Opportunity

Memory references are dynamically translated into physical addresses at run time.
   A process may be swapped in and out of main memory such that it occupies different regions.

A process may be broken up into pieces that do not need to be located contiguously in main memory.

Not all pieces of a process need to be loaded in main memory at once during execution.

Execution of a Program

Operating system brings into main memory a few pieces of the program.

*Resident set* - portion of process that is in main memory.

An interrupt is generated when an address is needed that is not in main memory - *page fault*.

Operating system places the process in a blocking state.

Piece of process that contains the logical address is brought into main memory.
   Operating system issues a disk I/O Read request.
   Another process is dispatched to run while the disk I/O takes place.
   An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state.
## Advantages of Breaking up a Process

More processes may be maintained in main memory
- Only load in some of the pieces of each process
- With so many processes in main memory, it is very likely at least one process will be in the Ready/Run state at any particular time

A process may be larger than all of main memory

## Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real memory</td>
<td>physical memory, RAM, main memory</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>secondary storage holding process images</td>
</tr>
<tr>
<td>Thrashing</td>
<td>swapping out a piece of a process shortly before that piece will be needed</td>
</tr>
<tr>
<td>Locality</td>
<td>program and data references within a process tend to cluster; implies that only a few pieces of a process will be needed over a short period of time; possible to make intelligent guesses about which pieces will be needed in the future; suggests that virtual memory may work efficiently</td>
</tr>
</tbody>
</table>
### Paging

Each process has its own **page table**

Each page table entry contains the frame number of the corresponding page in main memory

A **resident bit** is needed to indicate whether the page is currently in main memory

**Modify bit** is needed to indicate if the page has been altered since it was last loaded into main memory

If no change has been made, the page does not have to be written to the disk when it needs to be swapped out

---

### Address Translation

If page size is $2^k$...

- Page # is leftmost $n$ bits of virtual address...
  - ...so no arithmetic is necessary to extract it...

Frame # is just concatenated with offset to obtain physical address
Cost of Page Indexing

The entire page table may take up too much main memory:
- each process may typically be allowed a virtual memory space of 2 GB or more
- given 4 KB pages, and a 4GB virtual space, there could be \(2^{20}\) page table entries for each process
- each page table entry would occupy, say, 4 bytes of space, so the page table would occupy 4 MB of memory (per process), or \(2^{10}\) frames/pages

Therefore, page tables are also stored in virtual memory and loaded into main memory as needed.

The \(2^{20}\) page table entries can be efficiently indexed using a 2-level structure:
- the \(2^{10}\) pages of the page table can be indexed via a root page table with \(2^{10}\) entries, needing only 4 KB (one page) of main memory
- lock the root page table in main memory...

Two-Level Scheme for 32-bit Address

Root page table needs only 1 page of main memory
Root page table provides virtual page # of relevant section of process page table

Each page of the process page table provides access to \(2^{10}\) pages of the process in virtual space.
Page Tables

The entire page table may take up too much main memory
Page tables are also stored in virtual memory
When a process is running, part of its page table is in main memory

Inverted page table
- page number portion of a virtual address is mapped into a hash value
- hash value points to inverted page table
- fixed proportion of real memory is required for the tables regardless of the number of processes
- used on PowerPC, UltraSPARC, and IA-64 architecture

Inverted Page Table
Each virtual memory reference can cause two physical memory accesses
- one to fetch the page table
- one to fetch the data

To overcome this problem a high-speed cache is set up for page table entries
- called a Translation Lookaside Buffer (TLB)
- contains page table entries that have been most recently used

Given a virtual address, processor examines the TLB

If page table entry is present (**TLB hit**):
- the frame number is retrieved and the real address is formed

If page table entry is not found in the TLB (**TLB miss**):
- the page number is used to index the process page table
- if page is already in main memory, proceed
- if not, trigger a page fault
- update TLB to index the new page
Paged VM

Direct Mapping

Page # is index of corresponding entry in the page table, so the relevant table entry can be found in $O(1)$ time via the page table.

Virtual Address

Page # Offset

Page Table

Frame # Offset Real Address
Associative Mapping

Unfortunately, the TLB does not contain all of the entries that would be in the page table.

So, direct mapping won’t work in the TLB.

But, we still need efficient lookup.

The TLB would support fully associative lookup… in simplest terms, every location in the TLB can be compared to the desired value at once, so lookup will still be $O(1)$.

VM/Cache Interaction

Virtual address is resolved into a physical address by using the TLB and page table.

Physical address is passed to cache manager.

If requested location is not in the cache, it is fetched from main memory into the cache and then served up to the process.
Page Size

Small pages:
- less amount of internal fragmentation
- more pages required per process
- more pages per process means larger page tables
- larger page tables means large portion of page tables in virtual memory
- larger number of pages may be found in main memory, so reduced # of page faults
- process makes more page to page transitions, so more chances for a page fault

Large pages:
- secondary memory is designed to efficiently transfer large blocks of data so a large page size is better
- pages may be more likely to contain references to far-away locations, leading to increased # of page faults

Expectation: as time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.

