thread primitives supported by a library or the kernel.



Questions

A process is an execution of a program within a private virtual address space (VAS).

1. What are the system calls to operate on processes?
2. How does the kernel maintain the state of a process?

   Processes are the “basic unit of resource grouping”.

3. How is the process virtual address space laid out?

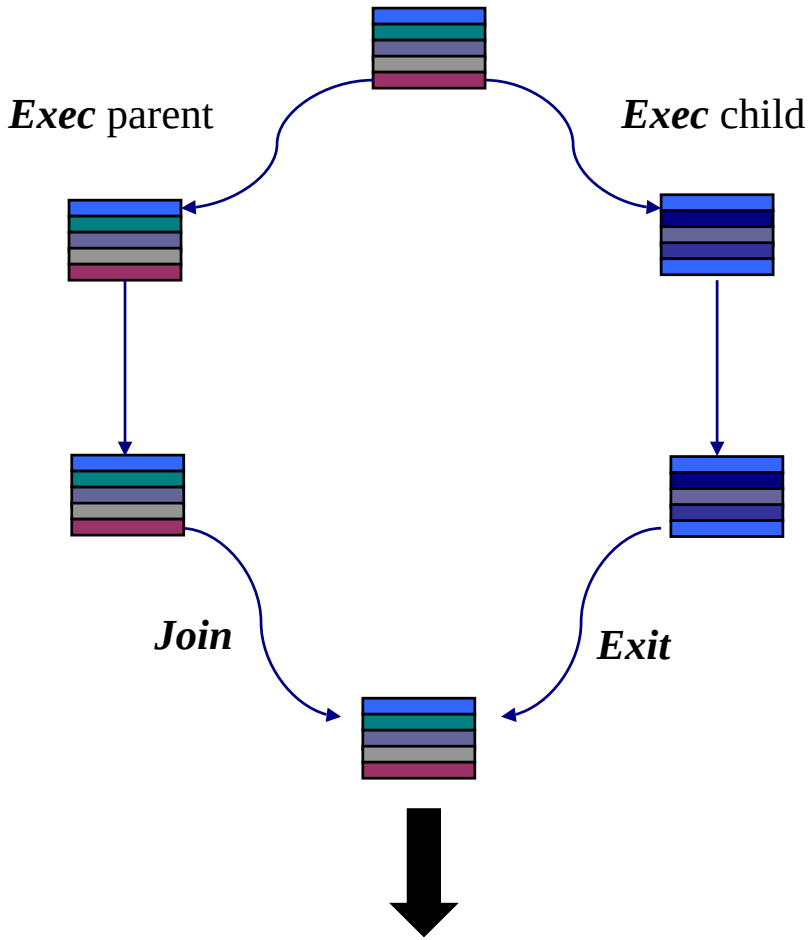
   What is the relationship between the program and the process?

4. How does the kernel create a new process?

   How to allocate physical memory for processes?

   How to create/initialize the virtual address space?

Nachos Exec/Exit/Join Example



SpaceID pid = Exec("myprogram", 0);

*Create a new process running the program "myprogram". Note: in Unix this is two separate system calls: **fork** to create the process and **exec** to execute the program.*

int status = Join(pid);

Called by the parent to wait for a child to exit, and "reap" its exit status. Note: child may have exited before parent calls Join!

Exit(status);

Exit with status, destroying process. Note: this is not the only way for a proess to exit!.

Mode Changes for Exec/Exit

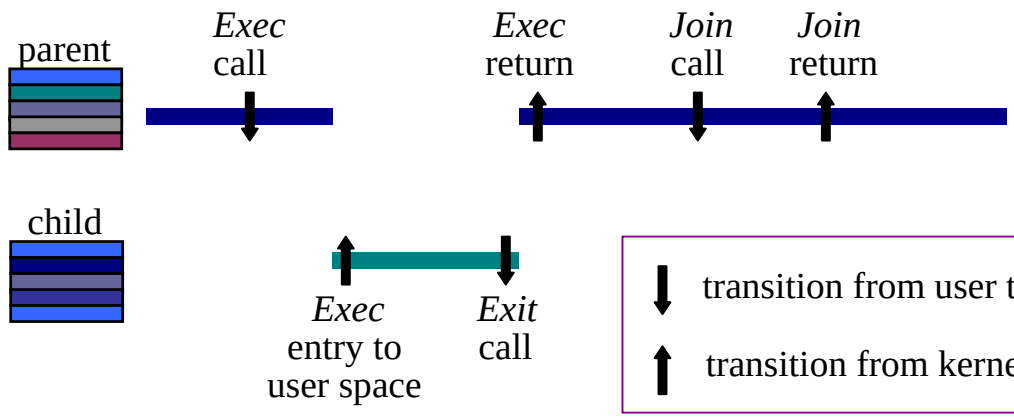
Syscall traps and “returns” are not always paired.

