Virtual-Memory Management

- Remember we said instructions must be in the main memory to be executed by the CPU.
  - Main memory = RAM = Physical Memory
  - Limits the size of a program. Now, it must be smaller than physical memory.
- There are stuff in programs that never or seldom runs.
  - Error handling
  - Arrays, lists and tables allocate more memory than they need.
    - We create 100 size arrays but sometimes just use 10.
  - Certain options may only run rarely.
- Even those cases where we need the entire program to be loaded into the memory, we may not need all those stuff at the same time.

- Therefore, if we are able to execute a program which is only partially in memory, it would be beneficial
  - A program would not be constrained by the available size of physical memory.
  - Users would be able to write programs for an extremely large virtual address space.
  - Each program would take less memory = more programs would be able to fit
    - More CPU utilization and throughput with no increase in response time
  - Less I/O would be needed to load or swap user programs into memory, so each program would run faster.
- Running a program that is not entirely in memory would benefit both the system and user.

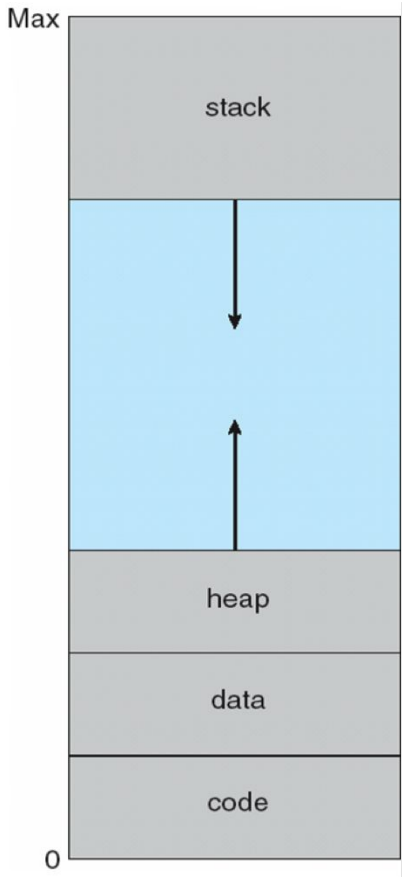
Virtual memory

- A technique that allows the execution of processes that are not completely in memory.
  - A major advantage: It allows programs to be larger than memory.
- It abstracts main memory into an extremely large storage.
  - Separates logical memory and physical memory.
- Frees programmers from concerns of memory-storage limitations.
  - Now, programming is easier because programmer no longer needs to worry about the size of physical memory.
  - Can concentrate on the problem itself.
- Allows processes to share files easily.
- Not easy to implement.
  - Decreases performance if used carelessly.

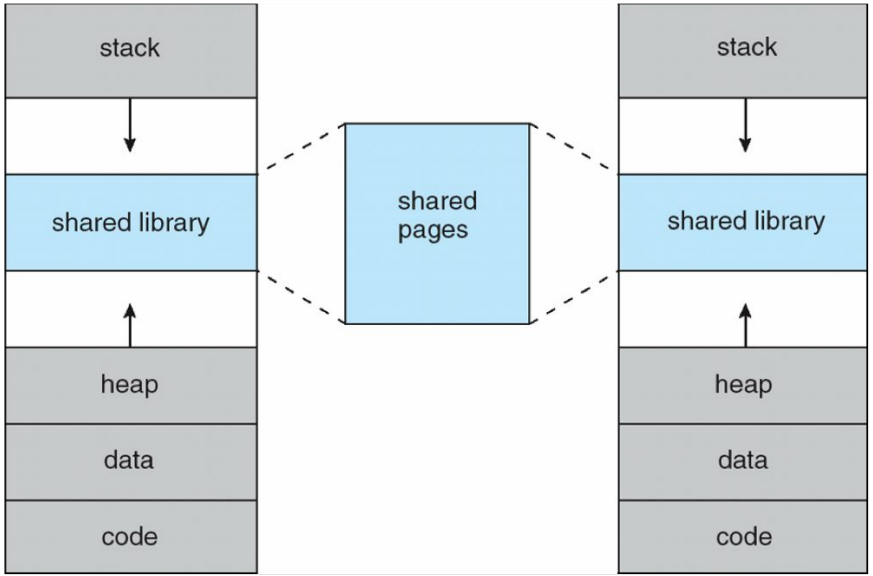
Virtual address space

- Virtual address space refers to the logical (virtual) view of how a process is stored in memory.
  - Typically it starts from a certain logical address (like 0) and exists in contiguous memory.
  - Remember that physical memory may be organized in *page frames* and physical page frames assigned to a process may not be contiguous.

- We allow heap to grow upward
  - Used for dynamic memory allocation
- We allow stack to grow downward through successive function calls.
- The large blank between the heap and stack is part of the virtual address space.
  - But require actual physical pages only if the heap or stack grows.
- Virtual addr. space that include holes are known as sparse address spaces.
- Good to use because holes can be filled as the stack or heap segments grow or if we wish to dynamically



- Virtual memory also allows files and memory to be shared by two or more processes through **page sharing**
  - System libraries can be shared by several processes through mapping of the sharing object into a virtual address space.
  - The actual pages where the libraries reside in physical memory are shared by all processes.



- Processes can share memory.
  - Two or more processes can communicate through the use of **shared memory**
- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes which are sharing this region consider it part of their virtual address space, yet the actually physical pages of memory are shared.
- Pages can be shared during process creation with **fork()** system call.

