Memory Hierarchy

Ideally, programmers would like a fast, infinitely large nonvolatile memory.

In reality, computers have a memory hierarchy:

Cache (SRAMS): Small (KBytes), expensive, volatile and very fast (< 10 nanosecs).

Main Memory (DRAM): Larger (MBytes), medium-priced, volatile and medium-speed (50-100 nanosecs).

Disk: GBytes, low-priced, non-volatile and slow (seek 10s milliseconds).

Therefore, the OS must manage these limited resources in order to provide the illusion of a fast, infinitely large main memory.

Memory Manager responsibilities:

- Track usage.
- Allocate/Deallocate memory.
- Manage swapping between main memory and disk.

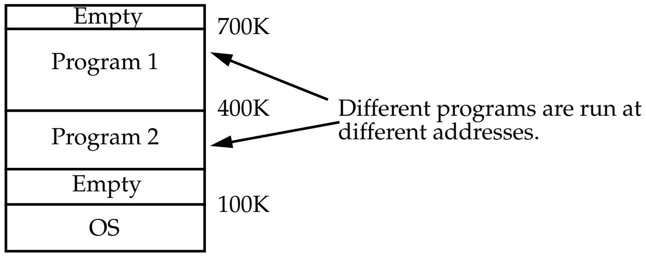
Memory management in a multiprogrammed environment:

Multiprogramming introduces two problems:

Relocation: A mechanism to allow a process to execute anywhere in physical memory.

Protection: A mechanism to protect one process from reading or writing to another process's address space.

Relocation

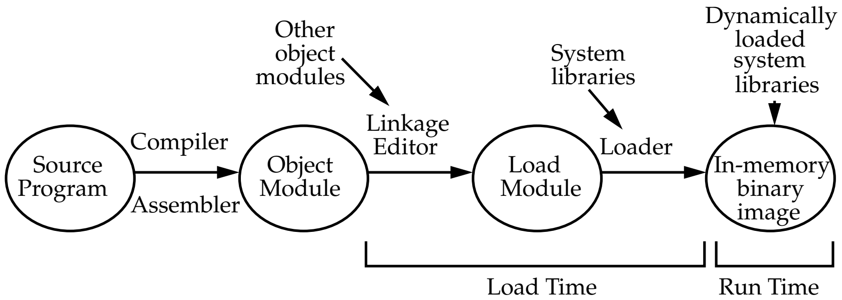


In a multiprogrammed environment, a program can be loaded in any part of physical memory to be executed.

Suppose Program 1's executable has instruction 'jmp to address 100'.

Without relocation, this will only work if the program is loaded at address 0 in physical memory.

Typical processing of a user program:



Compiler typically binds symbolic addresses in user programs (e.g. a user variable **count**) to relocatable addresses, e.g., 14 bytes from the beginning of this module.

One possibility is that the linkage editor or loader can then bind these relocatable addresses to absolute addresses at load time.

Another possibility is that the mapping is performed at execution time using special hardware.

Solutions to relocation:

One solution is to modify the instructions as the program is loaded into memory.

Loader performs **relocation** by using list or bit map in executable's header to identify instructions that require address adjustments.

- For example, Program 1 needs 400K added to each absolute memory reference.

Still have **protection** problem:

- This does not prevent malicious programs from building an instruction at run time that accesses or modifies another program's address space.

Another solution is to use **base** and **limit** registers.

- When a process is scheduled, base register is loaded with start address of partition.
- All memory references are translated using this value (relocation).
- All memory references are checked against the limit (protection).

Hardware solution has advantage that process can be **moved during its execution**. We will consider other execution time solutions soon.

Static Linking vs. **Dynamic Linking**

If OS supports only **static** linking, loader combines system libraries in a single binary image.

- Wastes disk space and main memory since user programs have a copy of system routines included in their binary.

If OS supports **dynamically linked** libraries, linking of system libraries is postponed until execution time.

- A stub routine is included in the binary for each library reference.
- Stub indicates how to locate or load the appropriate library routine.
- Stub replaces itself with the address of the routine when executed.