Exec “returns” (to child) from a trap that “never happened”

Exit system call trap never returns

system may switch processes between trap and return

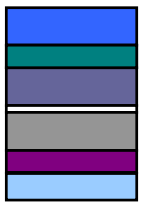
In contrast, interrupts and returns are strictly paired.



***Exec** enters the child by doctoring up a saved user context to “return” through.*

Process Internals

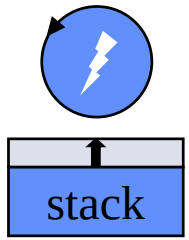
virtual address space



The address space is represented by *page table*, a set of translations to physical memory allocated from a kernel *memory manager*.

The kernel must initialize the process memory with the program image to run.

thread

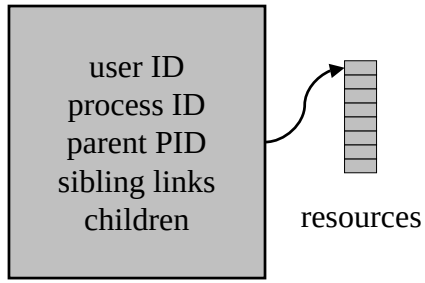


Each process has a thread bound to the VAS.

The thread has a saved user context as well as a system context.

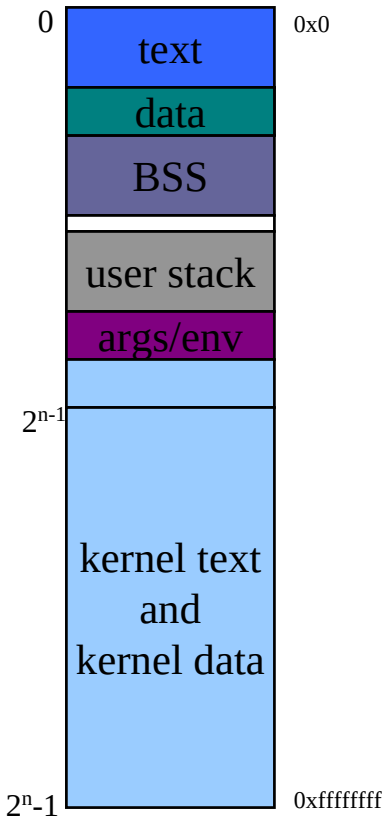
The kernel can manipulate the user context to start the thread in user mode wherever it wants.

process descriptor



Process state includes a file descriptor table, links to maintain the process tree, and a place to store the exit status.

