Memories…

- program code must be "memory resident" to be executed
- load modules, stored as disk files, are loaded into memory so program can run
- where does the loader put things?

Memory Hierarchy

- program code can be executed if it is in any of these
- cache
- main memory
- magnetic disks
- optical disks
- magnetic tape

Loader’s View of Memory

- the loader only has to be concerned about loading into “main memory”
- once program is running:
  - hardware handles what’s in cache
  - instructions handle what’s in registers
  - cache is transparent to instructions
- available memory = total memory - memory occupied by OS
- how to make memory space available to programs?

Fixed Partitioning

- divide available memory into fixed-size partitions
- process of size ≤ partition size can be loaded into available partition
- placement policy:
  - put load module into any available partition
  - if no avbl, swap some process out to free a partition
Fixed Partitioning

- problems:
  - program may be too big to fit (have to use overlays)
  - waste memory: unused space within a partition is unavailable ⇒ internal fragmentation

- doesn’t fit

Fixed Partitioning: unequal sized

- partitions are still fixed-size, but are of different sizes
  - can run bigger programs
  - more efficient usage for small programs
  - reduces but does not solve internal fragmentation problem

Fixed Partitioning: unequal sized

- placement policy:
  - assign proc to smallest partition that’s big enough ⇒ queue for each partition
  - minimizes wastage within a partition
  - but still wastes memory overall

- fits!

Fixed Partitioning: unequal sized

- placement policy:
  - assign proc to smallest partition that’s big enough ⇒ queue for each partition
  - minimizes wastage within a partition
  - but still wastes memory overall

- or:
  - single queue model
  - assign smallest available partition to arriving proc
  - when need to swap, pick smallest needed partition

Fixed Partitioning: Implementation

- attractive technique because is:
  - simple and cheap to implement
  - used in OS/MFT, OS/MVT (old IBM mainframe OS)
- unattractive technique because:
  - wastes memory because suffers from internal fragmentation
  - can waste memory and time by holding processes in queues waiting for suitable sized partition to be available

Dynamic Partitioning

- partition available memory into variable number of variable-sized partitions
- arriving process is allocated exactly the size of memory it needs
Dynamic Partitioning Problems

- cannot waste space inside partitions
  - they are made-to-order exact fits
- but can waste space outside (between) partitions
  - have "holes" in memory too small to be useful

⇒ external fragmentation

Dynamic Partitioning Problems

- resolve using compaction
  - coalesce "holes" into one big hole of available memory
  - OS relocates process images to make them contiguous
  - requires load modules that can be dynamically relocated

Compaon in Action

Dynamic Partitioning: Placement

- first fit: from beginning of memory, use first available hole that is large enough
- best fit: search all of memory and use smallest hole that is large enough
- next fit: from last allocation, look forward in memory and use first available hole that is large enough
- worst fit: scan all of memory and use the largest hole available

Dynamic Partitioning: Replacement

- need not only a placement policy, but also a replacement policy
  - used when insufficient memory is available (even after compaction)
  - and all current processes are blocked
- replacement policy determines how to pick a process to be swapped out to make room for a new process
  - defer discussion to later

Dynamic Partitioning: Placement Performance

- first is best and fastest
- next expedites external fragmentation
  - breaks up large hole at end of memory fastest
- best worse than first or next
  - creates largest number of unusably small holes
- requires most frequent compaction
- worst is worst
Buddy System

- Knuth developed a memory allocation scheme that is:
  - dynamic
  - provides memory partitions $2^k$ bytes in size
  - quite fast
- basic idea:
  - start with total available memory partition of size $2^k$ bytes
  - successively split partitions in half until smallest size that is large enough is created

Buddy System Example

- from a total available memory space of 1 Mb, need to allocate 98 Kb for a process:
  - 1024 Kb
  - 512 Kb
  - 512 Kb

Buddy System Example

- from a total available memory space of 1 Mb, need to allocate 98 Kb for a process:
  - 1024 Kb
  - 512 Kb
  - 512 Kb
  - 256 Kb
  - 256 Kb
  - 512 Kb

Handling Relocatable Code

- best flexibility when load module contains relative addresses that are bound to real addresses only as they are encountered at run time
- such load modules can be swapped out and later swapped in to a different location
- resort to hardware to do this translating of relative load module address to real memory address: dynamic address translation
Handling Relocatable Code

- addresses in load module are logical or virtual addresses
- need to be able to translate a virtual address to a real or physical address at run time
- resort to hardware for dynamic address translation

Dynamic Address Translation

- basic DAT uses special hardware and some extra registers:
  - BR: base register = address where module really starts (real or physical address of module start)
  - LR: limit register = address of end of module

Dynamic Address Translation

- DAT performs, for each virtual address:
  \[ \text{VA} + \text{BR} - \text{LR} \]
  - VA: virt addr from load module
  - BR: base reg set when image loaded
  - LR: real address of target limit register
  - within range: access this location in real memory
  - not within range: generate MMU trap

Fixed Partitions, Again

- fixed-sized partitions made memory management easy
- suffered from internal fragmentation
  - but what if partitions are small?
  - then program may not fit
  - but what if we don’t need to put entire program into one partition?