Advantages:

- Only one memory resident copy of the routine is needed, all processes reference it.
- Bug fixes to library routines do not require all programs that use the routine to be recompiled.

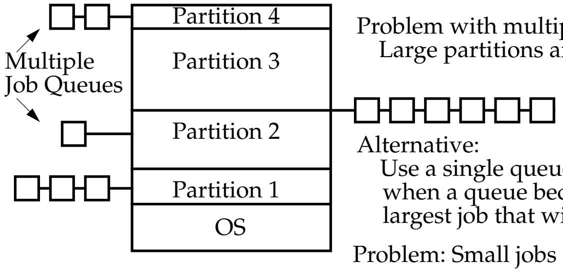
Version information is included in both the binaries and the library to prevent processes from executing new, incompatible versions.

Dynamic linking requires help from the OS.

- Normally, memory protection hardware prevents one process from accessing another's address space.
- In this case, OS must allow a process to access another process's address space so that sharing is possible.

Simple Memory Management:

Multiprogramming can be achieved simply by dividing up memory into n (possibly unequal) fixed sized partitions defined at system start-up.



Problem with multiple job queues:
Large partitions are wasted.

Alternative:
Use a single queue which is searched when a queue becomes empty for largest job that will fit.

Problem: Small jobs are starved.

These memory management schemes may be sufficient in batch processing systems.

Advantage: They are very simple to implement.

Note that the hard disk is not used in these memory management schemes.

In time sharing systems, queueing up jobs in this manner would lead to unacceptable response time for user processes.

The hard disk can be used to allow more processes to run than would normally fit in main memory.

When a process blocks for I/O (e.g. keyboard input), it can be swapped out to disk, allowing other processes to run, improving response time.

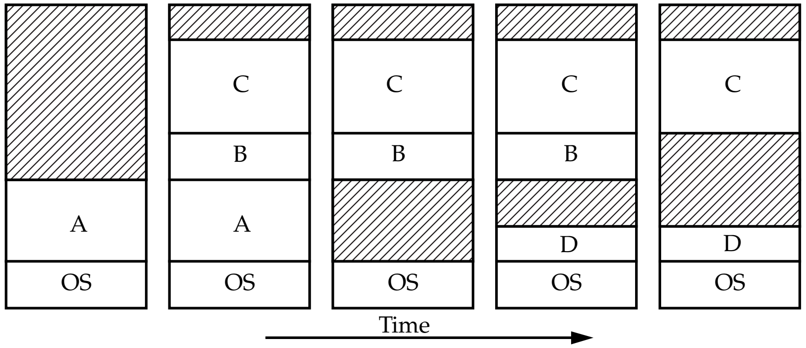
The movement of whole processes to and from disk is called **swapping**.

A swap partition on a hard disk provides for another possibility; the execution of processes that are larger than physical memory.

If a OS supports **virtual memory**, it allows the execution of processes that are only partially present in main memory.

Swapping:

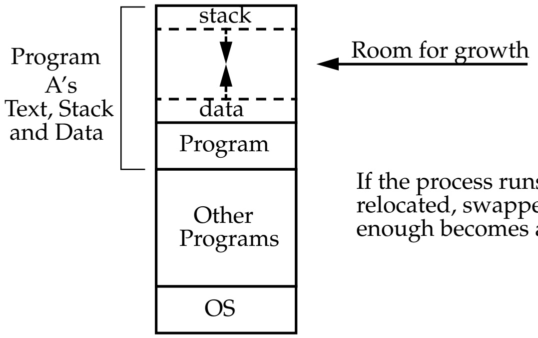
Unlike fixed size scheme, number, location and size of partitions vary dynamically in a swapping scheme, e.g.,



Initially process A is in memory.
Then B and C are created or swapped in from disk.
A either terminates or is swapped out, etc.

Problems:

- Dynamic partition size increases memory utilization but complicates allocation/deallocation by causing holes, **external fragmentation**.
  - Memory compaction is rarely used because of the CPU time involved.
- Processes' data segments can change dynamically, e.g. malloc()
  - If process does not have room to grow, it will need to be moved or other adjacent processes will need to be swapped out.



