Memory Management I

Tom Kelliher, CS 311

Apr. 11, 2012

1 Administrivia

Announcements

Assignment

From Last Time

Introduction to Linux kernel modules.

Outline

1. Address binding, loading, libraries, and overlays.

2. Logical and physical address spaces.

3. Swapping.

4. Contiguous and non-contiguous allocation.

Coming Up

Memory management II.

1. Address binding, loading, libraries, and overlays.
2. Logical and physical address spaces.
3. Swapping.
4. Contiguous and non-contiguous allocation.

2 Preliminaries

Assumption: several processes competing for memory. Questions to keep in mind:

1. How is memory allocated among processes?
2. Sharing?
3. Contiguous/non-contiguous allocation?
4. Fixed- or variable-sized allocation chunks?
5. Over-commitment. What happens?

2.1 Address Binding

1. Sometimes, a program can't be loaded "just anywhere" in memory.
2. When does a symbolic name become associated with a memory location?
3. How is that location addressed? (Direct, indexed, indirect, etc.)

Address binding options and consequences

1. Compile time. Absolute (direct) addressing used. Program *must* be loaded into memory at a fixed location. MS-DOS `.COM` programs.

2. Load time. Compiler/linker insert "dummy" references in code. Loader knows load address at run time, finds and fixes-up dummy references.

   (a) Addressing modes?

   (b) Re-location? Must re-load to re-locate.

3. Run time. Pc-relative or base addressing. Base register points to load address.

2.2 Dynamic Loading

1. On-demand module loading — "lazy" loading.

2. Load only what's necessary.

2.3 Dynamic Linking

1. Static linking — link at compile-time.

   (a) Larger binaries — wasted disk space.

   (b) Must re-link source when updating libraries.

2. Dynamic linking (shared libraries) — link at run-time.

   (a) Smaller binaries.

   (b) Stubs: locating the library in memory or on disk.

   (c) Automatic library updates to fix library faults in *all* binaries. (With static must re-link all binaries.)

   (d) Shared libraries. Major (new version #) and minor (same version #) versions.

   (e) DLLs in Wintel world. Broken programs.

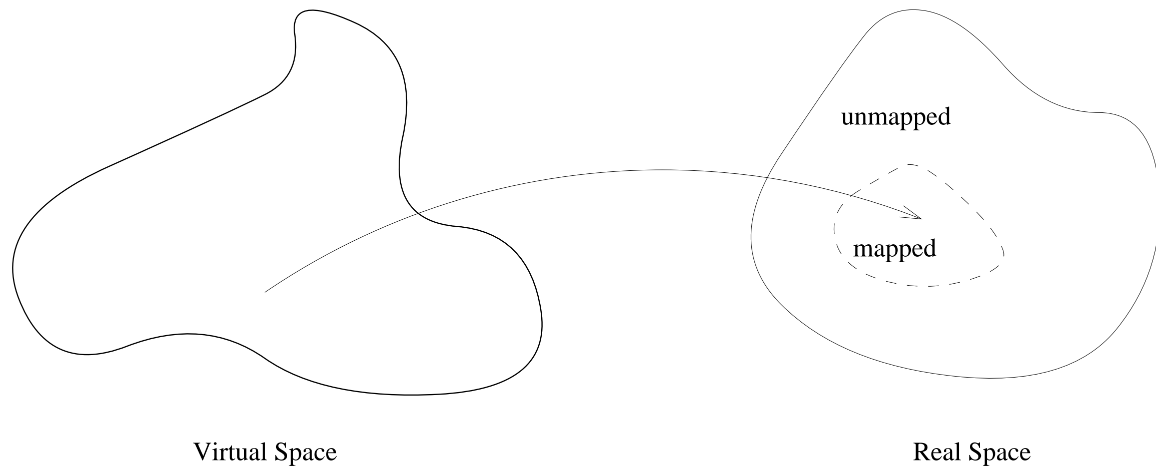
2.4 Overlays

Umm, my program is larger than physical memory. What do I do? Example: two-pass assembler.