Review: The Virtual Address Space

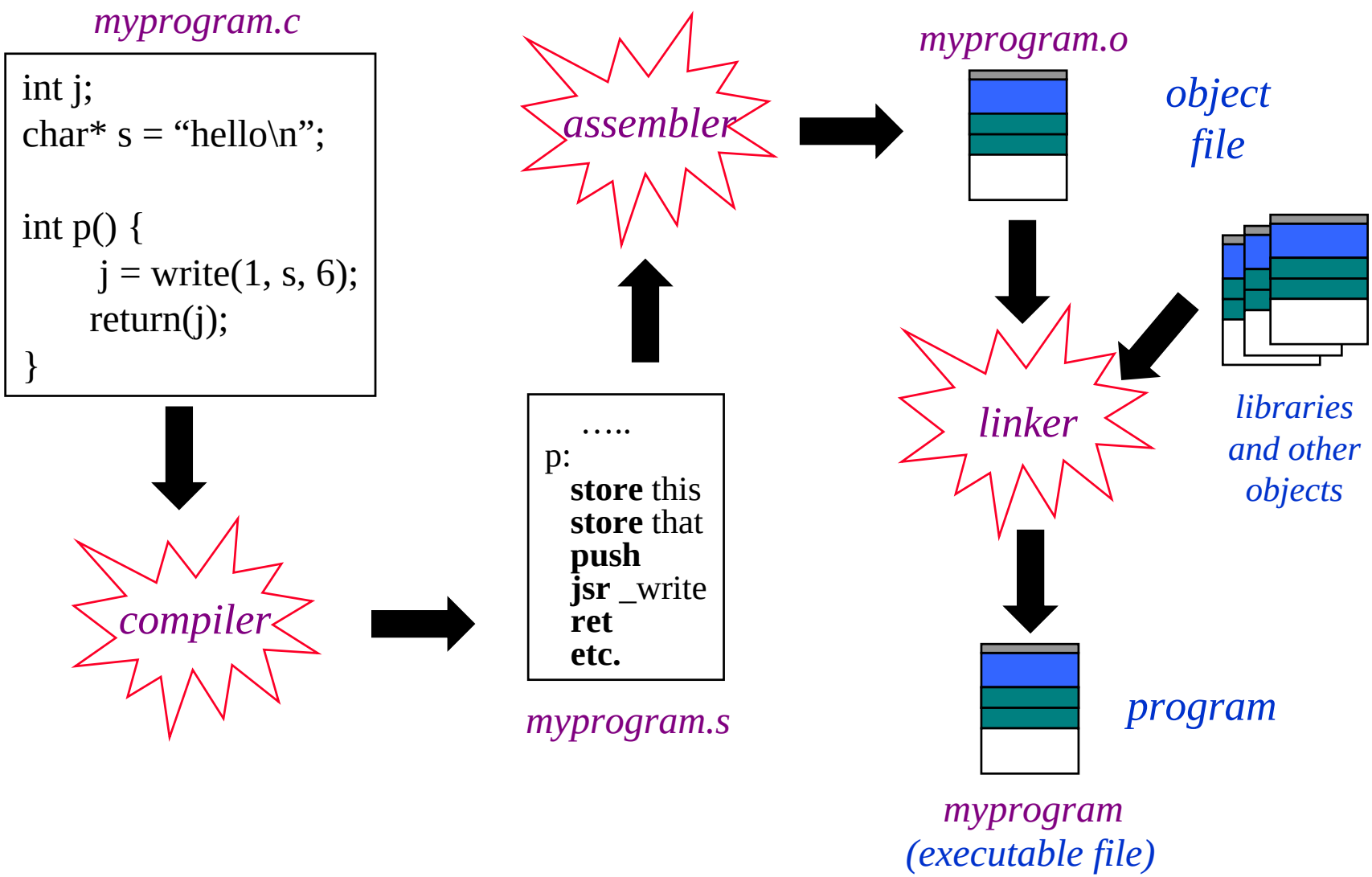


A *typical* process VAS space includes:

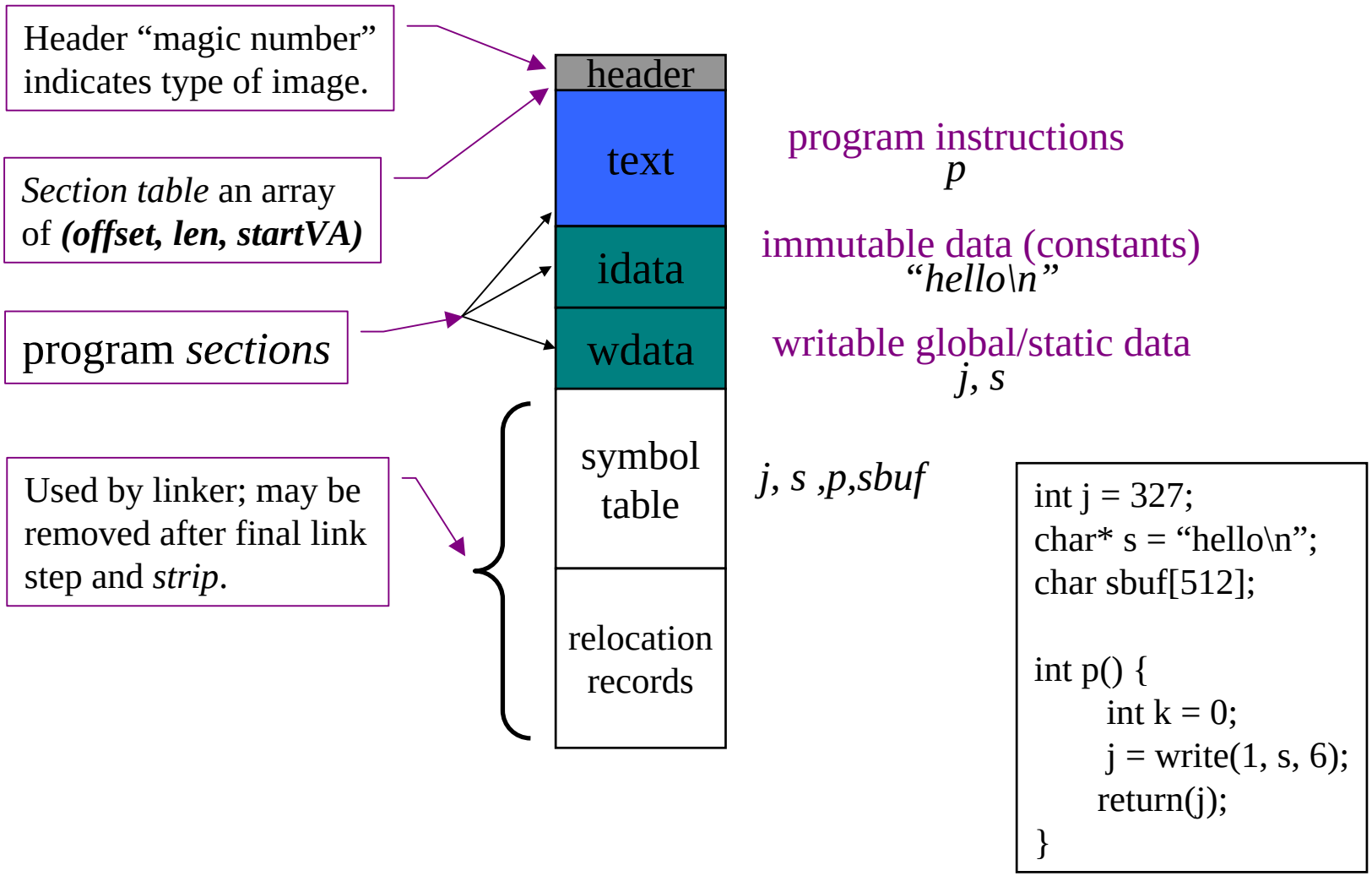
- user regions in the lower half
  - V->P mappings specific to each process
  - accessible to user or kernel code
- kernel regions in upper half
  - shared by all processes
  - accessible only to kernel code
- **Nachos:** process virtual address space includes only user portions.
  - mappings change on each process switch