If the process runs out of space, it must be relocated, swapped out until a hole large enough becomes available or killed.

Virtual Memory

Combined size of the process (program, data and stack) may exceed the amount of physical memory (installed DRAM).

OS keeps the parts of the process that are currently in use in main memory and the rest of the process on disk.

When a new portion of the process is needed, the OS overwrites a portion of main memory with the new fragment.

Virtual memory also works in a multiprogrammed system.

- Main memory is composed of bits and pieces of many programs.
- A process blocks whenever it requires a portion of itself that is on disk, much in the same way it blocks to do I/O.
- OS can schedule another process to run until this is accomplished.

Paging

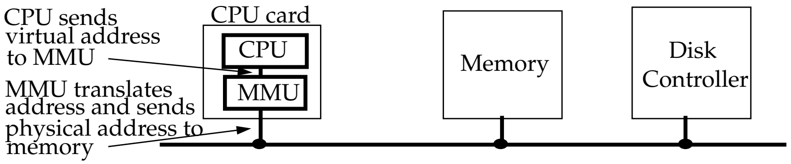
Most virtual memory systems use a technique call **paging**.

We will refer to addresses which appear on the address bus of main memory as a **physical** addresses.

Programs generate **virtual** addresses, e.g., to move the contents of a memory location 1000 into a register, Mov EAX, [1000]

Without virtual memory, a virtual address generated by a process is a physical address.

When virtual memory is used, virtual addresses are sent to a Memory Management Unit (MMU) which translates the virtual address to a physical address.



The virtual address space is divided up into **pages**, 512 to 64K in size.

Corresponding units in physical memory are called **page frames**.

Pages and page frames are always the same size.

For example,

Assume page size is 4K.

Assume physical memory is 32K.

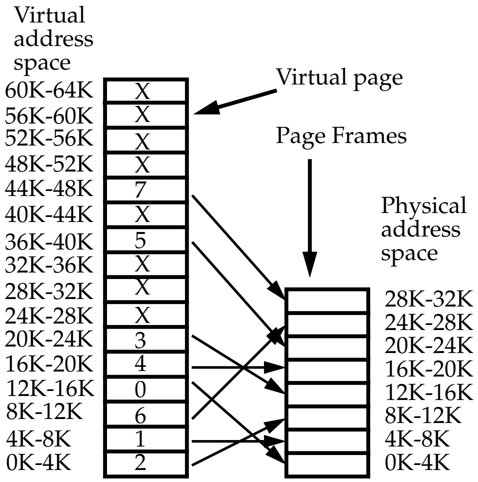
Assume virtual memory is 64K.

Therefore, we have 16 virtual pages and 8 page frames.

Program issued virtual address:

0 translated to 8192

20500 translated to 12K + 20 = 12308



Note that 8 virtual pages are not mapped into physical memory (indicated by an X on the previous slide).

A **present/absent** bit in the hardware indicates which virtual pages are mapped and which ones are not.

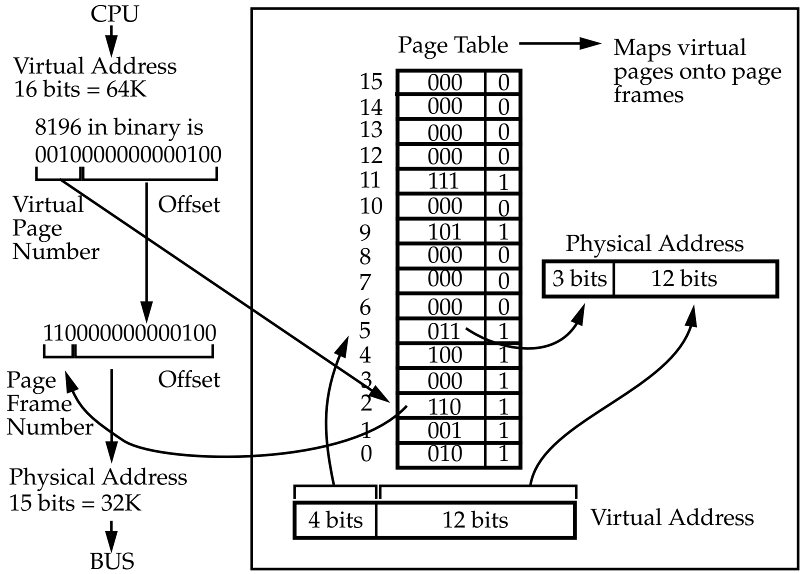
What happens when a process issues an address to an unmapped page ?

