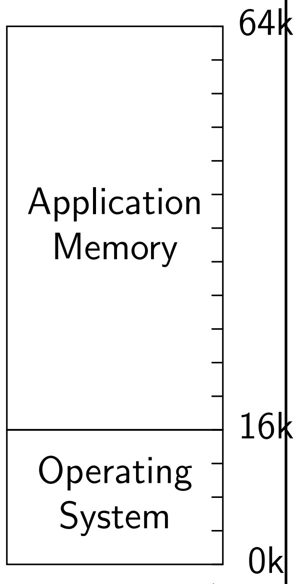
Memory Management

Once Upon a Time

- Main memory is a resource
- Operating systems have to manage it
- Once, it was simple: put the OS at the bottom, and the application gets the rest
- Well, it wasn't really that simple. . .



2 / 41

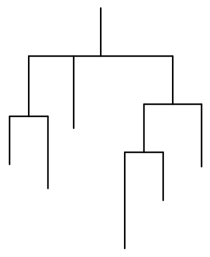
Memory as a Resource

- Memory has always been scarce
- More precisely, memory has generally been expensive
- We've never had enough to satisfy all applications
- We've always had to cope

3 / 41

Static Overlays

- Back in the mists of time, application programmers had to manage memory allocation themselves
- Different subroutines occupied the same area of memory at different times
- The OS provided library routines to support this, but the layout — and the safety of the scheme — was up to the programmer



4 / 41

Memory and Multiprogramming

- Memory sizes increased
- We used the increase to support multiprogramming
- Applications *still* had to manage their own memory
- But the OS had new problems

5 / 41

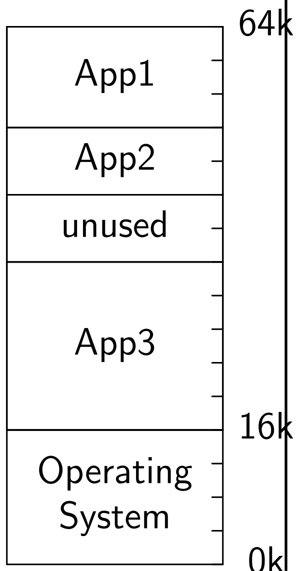
Challenges

- Relocation
- Memory protection
- Allocation

6 / 41

Relocation

- Originally, all applications started at the same address, perhaps 16K
- With multiprogramming, applications start at different spots
- In fact, a given application could start at a different location each time it ran



7 / 41

Relocation

- Position-independent code
- Load-time relocation
- Hardware assist

Position-Independent Code (PIC)

- Some code is naturally *position-independent*
- There are no references to absolute memory locations; everything is relative to a register or the program counter:

```
L       R3,16(R12)
B       *-4
```

- (Easier on some hardware than others)
- Used for shared libraries on most modern Unixes
- Generally a compiler option today: `-fpic` or `-fPIC` when using `gcc`

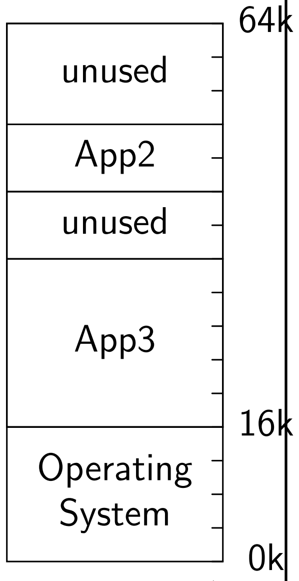
Load-Time Relocaton

- At program load time, the execution address is known
- Add the actual offset to the presumed location specified by the compiler and linker
- The cost of this — that is, the number of words that need to be relocated — varies depending on the hardware architecture and compiler

10 / 41

Hardware-Assisted Relocation

- PIC is sometimes awkward (and doesn't apply after loading)
- Software relocation can be expensive
- We sometimes want to move programs around
- We have 20K free here, but the largest program we can run is 12K