A VAS for a private address space system (e.g., Unix) executing on a typical 32-bit architecture.

The Birth of a Program

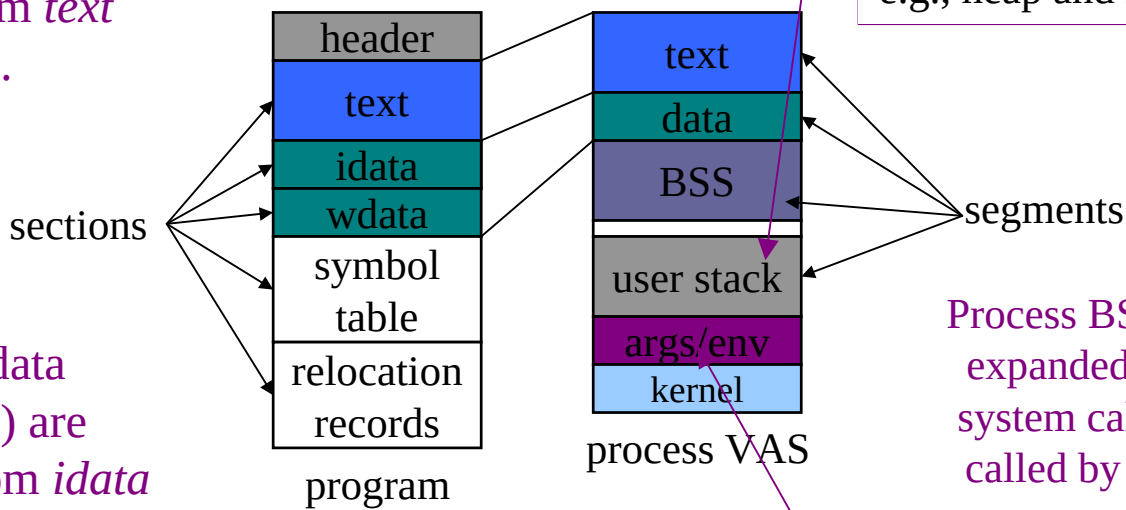


What's in an Object File or Executable?



The Program and the Process VAS

Process *text* segment is initialized directly from program *text* section.



BSS
"Block Started by Symbol"
(uninitialized global data)
e.g., heap and *sbuf* go here.

Process data segment(s) are initialized from *idata* and *wdata* sections.

Process BSS segment may be expanded at runtime with a system call (e.g., Unix sbrk) called by the heap manager routines.

Text and *idata* segments may be write-protected.