- MMU notes page is unmapped using present/absent bit.
- MMU causes CPU to trap to OS - **page fault**.
- OS selects a page frame to replace and saves it current contents to disk.
- OS fetches the page referenced and places it into the free page frame.
- OS changes the map and **restarts** the instruction that caused the trap.

Note: Paging allows the physical address space a process to be **noncontiguous** !

Provides a solution to the **external fragmentation** problem of partition schemes.

Addresses Translation by the MMU:



Two important issues of the Page Table:
- Size
- Performance

Size:

The Pentium uses 32 bit virtual addresses. With a 4K page size, a 32-bit address space has $2^{32}/2^{12} = 2^{20}$ or 1,048,576 virtual page numbers !

For 64-bit machines, there are 2^{52} virtual page numbers !!!

Performance:

The mapping from virtual-to-physical addresses must be done for EVERY memory reference.

Every instruction fetch requires a memory reference.

Many instructions have 1 or more memory operands.

Therefore, the mapping must be extremely fast, a couple nanoseconds otherwise it becomes the bottleneck.

Page table design alternatives:

- Single page table consisting of an array of fast hardware registers.
  OS loads registers from memory when a process is started.
  Advantage:
  - No memory references are needed for the page table.

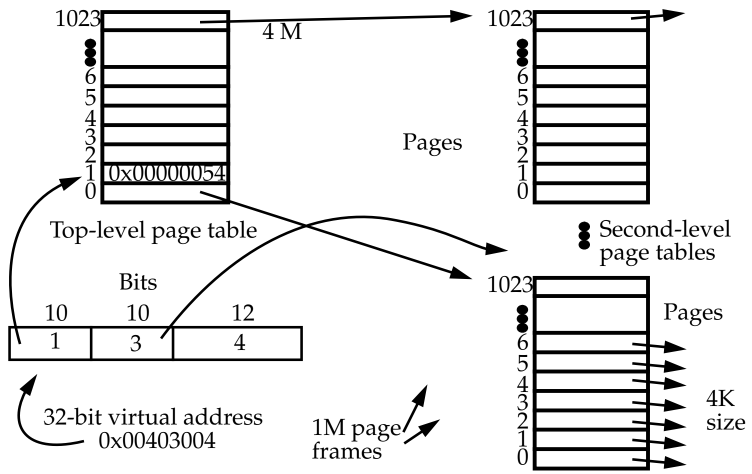
  Disadvantage:
  - Context switches require the entire page table to be loaded. If it is large, this will be expensive.

- Page table kept entirely in main memory.
  Single register points to the start of the page table.
  Advantage:
  - Context switches only require updating the register pointer.

  Disadvantage:
  - One or more memory references are needed to read page table entries for each instruction.

Real computers use hybrid approaches to be discussed.

Multilevel Page Tables: Addresses page table size problem.



Many of second-level page tables are not present in main memory.

0x00000054 is address of second-level page table. This value is added to 3*page frame size to get physical address of page. Offset then added.

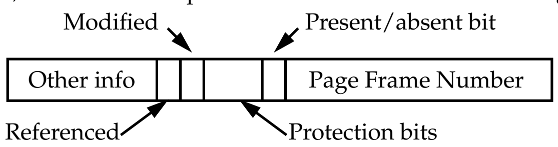
Using a two level page table, two page faults can occur for a single address reference.

If the second-level page table is not in memory, a page fault occurs.

If the page that the second-level entry refers to is not in memory, another page fault occurs.

Page Frame structure:

In general, it is machine dependent but most have the following form:



Page Frame address: physical memory address

Present/Absent bit: If 1, page is in memory, if 0, it is not.