Empirical Paging Results

As page size increases, the page fault rate initially rises since the process has fewer pages resident in main memory, then falls to zero as page size approaches the size of the process image.

As the number of allocated frames approaches the number of pages N in the process image, page faults drop to zero (surprise!).

The decrease is most rapid before the frame allocation is sufficient to hold the working set of the process.
Fetch Policy

- determines when a page should be brought into memory

Demand paging only brings pages into main memory when a reference is made to a location on the page
- many page faults when process first started
- fetches only pages that are actually needed by the process
- easy to make decision
- universal choice for deployed paged VM systems

Prepaging brings in more pages than needed
- more efficient to bring in pages that reside contiguously on the disk
- need some criteria for picking pages to pre-fetch, no good ones seem to exist
- also called anticipatory fetching
- intuitively appealing, not established as effective
Replacement Policy

Replacement policy decides which resident page will be replaced.
- Belady’s Optimal policy: page removed should be the page least likely to be referenced in the near future
- Belady’s Optimal policy is clearly infeasible
- most policies predict the future behavior on the basis of past behavior

If selected page has been modified it must be written back to virtual memory before being overwritten.

Frame Locking
- if frame is locked, it may not be replaced
- kernel of the operating system
- control structures
- I/O buffers
- associate a lock bit with each frame

Basic Replacement Algorithms

Optimal policy
Selects for replacement that page for which the time to the next reference is the longest
Impossible to have perfect knowledge of future events

Least Recently Used (LRU)
Replaces the page that has not been referenced for the longest time
By the principle of locality, this should be the page least likely to be referenced in the near future
Each page could be tagged with the time of last reference. This would require a great deal of overhead.

First-in, first-out (FIFO)
Treats page frames allocated to a process as a circular buffer
Pages are removed in round-robin style
Simplest replacement policy to implement
Page that has been in memory the longest is replaced
These pages may be needed again very soon
Clock Replacement Algorithm

Clock Policy

- Additional bit called a use bit
- When a page is first loaded in memory, the use bit is set to 1
- When the page is referenced, the use bit is set to 1
- When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
- During the search for replacement, each use bit set to 1 is changed to 0

Examples

<table>
<thead>
<tr>
<th>Page address stream</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>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F = page fault occurring after the frame allocation is initially filled
**Empirical Comparison**

Experimental simulation:
- 250,000 address references
- fixed frame allocation

Comparison:
- largest differences with small allocations of frames
- all four protocols have expected theoretical shape (slide 18)
- FIFO is about twice as bad as Belady’s Optimal, and worst overall in all cases
- ideal performance would seem to result from being close to the “knee” of the curve

---

**Page Buffering**

LRU and the Clock Algorithm are generally superior to FIFO, but they are more complex and involve more overhead.

Also, it costs more to replace a modified page than an unmodified page.

Page Buffering
- basic strategy is FIFO, but…
- page table entry for replaced page is added to one of two lists:
  - free page list, if page has not been modified
  - modified page list, otherwise
- try to keep at least a few frames free at all times
- pages are not removed from memory until they MUST be overwritten
- modified pages are replaced only if there is no free frame
- used in VAX/VMS
Resident Set Size

How many frames should be allocated to a process?

Fixed-allocation
   Gives a process a fixed number of pages within which to execute
   When a page fault occurs, one of the pages of that process must be replaced

Variable-allocation
   Number of pages allocated to a process varies over the lifetime of the process

Fixed Allocation, Local Scope

Decide ahead of time the amount of allocation to give a process

If allocation is too small, there will be a high page fault rate

If allocation is too large there will be too few programs in main memory
Variable Allocation, Global Scope

Easiest to implement

Adopted by many operating systems

Operating system keeps list of free frames

Free frame is added to resident set of process when a page fault occurs

If no free frame, replaces one from another process

Variable Allocation, Local Scope

When new process added, allocate number of page frames based on application type, program request, or other criteria

When page fault occurs, select page from among the resident set of the process that suffers the fault

Reevaluate allocation from time to time
Cleaning Policy

Demand cleaning
A page is written out only when it has been selected for replacement

Precleaning
Pages are written out in batches

Best approach uses page buffering
Replaced pages are placed in two lists
Modified and unmodified
Pages in the modified list are periodically written out in batches
Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

Load Control

Determines the number of processes that will be resident in main memory

Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping

Too many processes will lead to thrashing
**Process Suspension**

- Lowest priority process
- Faulting process
  - This process does not have its working set in main memory so it will be blocked anyway
- Last process activated
  - This process is least likely to have its working set resident
- Process with smallest resident set
  - This process requires the least future effort to reload
- Largest process
  - Obtains the most free frames
- Process with the largest remaining execution window

**Linux Memory Management**

- Page directory
- Page middle directory
- Page table