Paging

- now we do to memory what Gutenberg did for printing...
- break up all of memory into small fixed-sized partitions called frames
  - typically 2 to 4 Kb in size
  - each frame can hold exactly one page
  - a process is given a number of pages of memory
  - e.g., 98 Kb gets 49 2-Kb pages, wastes 0 bytes
  - e.g., 98 Kb gets 25 4-Kb pages, wastes 2 Kb on last page
DAT and Paging

- We used to look at addresses as:

- Now, we look at them differently:
  - For 4 KB pages (hence 4 KB frames)
  - \( \log_2(4096) = 12 \)
  - Need 12 bits to represent byte offset within a page
  - 20-bit page number = 1,048,576 pages

Page Tables

- Don’t usually store page number (as in previous example)
- Each process has its own page table
- To show mapping between its pages and real memory frames
- A page of a process can be in any frame
- E.g., previous example had consecutive frames, but this is not necessary

Paging Example

- 15 available page frames in main memory (example from Stallings):
  - Now we load processes A, B, and C

- 15 frames now loaded with 4 pages of process A, 3 pages of process B, and 4 pages of process C
  - What happens now when we swap B out?
Paging Example

15 frames now loaded with 4 pages of process A, and 4 pages of process C. Process B has been swapped out.

what happens now when we load new process D, needing 5 pages?

An Alternative to Paging

• divide program into segments
  • segments can be variable-size (up to some max)
  • this doesn’t mean segment size varies as program run, only that individual segments can be different sizes
  • virtual addresses are now combination of segment number + offset into segment
  • processes have segment tables
  • map segment number to real start address of segment
  • also have length for each segment

DAT and Segmentation
Comparison of Addressing Modes

0001010111011110
Relative Address=5598.

Segment 0
Segment 1

Comparison of Addressing Modes

0001010111011110
Page 5, offset 478

Segment 0
Segment 1

Comparison of Addressing Modes

0001010111011110
Segment 1, offset 1502

Segment 0
Segment 1

Page Tables

- each process has its own page table
- for page size of 4 Kb, with 32 bit addresses, have 20 bits for PTE entries, i.e., 1,048,576
  - minimally 4 bytes per entry
  - usually more: ≥ 1 additional byte for status flags
  - so, may have 5 Mb of page table per process
- other architectures have smaller pages
  - e.g., VAX uses 512-byte pages so has \(2^{22}\) possible PTE entries
  ⇒ page tables can be big

Handling Large Page Tables: TLBs

- can cache recently used PTEs
- use specialized cache called a translation look-aside buffer (TLB)
  - special because it can search entire cache simultaneously
  - uses associative memory (good but very $$$)
  - same problems as other caches, e.g.,
    - have to be able to replace entries if need to add a new one and table is full
    - have to flush entire TLB when change page table as, e.g., context switch

Page Tables

- hardware used for DAT to make acceptably fast
- small page tables fit into special register structure within MMU
  - but usually no more than 1024 PTEs
- larger page tables have to be in main memory
  - what happens to speed?
  - potentially 2 memory accesses per access
    - one to look-up PTE
    - another, using PTE, to get target
Handling Large Page Tables: TLBs

- Using TLB associative lookup differs from what we've seen so far:

  ![Diagram of TLB associative lookup]

Handling Large Page Tables: TLBs

- If page number is not in TLB we have a "miss":
  - Same as cache miss
  - Have to resort to looking up in full page table, in memory

Paging and Caches

- How does a cache memory fit into this scheme?
  - Sits "downstream" of the paging DAT:

Handling Large Page Tables: Multi-Level

- Can increase number of levels in PTE hierarchy
- Makes upper level table smaller
- Some paging systems decompose virtual address:
  - Page dir
  - Page num
  - Offset
  - Page dir finds a table entry pointing to one of a set of separate table of PTEs

Handling Large Page Tables: Multi-Level

- Real address
Handling Large Page Tables: Multi-Level

- We are, in effect, paging the page table.
- The higher level table (page dir) may have 10 bits for entry numbers.
  - Hence 1024 entries, reasonable once again.
- Then second level entries are 10 bits.
- Page offsets (for 4 Kb pages) are, 12 bits, as before.
- Supported on Intel Pentium, Motorola 68030, SPARC.