11 / 41

Base/Limit Registers

- The base and limit registers are set by the OS
- The contents of the base register is added to all memory references
- Any memory reference larger than the limit register is invalid and causes a trap

12 / 41

Problems with Base/Limit Registers

- Copying programs is expensive
- An extra addition on every memory reference is expensive, too
- We need more flexibility

13 / 41

Swapping

- Write the entire program out to disk when it's blocked
- Read it all back in again, possibly at a different address, when it's ready to run
- Can use this to *compact* memory
- Apparently frees up the CPU, but disk I/O consumes memory bandwidth

Memory Protection

- Need some way to keep applications from touching other applications' memory (or the operating system's)
- Base/limit registers handle that automatically
- Another solution: attach protection information to each memory block
- Example: on IBM System/360, each 4K of memory had a 4-bit storage key; it had to match the key in the program status word

Memory Allocation

The Memory Allocation Problem

- The OS knows that certain areas of memory are free
- An application requests some
- What block of memory is given to that application?
- For now, assume a linked list of free blocks

First Fit

- Traverse the list until a large-enough block is found
- Allocate as much of it as is needed
- Return the rest to the free list
- When freeing memory, must coalesce adjacent blocks
- Obvious answer: order linked list by address

Best Fit

- First fit seems wasteful — there may be an exact match further down the list
- Solution: *best fit*
- Find the free block that's closest in size to the request
- Obvious data structure: order linked list by size
- Makes coalescence less efficient
- And best fit isn't more efficient!

19 / 41

Worst Fit

- Best fit tends to leave small, useless blocks (fragmentation)
- What about *worst fit* — pick the largest block and sub-divide it?
- That's not very good, it turns out; first fit and best fit are better and are pretty close
- Note: this is a simulation result based on trace data
- For first fit, as much as $1/3$ of memory may be unusable!
- Many other algorithms; see Knuth, Vol. 1, for details

20 / 41

The Status of Memory Allocation

- Classic memory allocation isn't very important for the OS, because of virtual memory
- But applications still need memory allocation within their OS allocation: `malloc()` in `C`; `new` in Java and `C++`
- The same sorts of algorithms are used, with one important addition: applications can ask the OS for more memory

21 / 41

Virtual Memory

22 / 41

Problems with Earlier Schemes

- Programmer time wasted on memory allocation
- Inefficiency of block allocation algorithms
- Expense of compaction
- Expense of base registers
- The need for special memory protection mechanisms
- The solution: *virtual memory*

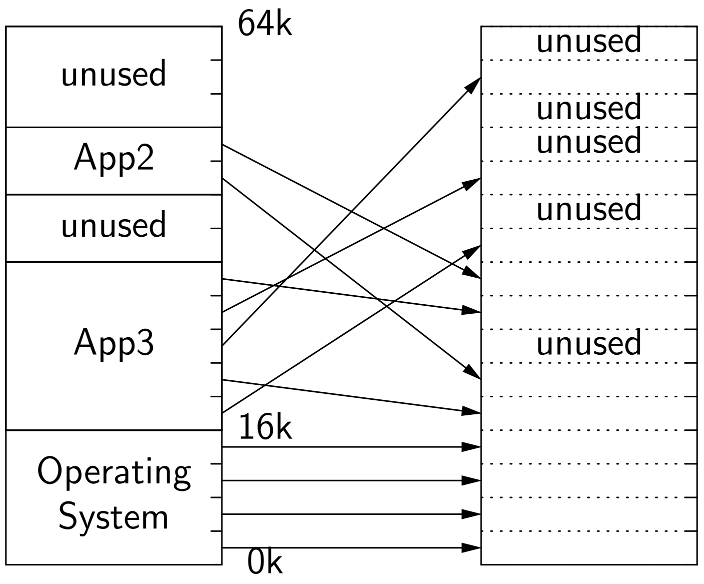
23 / 41

Virtual Memory

- Convert a *virtual address* — the address as seen by the program — to a *physical address*
- Mapping is done by *pages* — groups of (perhaps) 4K bytes
- The mapping need not refer to contiguous areas of physical memory
- Eliminates all problems of allocation inefficiency, compaction, relocation, memory protection, and more