Modified bit: If set, page has been written to, e.g. it is 'dirty'.

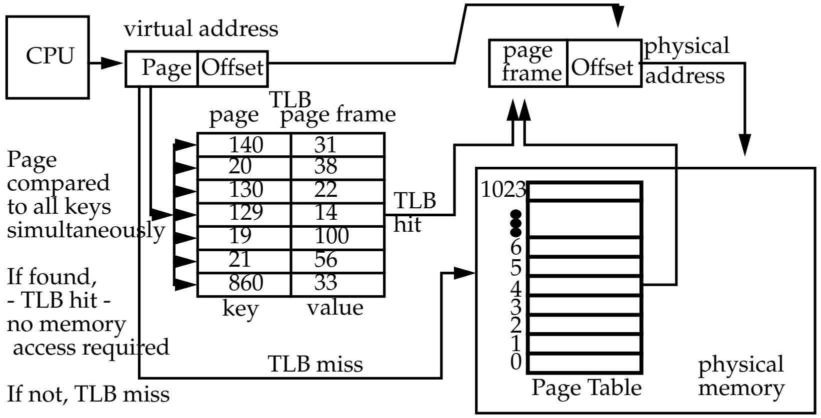
Referenced bit: Used in page replacement algorithms to be discussed.

Protection bits: Specifies if data in page can be read/written and/or executed.

Translation Lookaside Buffers (TLBs):

With two-level paging, one memory reference requires **three** memory accesses reducing performance by 2/3 !

This is unacceptable. Standard solution is to use a special, small, fast lookup hardware cache called a TLB.



Number of TLB entries varies from 8 to 2048. Typically around 64.

When a TLB miss occurs:

- A trap occurs and an OS routine (in newer processors) handles the fault. The instruction is then restarted.
- The OS routine copies one (or more) page frame(s) from the page table in memory to one (or more) of the TLB entries.

Therefore, if page is referenced again soon, a TLB hit occurs eliminating the memory reference for the page frame.

In order to reduce the number of TLB misses for page frames that contain the page tables:

A large software cache of TLB entries is usually kept in a fixed location in memory and

The page frame for that cache is locked into the TLB.

Page Replacement Algorithms:

When a page fault occurs, the OS must select a page in memory to replace with the page on the swap device.

If the page to be replaced is dirty (different than the disk copy), then it must be written out to the swap device.

If it is not dirty, then it can be overwritten with the new page.

Optimal Page Replacement Algorithm (OPR):

Replace the page that will be referenced furthest in the future.

Similar to the shortest job first scheduling algorithm, it is not realizable.

However, it does serve as a reference in evaluating other algorithms.

We can determine the performance of the OPR algorithm using simulations in which we simply run the same program twice.

Not Recently Used (**NRU**) Algorithm:

We previously identified two status bits that were associated with each page frame, Referenced and Modified.

Referenced: Set when the page is read or written.
Modified: Set when the page is modified (written).

The hardware sets these bits automatically. It is the job of the OS to clear them at the appropriate time.

Using these bits, we can implement a simple page replacement algorithm:

- At program start-up, both bits of all pages belonging to the process are cleared by the OS.
- Periodically (e.g. on each clock interrupt), the R bit is cleared to distinguish pages that have not been recently referenced from those that have been.

- At any point in time, four possible states exist:
  Class 0: Not referenced, not modified
  Class 1: Not referenced, modified
  Class 2: referenced, not modified
  Class 3: referenced, modified
  Note: Class 1 states are created by the clock interrupt.

When a page fault occurs, NRU removes a page at random from the lowest numbered nonempty class.

Advantages of NRU:
- Efficient to implement.
- Often yields adequate performance.

First-In, First-Out (**FIFO**) Algorithm:

OS maintains a list of all pages in the system.

Page at the head of the list is the oldest, page at the end is the most recent arrival.

On the next page fault, page at beginning of list removed and new page is added to the end of the list.

Rarely used because it assumes a relationship between age and frequency of reference.

It is easy to imagine a case where a process references a page continuously over an extended period of time, e.g. a recursive loop.