Handling Large Page Tables: Inverted Table

- So far, page tables contain PTEs for each possible virtual page and each process has its own (possibly very large) table.
- Inverted table: have one table with entries for real pages.
  - Fixed size because real memory is fixed size.
- How does process determine if its page i is resident?

Inverted Page Tables

- Compute hash function using page number in virtual address: get entry in table.
  - Match actual page against table entry.
  - Short chains possible from table entries.
- Used by Motorola PowerPC series processors.

Combinations

- Possible to combine segmented and paged memory.
- Segmented–paged:
  - Virtual address consists of segment number and offset.
  - Resolution of segment number + offset gives an address that is page number + offset.
  - Resolution of page number + offset gives real address.

The Question...

- Given that we can:
  - Make all memory references use virtual addresses.
  - The virtual addresses are translated to real addresses at run time using special hardware.
  - We break processes into small chunks that do not have to be contiguous in real memory.
- Then:
  - Do we need to put all of the program into memory at the same time?
  - How many of its segments or pages need to be resident at a time?
The Answer

- no! we don’t need all of it at once
- step by step:
  - start a new process = load first page into a frame
  - begin executing on that first page
  - at some point we reference a location not on that page: now what?

Getting a new page

- can’t find a PTE for page being referenced ⇒ no frame holds the page we want
- result is a trap
  - page fault in paged systems
  - segment fault in segmented systems (not to be confused with SIGSEGV)

Getting a new page

- OS must locate needed page
  - on disk somewhere, usually
- OS must load needed page into a frame
  - easy if a frame is available
  - harder if no frames are available
- need to re-try the instruction that caused the page fault
  - Motorola 68000 couldn’t do this

Some Implications

- we have more memory ‘free’ since we only have the parts (pages or segments) we ‘need’ in memory at a time ⇒
  1. can run processes that use more [real] memory than physically exists on the computer
     - but cannot exceed real memory + disk space allocated to holding pages

- we have more memory ‘free’ since we only have the parts (pages or segments) we ‘need’ in memory at a time ⇒
  1. can run processes that use more [real] memory than physically exists on the computer
     - but cannot exceed real memory + disk space allocated to holding pages
  2. can put more programs in memory at a time
Some Terms

- **virtual memory**: total memory region consisting of entirety of real memory + some amount of secondary (usually disk) space
- **resident set**: the set of pages (or segments) of a process currently in real memory
- **page fault**: the consequence of accessing a page not resident
  - resolution of page faults is expensive in time

Page Faults

- are expensive time-wise
  - so want to minimize their occurrence
  - if a frame is available, the needed page can be loaded into that frame
  - if no frame is available, then:
    - need to find a frame whose content can be replaced (with the newly needed page)
    - several algorithms to choose from for selecting a frame
    - matters if page in frame is modified or not?

Time Cost of Page Fault

- if no page fault, then memory access time is ≈ 100 ns
- if page fault:
  - trap handler
  - OS intervention
  - disk I/O
    - dominates timing, ≈ 25 milliseconds
    - disk space for holding pages (paging space) usually not organized same way as ordinary file systems so as to enhance paging performance
- effective access time = (1 – p)(100 ns) + p(25 ms)
  - p is incidence of page faults, 0 ≤ p ≤ 1
  - could be worse?

Virtual Memory Issues

- how big should a page be?
- how many frames do we give a process?
- how do we pick pages for replacement?
pages large enough to contain entire process: will never have a PF

more pages in RS ⇒ pages capture locality clusters so few PFs

increasing page size ⇒ fewer resident pages ⇒ poorer locality performance ⇒ more page faults

pages become large enough to contain significant parts of the program, reducing PFs

PFR for fixed page size with varying number of page frames
Of Page Sizes and Page Faults

- locality plays major role in PFR
- current programming practices (e.g., OOP) tend to reduce locality
- increased concurrency in implementations (e.g., multi-threaded) tends to cause sudden shifts in locality
- TLB hit rate goes down as
  - process size goes up and
  - locality goes down