24 / 41

Virtual Memory



25 / 41

Virtual Memory

- Each page occupies a *page frame* in physical memory
- There is no relationship between a page's virtual address and its physical address
- Each application could, in fact, start at the same virtual address as every other application
- The OS may or may not be visible in the application's address space
- Physical memory that isn't pointed to can't be accessed — no need for special memory protection (but some forms are still used)

26 / 41

Implementing Virtual Memory

- Divide a virtual address A into $\langle V, O\rangle$, where V is the *virtual page number* and O is the offset within the page
- The *memory management unit* (MMU) maps V into P, the *physical page number* and produces $\langle P, O\rangle$
- In its simplest form, this mapping is done by direct indexing into an MMU register bank M:
$$A \rightarrow \langle M[V], O\rangle$$
- This scheme can be faster than a base register; no additions are involved

27 / 41

That's Too Simple

- Once, that could work; page sizes were large relative to memory
- The PDP-11s on which Unix grew up had only 8 pages of address space — that many registers were affordable
- The CLIC machines have 1G of memory and pages of 4K bytes — or 256K possible pages
- We can't have that many MMU registers
- We need a separate page table for each process — we can't afford to copy that many MMU registers during process switches

28 / 41

Page Tables in RAM

- First-order solution: put the page table in RAM
- Still problematic:
  - It's slow — an extra RAM access for each memory reference
  - It's a lot of RAM for page tables
- Page tables do live in RAM, but there are optimizations

29 / 41

Translation Lookaside Buffer

- Mappings from $V \to P$ are cached in the *Translation Lookaside Buffer* (TLB)
- The TLB is an associative memory — it maps a *value*, rather than an index, into another value
- When there's a TLB miss, an old entry is discarded, a memory lookup is done, and the new entry is inserted into the TLB
- This works because programs tend to exhibit *locality of reference*
- On many systems, the TLB is managed in hardware; on RISC systems, it is explicitly populated by the OS

30 / 41

The Need for Segmentation Tables

- We still don't want to have 256K page table entries
- We could have a limit register — but sometimes, virtual memory is sparse
- Example: put the OS at (virtual) address 2G, the executable at (virtual) address 128M, and have the stack grown down from 192M. On Solaris, applications start (near) 0, but the stack grows down from 4G.
- We need two-level page tables

31 / 41

Segmentation Tables

- A is treated as $\langle S, V, O \rangle$
- A *segment number* S points to a page table; V, the page number, is an index into the page table array pointed to by S
- The full translation is thus

$$V \to \langle R[S[P]], O \rangle$$

  where R is the segment table address
- The TLB is used to speed this up, too

32 / 41

VM and the Cache

- Some caches work on physical memory addresses; others work on virtual addresses
- Either way, the cache is largely useless after a VM map change
- Either the cache has to be invalidated or different physical addresses are used

33 / 41

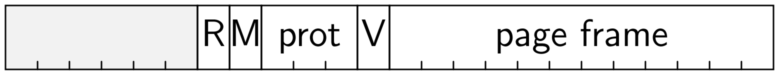
Switching Processes

- Load a single register pointing to the new segment table
- Purge the old TLB (often done explicitly)
- The first reference to memory uses the segment register as find a page table pointer in memory; it then makes a second memory reference to find the physical page number
- Again, locality of reference helps
- For kernel mode, use a full segment table; for user mode, use a segment table that doesn't include the kernel pages
- Memory map changes are *extremely* expensive because of the effect on the cache and the TLB

Segment Table Entries

- A segment table entry has control information as well as a page table address
- In particular, it has a valid/invalid bit
- Unused parts of the address space have invalid segment table entries, and hence no page table