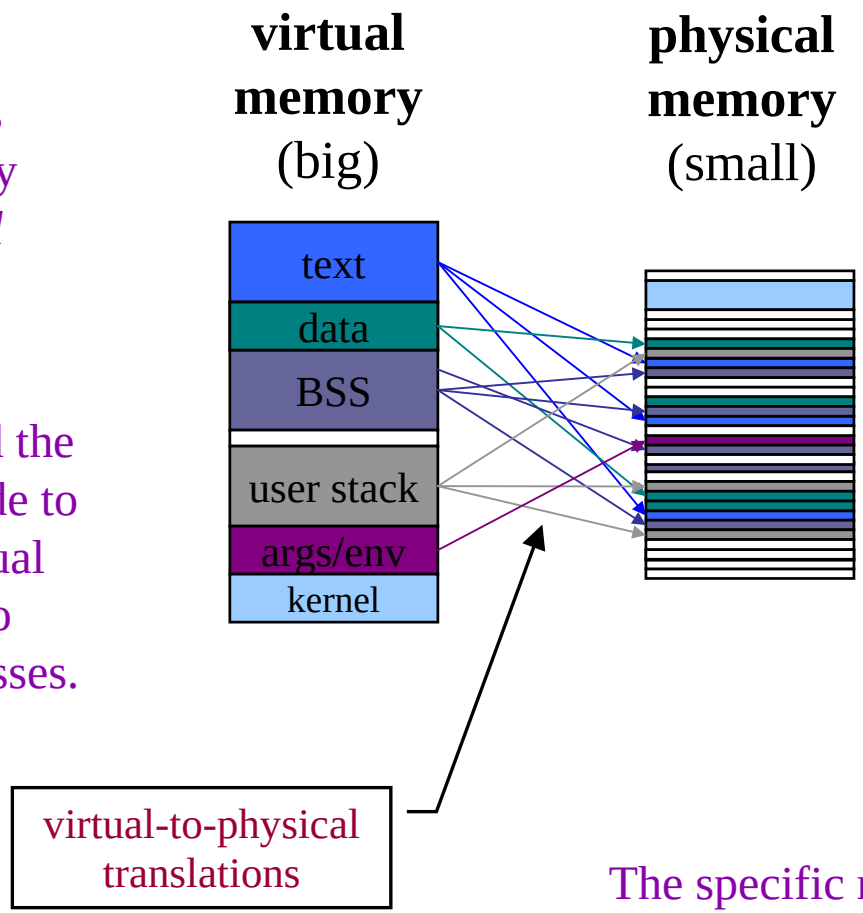
Process stack and BSS (e.g., heap) segment(s) are zero-filled.

Args/env strings copied in by kernel when the process is created.

Review: Virtual Addressing

User processes address memory through *virtual addresses*.

The kernel and the machine collude to translate virtual addresses to physical addresses.



The kernel controls the virtual-physical translations in effect for each space.

The machine does not allow a user process to access memory unless the kernel "says it's OK".

The specific mechanisms for memory management and address translation are *machine-dependent*.

Memory Management 101

Once upon a time...memory was called “core”, and programs (“jobs”) were loaded and executed one by one.

- load image in contiguous physical memory

  start execution at a known physical location

  allocate space in high memory for stack and data

- address text and data using physical addresses

  prelink executables for known start address

- run to completion

Memory and Multiprogramming

One day, IBM decided to load multiple jobs in memory at once.

- improve utilization of that expensive CPU
- improve system throughput

Problem 1: how do programs address their memory space?

load-time relocation?

Problem 2: how does the OS protect memory from rogue programs?

???

Base and Bound Registers

Goal: isolate jobs from one another, and from their placement in the machine memory.

- addresses are offsets from the job's *base address*

  stored in a machine *base register*

  machine computes *effective address* on each reference

  initialized by OS when job is loaded

- machine checks each offset against job size

  placed by OS in a *bound register*

Base and Bound: Pros and Cons

Pro:

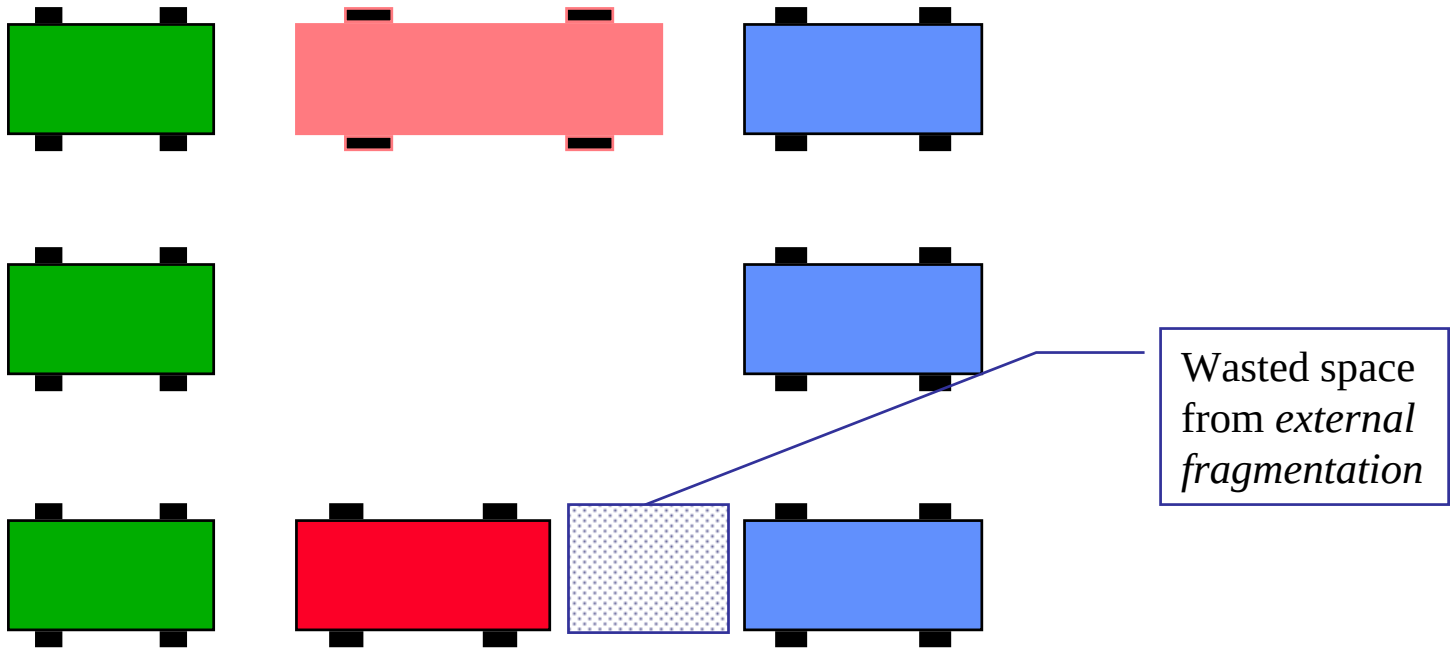
- each job is physically contiguous
- simple hardware and software
- no need for load-time relocation of linked addresses
- OS may swap or move jobs as it sees fit

Con:

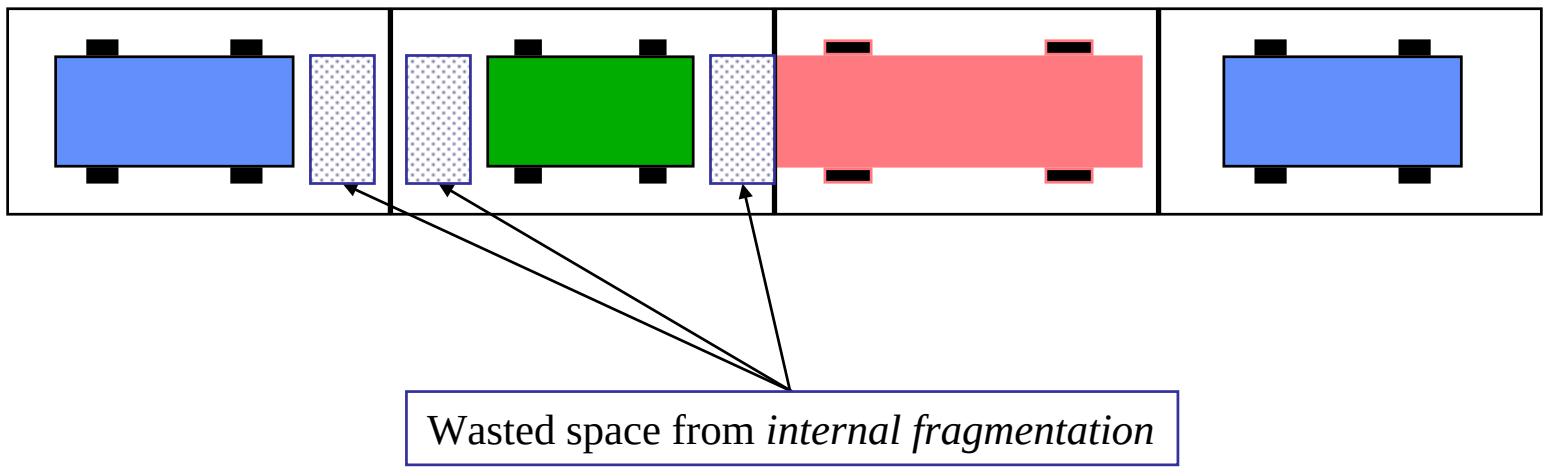
- memory allocation is a royal pain
- job size is limited by available memory

Variable Partitioning

Variable partitioning is the strategy of parking differently sized cars along a street with no marked parking space dividers.