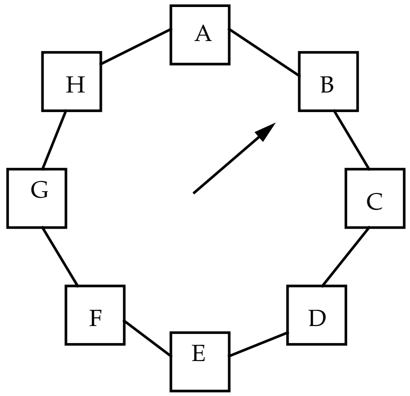
Modification: **Second Chance** (**SC**)Page Replacement Algorithm:

Simply inspect the R bit of the oldest page. If referenced, put it on the end of the list and continue the search.

If all pages have R set, SC degenerates to FIFO.

The **Clock** Algorithm:

Disadvantage of **SC**: Inefficient since it constantly moves pages in list.



Instead, keep all pages in a circular list with pointer to oldest page.
When page fault occurs, inspect oldest page.
If R is clear, evict. If R is set, clear R and move to next oldest page.

Least Recently Used (**LRU**) Algorithm:

Based on assumption that:

Pages referenced heavily in last few instructions will probably be referenced heavily in next few instructions.
Those that have not, will probably not be referenced for a long time.

Therefore, LRU evicts the page that has been **unused** for the longest time.

Disadvantage: Expensive to implement.
- Requires sorted list with most recently used page on front and least recently used page on rear.
- List must be updated on every memory reference.
- Page referenced must be found in list and moved to front of the list !

Not Frequently Used (**NFU**) Algorithm:

Closely related to LRU but realizable.

Requires a software counter associated with each page, initially zero.
At each clock interrupt, OS scans all pages in memory.

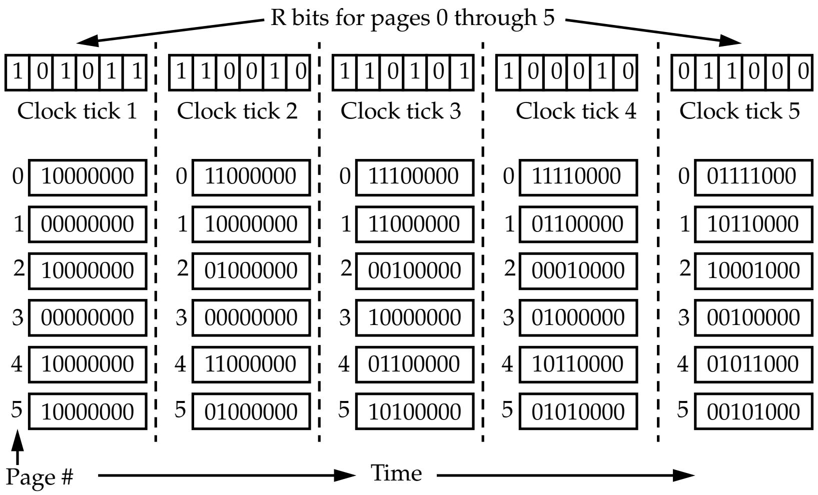
For each page, the R bit (which is zero or one) is added to the counter.

On a page fault, page with lowest value selected for replacement.

Problem: **Never forgets** anything.
Solution:

Counters shifted right one bit.
R is added to leftmost, not rightmost, bit.

NFU (**aging**) algorithm example:



At clock tick 1, pages 0, 2, 4 and 5 were referenced over the interval 0->1.
Pages show shifted counters with leftmost bits updated based on references.

It is clear, that a page that has not been referenced for x clock cycles will have x 0s shifted in on the left.

On page fault, page with lowest value of counter is evicted.

NFU differs from LRU in two ways:

- In some cases, it is not possible for the OS to determine which page was referenced furthest in the past.
  - For example, pages 3 and 5 at clock tick 4 have identical values in the left three bits.
  - Both have not been referenced for 2 clock cycles
  - Both were referenced at some time during the interval 2->3.
  - The order in which they were referenced over the clock interval 2->3 is lost.
- With an 8 bit counter, we can only record references over the last 8 clock intervals.