Page Table Entries



R	page referenced
M	page modified
prot	memory protection: read, write, execute
V	entry valid
page frame	page frame in physical memory

36 / 41

Page Faults

- If an invalid segment or page table entry is referenced, a *page fault* occurs
- The kernel looks at the offending address and decides what to do
- Perhaps it's a program bug — send an appropriate signal
- Perhaps the memory is *paged out* to disk — block the process, read in the page, fix up the page table, and retry

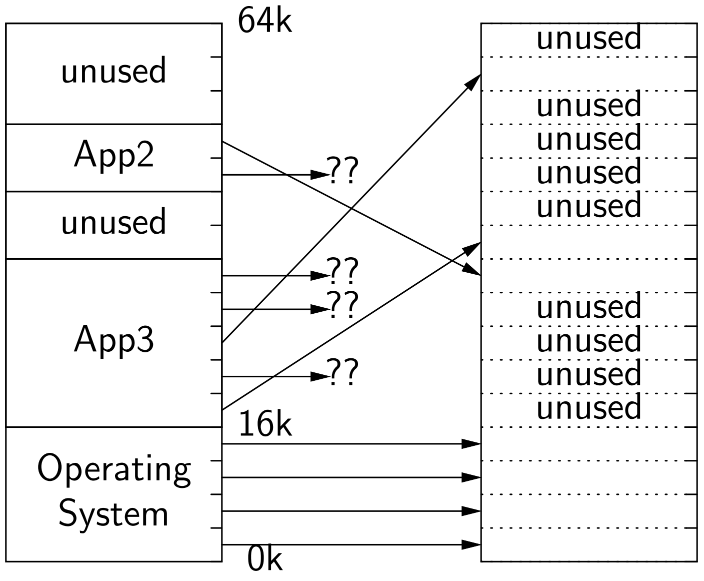
37 / 41

Other Control Bits

- *Page referenced*: used to decide which pages aren't in use right now, and hence can be paged out
- *Modified*: "dirty" pages have to be written to disk; unmodified pages may still be there from the last time they were paged out
- Memory protection: used to limit how some pages are used

38 / 41

Unmapped Pages



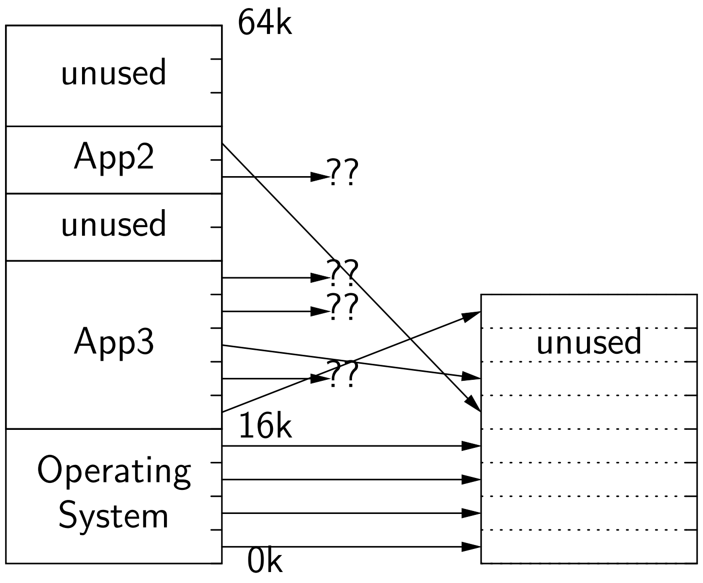
39 / 41

Unmapped Pages

- Not all pages need to exist in main memory at the same time
- We can have a larger virtual memory than physical memory
- Again, locality of reference helps — most of an application isn't needed most of the time
- Deciding what pages to keep in memory is crucial to system performance

40 / 41

Less RAM Needed



41 / 41