Fixed Partitioning



The Storage Allocation Problem

- fixed partitioning leads to *internal fragmentation*
- variable partitioning leads to *external fragmentation*

  which partition to choose? *first fit*, *best fit*, *worst fit*, *next fit*?

  these strategies don't help much

- external fragmentation can be fixed by:

  compaction (e.g., *copying garbage collection*)

  coalescing (e.g., *buddy system*)

- these issues arise in *heap managers*

  e.g., runtime support for C++ *new* and *delete*

Managing Storage with Pages or Blocks

Idea: allow *noncontiguous* allocation in fixed blocks.

- partition each (file, memory) into *blocks* of *2**N* bytes
- partition storage into *slots* of size *2**N* bytes

  blocks are often called *logical blocks* or *pages*

  slots are often called *physical blocks* or *frames*

Paged allocation simplifies storage management:

- allocate a slot for each block independently
- slots are reusable and interchangeable

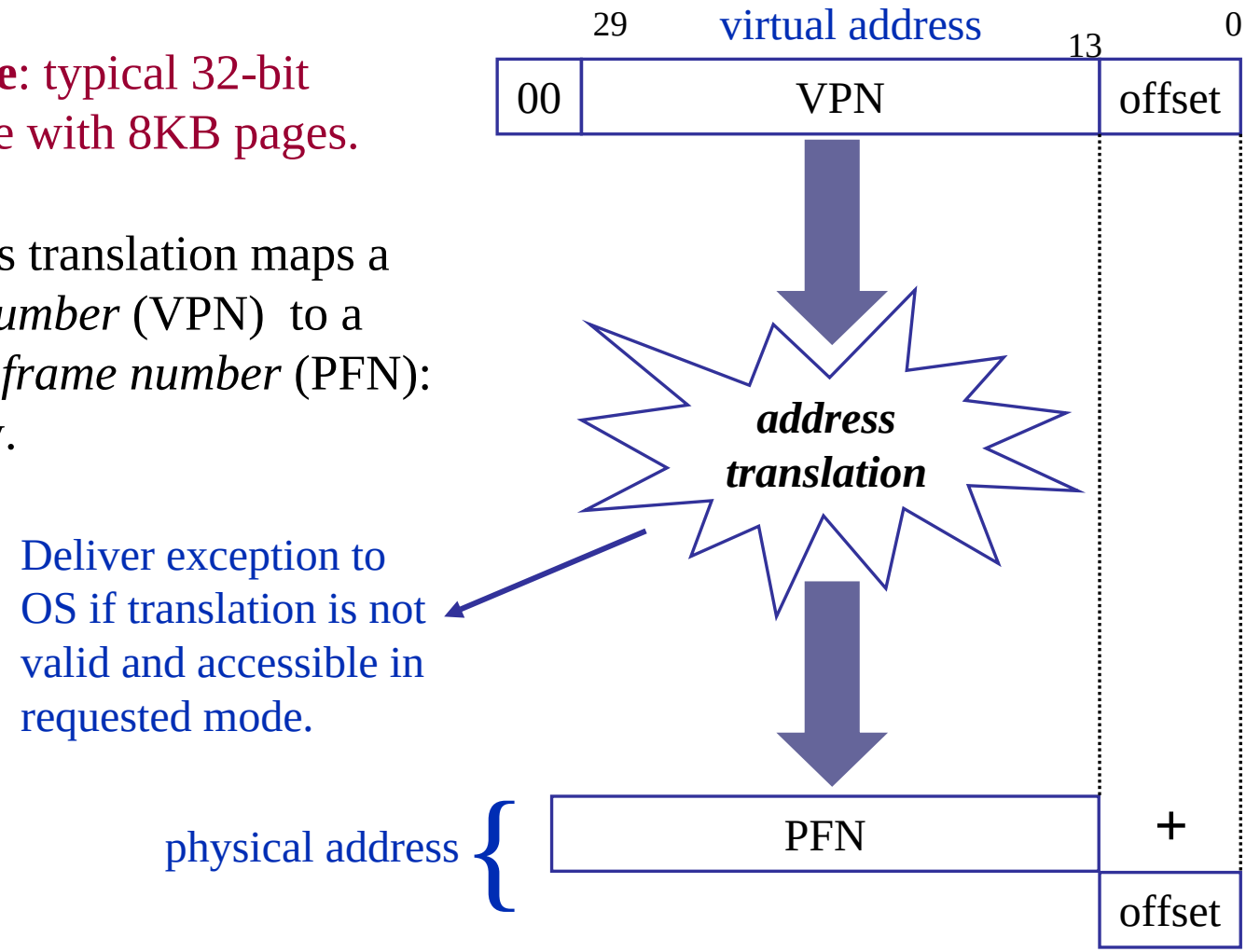
  no need to search for a "good" slot; any free one will do

- no external fragmentation; low internal fragmentation

Virtual Address Translation

Example: typical 32-bit architecture with 8KB pages.