Design Issues for Paging Systems:

Demand Paging: Pages belonging to a process are brought into memory as page faults occur.

Locality of Reference: During any phase of a process's execution, the process references only a relatively small fraction of its pages.

The **Working Set**: The set of pages that a process is currently using. If the entire working set of pages is in memory, a process will generate relatively few page faults.

Thrashing: A process thrashes when it generates a page fault at a rate, for example, of one or two instructions per 20 milliseconds.

A process will make very little progress in this case since the service time for a page fault is 10s of milliseconds (must be read from disk).

Thrashing occurs when a process's working set is not in memory.

In time sharing systems, processes are frequently moved to disk to allow other processes to run.

The **Working Set** model:

Paging systems based on this model keep track of every process's working set and allow them to run only when they are in memory.

The alternative is to allow the process on the swap drive to run and demand page its working set back into memory.

This is expensive. If OS knows a process's working set, it could load its pages before allowing it to run (**prepaging**).

Working sets can be determined, for example, using NFU by evaluating the counters. A page could be considered in the working set if a 1 appears in one or more of the leftmost **n** positions.

The parameter **n** needs to be evaluated experimentally.

Local vs. Global Allocation Policy:

The number of pages any given process has in memory is another consideration that can be taken into account when selecting a page to evict.

Page replacement algorithms described previously select among the entire set of pages (**global**).

Local algorithms, on the other hand, also consider the total number of page frames each process possesses.

Local algorithms allocate a fixed fraction of memory to each process.
Global algorithms allow the number of page frames per process to vary over time.

Global algorithms work better when the size of working set can change over time. Thrashing occurs if a local algorithm is used and the working set increases in size.

Methods of fixing the page frame allocation size for local algorithms.

Each process gets an equal number of page frames:

Wasteful since small processes get the same number as large processes.

Each process gets a fraction proportional to its size:

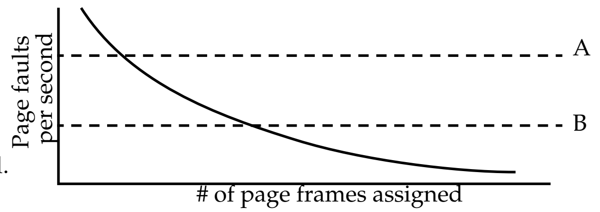
e.g., 300K processes get 30 times more page frames than 10K processes.

Neither of these approaches addresses thrashing problem directly.

Page Fault Frequency (**PFF**): A better way to control thrashing.

For most page replacement algorithms, it is known that the page fault rate decreases as more page frames are assigned.

Typical page fault rate for one process as a function of total page frames assigned.



Above dashed line A, the fault rate is unacceptable, so faulting process is given more page frames.
Below B, the fault rate is too low - process has too many page frames, and therefore, page frames can be taken away.

PPF attempts to keep the page fault rate between A and B.
PPF can use swapping if its not possible to keep all processes below A.

Page Size:

Internal fragmentation: The fraction of the final page that a process does not use and is wasted.

On average, half of the final page will be empty.

Each process typically has a text, data and stack segment. With **n** segments in memory and a page size of **p** bytes, **np/2** bytes will be wasted on internal fragmentation.

This reasoning argues for a small page size.

However, the smaller the page size, the larger the page table.
Also, transfers from disk require about as much time for a large page size (e.g. 8K) as it does for a small page (e.g. 512 bytes) because seek and rotational delay dominate transfer time.

Consider total overhead:

overhead = se/p + p/2 where

s is the average process size in bytes.
p is the page size in bytes.
e is the amount in bytes that each page **entry** requires.

Therefore, the average number of pages per process is s/p and the page table space is **se/p**.
Also, the wasted memory in the last page due to internal fragmentation is **p/2**.

First term, page table size, is large when page size is small.
Second term, internal fragmentation, is large when page size is large.

Using calculus, we can find the optimum by taking the first derivative and solving for p yielding:

p = sqrt(2se)

For example, for s = 128K and e = 8, optimal size is 1448 - approx. 2K.
Most systems use between 512 bytes to 64K page sizes.