- make TLB bigger?
  - expensive
  - side effects (e.g., degradation of cache effectiveness, number of accesses per cycle)
- make pages bigger so fewer TLB entries?
  - but bigger pages can increase PFR
- provide multiple page sizes
  - hardware supports now (UltraSPARC [8 Kb – 4 Mb], Pentium [4 Kb or 4 Mb])
  - software, OS, lag in supporting

Memory Management in a Paged Universe

- “As we shall see, the task of memory management in a paging environment is fiendishly complex”
  - Stallings
- organize our approach:
  - set of policies governing memory operation
  - base on:
    - real mem size
    - relative speed of main to secondary memory
    - size & number of processes
    - execution behaviour of individual programs

Virtual Memory: Policy Issues

- fetch policy: when do we bring a page in?
  - when it is referenced and not present (i.e., on a page fault)? \(\Rightarrow\) demand paging
  - ahead of time: need some way to ‘know’ what page(s) will be needed \(\Rightarrow\) prepaging

- placement policy: where do you put what you fetched?
  - less a paging problem (but see next item)
  - in segmented virt mem.: use first-, next-, best-fit

- replacement policy: how to select a page to be replaced?
  - how best to select page to replace with new page
  - maybe have to save the selected page, maybe not
  - frames may be “locked” so pages never replaced
    - e.g., real-time
    - e.g., kernel
Reference String

- list of page numbers used on successive memory accesses
  - e.g., 232152443252
- tool to help us track virtual memory replacement policies

Replacement Policies

- **Least Recently Used (LRU)**: select for replacement the page that has been least recently accessed
  - how to know what “least recently” is:
    1. timer: each frame has a time register holding time of last access
    2. stack: push page number onto stack with each access
      - need pointers to top and bottom of stack
      - need pointers into stack: pages can be moved from anywhere in stack to top
    3. other tricks (we’ll see later)

Replacement Policies: LRU

- how to know what “least recently” is:
  1. timer: each frame has a time register holding time of last access
  2. stack: push page number onto stack with each access
    - need pointers to top and bottom of stack
    - need pointers into stack: pages can be moved from anywhere in stack to top
  3. other tricks (we’ll see later)

Replacement Policies: LRU

- why does LRU work?
  - locality: pages long unreferenced are less likely to be accessed than pages more recently accessed
  - try it out with 3 frames of memory and reference string (from SG&G):
    70120304230321201701

Replacement Policies: LRU

- why does LRU work?
  - locality: pages long unreferenced are less likely to be accessed than pages more recently accessed
  - try it out with 3 frames of memory and reference string (from SG&G):
    70120304230321201701

Replacement Policies: LRU

- why does LRU work?
  - locality: pages long unreferenced are less likely to be accessed than pages more recently accessed
  - try it out with 3 frames of memory and reference string (from SG&G):
    70120304230321201701
Replacement Policies: LRU

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PF</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Replacement Policies: LRU

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PF</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Replacement Policies: LRU

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PF</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Replacement Policies: LRU

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PF</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

© Charles M Snow 17
Replacement Policies: LRU

4: 1
0: -2
3: -3

4: 2
0: -3
2: -1

Replacement Policies: FIFO

First-in First-Out (FIFO): select for replacement the oldest page
- view frames allocated to process as circular buffer, pages removed round-robin
- why FIFO works: page that’s been around longest is least likely to be needed
- except, e.g., if you’re doing a loop…

Replacement Policies: FIFO

FIFO with short reference string, 3 frames

PFs: 1
Replacement Policies: FIFO

- FIFO Anomaly
  - of page replacement policies, FIFO shows the anomalous behaviour of having more PFs when its frame allocation is increased
  - reported by Belady, Nelson & Shedler in 1969: usually attributed to Belady ("Belady's Anomaly")
  - Belady did other work...

BOR

- Belady Optimal Replacement (BOR): select for replacement the page that will not be used for the longest time in the future
- unimplementable
  - with current technology and understanding of the universe

FIFO Anomaly

- more frames: more page faults?!!

BOR example

- 7:14
- 0:1
- 1:10
- PFs: 4
**Comparing Methods**

- with reference string, so far we have:
  - LRU: 12 page faults
  - FIFO: 15 page faults
  - BOR: 9 page faults
- we will examine other methods

**Words About the Midterm...**

- watch the course web page to know details of where you write the exam
- midterm covers everything covered in class including today
- exam will run 2.5 hours (16:30 – 19:00)
- questions will be short answer, fill-in blanks, brief explanations
- any code to write will be pseudo-code
  - no real programming
- no notes or crib-sheets; closed-book