Virtual address translation maps a *virtual page number* (VPN) to a physical *page frame number* (PFN): the rest is easy.



Translating the Logical Address Space

Problem: the system must locate the slot for each block on-the-fly as programs reference their data.

Applications name data through a *logical (virtual) address space* that isolates them from the details of how storage is allocated.

Translate addresses indirectly through a *logical-physical map*.

The map *M* is a function that maps a *logical block* number in the address space to a *physical slot* number in storage.

slot_index = Map(logical_address >> N)

Block offset (low-order *N* bits of the address) is unchanged.

*offset = logical_address & ((2**N) - 1)*

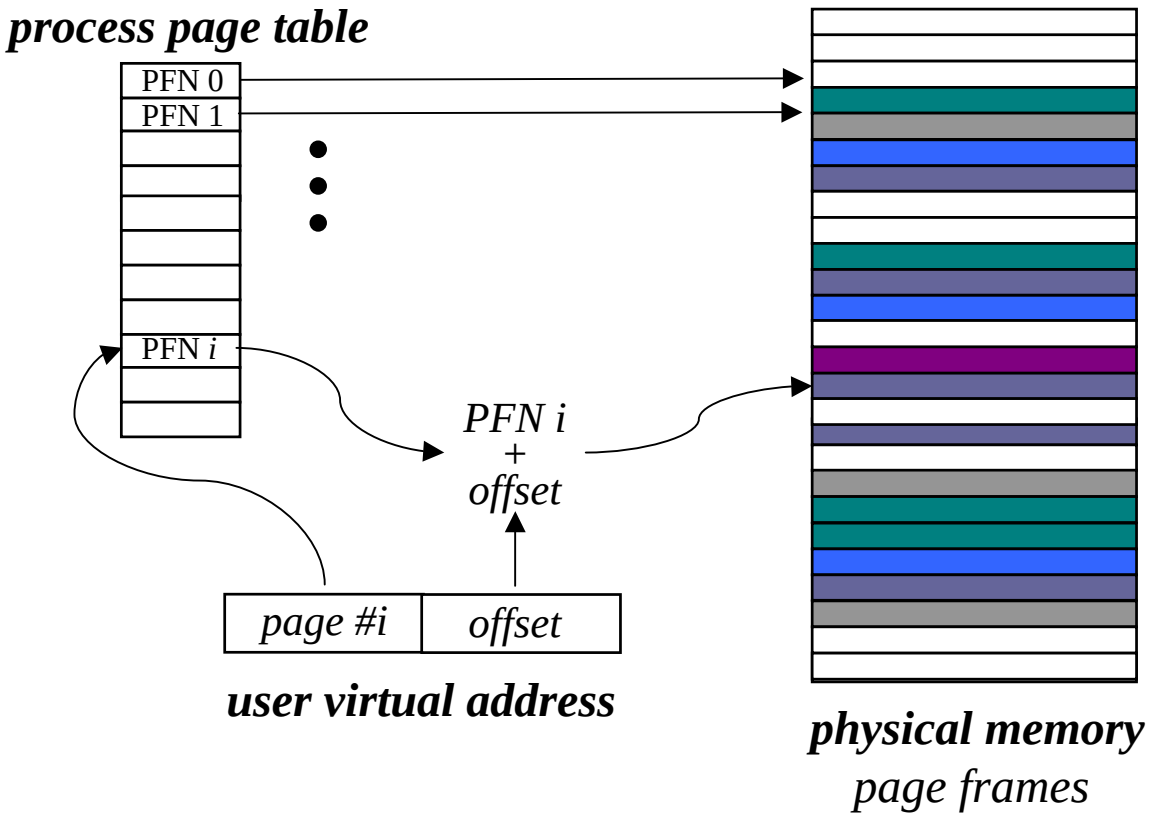
physical_address = (slot_index << N) + offset

Examples of Logical-to-Physical Maps

1. ***files***: block map ("inode" in Unix)
   - logical-physical map is part of the file metadata

     map grows dynamically; file's byte length is stored in the inode
   - the block map is stored on disk and cached in memory
   - block size is a power-of-two multiple of the disk sector size
2. ***virtual memory***: page tables
   - *virtual address = virtual page number + offset*
   - page table is a collection of ***p****age* ***t****able* ***e****ntries* (*ptes*)
   - each valid pte maps a virtual page to a *page frame number*

A Simple Page Table



Each process/VAS has its own page table. Virtual addresses are translated relative to the current page table.

In this example, each VPN *j* maps to PFN *j*, but in practice any physical frame may be used for any virtual page.

The page tables are themselves stored in memory; a protected register holds a pointer to the current page table.

Nachos: A Peek Under the Hood