3 Logical and Physical Addresses

Logical = Virtual

Recall:

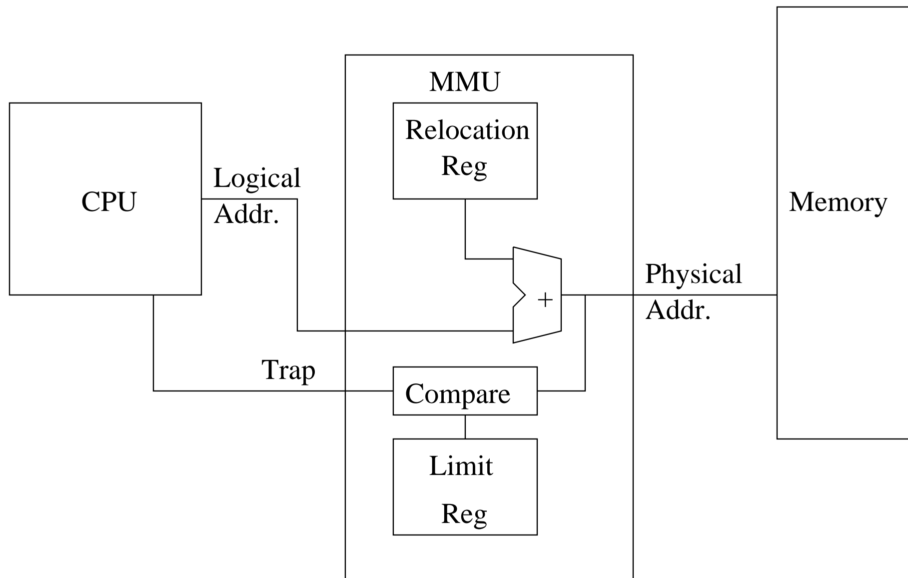


1. Program (CPU) generates logical addresses.

2. MMU converts them to physical addresses.

3. Compile-time, load-time binding: physical address = logical address.

Simple MMU scheme:



Note: Program can utilize compile-time binding.

4 Swapping

1. Process *must* be *entirely* in memory to execute.
2. If in ready Q, can swap-out to "backing store."
3. If in I/O wait must "lock" in memory. Why?
4. Must process be swapped-in to same memory space? Depends on loader design.
5. What needs to be swapped out, exactly? The entire partition? The entire process space?
6. Swapping in Unix occurs:
   (a) The system page map has become too fragmented to allocate page tables for some process.

   (b) There are at least two runnable processes, the average amount of free memory has been less than that desired (*desfree*) over the last 5 and 30 seconds, and either the paging rate is excessive or the short-term average free memory is less than *minfree*.

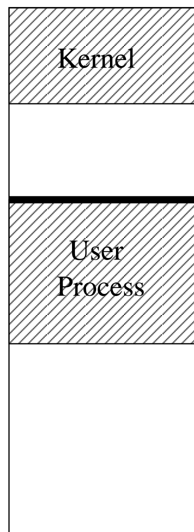
(c) A swapped-out process is ready to run and a process is found in memory that has been sleeping for at least 20 seconds; or the swapped-out process has been out for at least 10 seconds and the process chosen for swapping out has been in memory at least 20 seconds.

5 Contiguous Allocation

Key: *contiguous*.

What are we leading up to, here?

5.1 Single User Partition



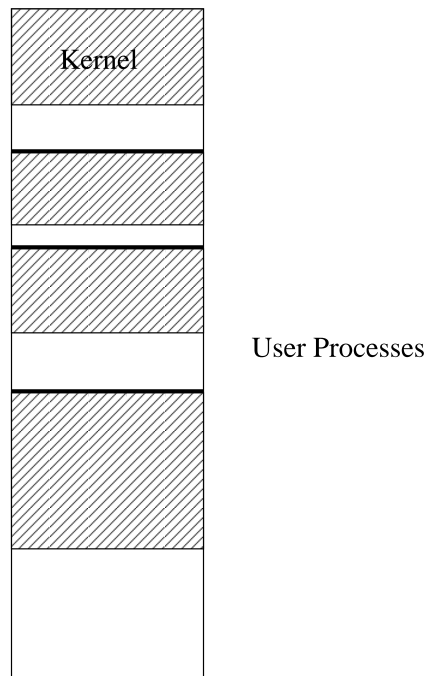
1. Only one process in memory at a time.

2. Must swap to do a context switch. Speed?

3. Unused Memory: *internal fragmentation*.

5.2 Multiple User Partitions

5.2.1 Fixed Partitions



1. Partitions usually of differing sizes.

2. Job placement policy.

3. Can swap jobs in and out of partitions.

4. Internal fragmentation.

5.2.2 Variable Partitions

1. Number of partitions varies.

2. Partitions allocated according to process requirements.

3. What if a process grows?

4. Free list of unallocated *holes*.

5. Placement policies:

    (a) First fit. Best time.

    (b) Best fit.

    (c) Worst fit — worst time, storage utilization.

6. *External fragmentation*.

7. Combining adjacent holes.

8. Memory compaction — must be using relocatable code. Minimizing copying.