Demand paging

- We have a program and it will be loaded from disk to memory.
- One option is to load everything at once.
  - However, we may not initially need the entire program in memory.
  - Let’s say there is a program starting with a list of available options.
  - When we load all, we also every code for **all** options.
- Another way is just to load pages only as they are needed.
- This is known as **demand paging**
  - Commonly used in virtual memory systems.
- Pages are loaded only when they are demanded during program execution.
- Pages that are never accessed are thus never loaded into physical memory.

Demand paging

- Similar to swapping where processes reside in secondary memory.
- When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process, we use a **lazy swapper**
- It never swaps a page into memory unless that page will be needed.
  - In the context of demand-paging, **swapper** is not a correct word.
  - A swapper manipulates entire processes.
  - A **pager** is concerned with individual pages of a process.
  - Therefore, **pager** is used rather than **swapper** in demand paging.
- e.g.
  - Imagine a book. Book is divided into *chapters* (pages). Your desk (*main memory*) can only hold a certain amount of pages. Paging lets you bring relevant chapters to your desk while other pages are stored in the bookshelf (secondary memory).

- We need hardware support to distinguish between the pages that are in memory and that are on the disk.
  - We need a **page table** and a **secondary memory**
  - Same hardware for paging and swapping.
    - Secondary memory is also known as **swap device**
    - It has a section used for swapping purposes: **swap space**
- If you want to reach a page which is not in the memory:
  - **page fault** occurs.
- When that happens, we need to restart any instruction after a page fault.
  - Because we save the state of the interrupted process when the page fault occurs.
  - For that, the desired page must be in memory and accessible.

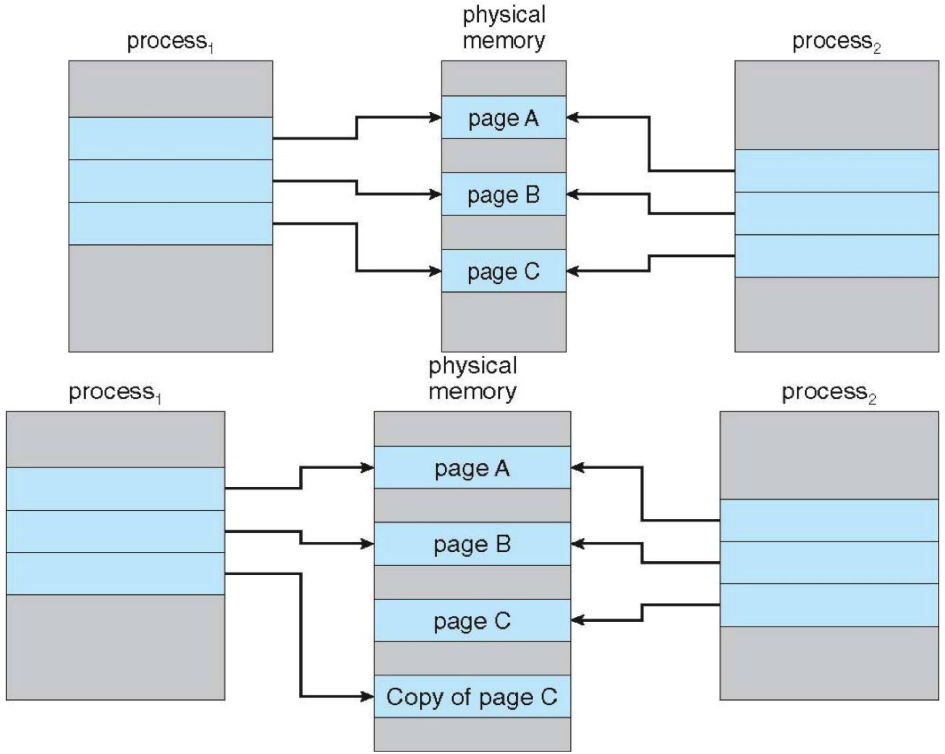
After page fault

- OS receives a signal indicating a page fault occurred. OS identifies the virtual address that caused the fault.
- Checks if the address caused the fault is valid and if the process has the right to access it.
    - If the addr is invalid or access not allowed, OS may terminate the process or handle the error in some other way.
    - If addr is valid, OS locates the data that should be at that address (the one caused page fault). Typically stored on secondary storage.
- OS finds a free frame in RAM or makes room by moving other data out (swapping, page replacement algo)
- Updates the page table to reflect the new location of data in RAM
- Process that caused the page fault is allowed to resume execution

copy-on-write

- Fork() system call creates a child process that is a duplicate of the parent.
  - Works by creating a copy of parent's address space for the child
  - Duplicates the pages belonging to the parent.
- Many child processes invoke exec() after creation.
  - It means that copying of the parent's address space may be unnecessary.
- So, we use a technique known as copy-on-write.
  - Works by allowing the parent and child processes initially to share the same pages.
  - These shared pages are marked as copy on write pages
    - If either process writes to a shared page, a copy of the shared page is created.

- If either process writes to a shared page, a copy is created.
- Assume that child attempted to modify a page.
    - OS will create a copy of this page, mapping it to the address space of the child process.
    - Child will then modify its copied page and not the one belonging to parent.



Page replacement

- When a program needs to access data which is not in main memory, page fault occurs.
- If there is no free space in memory, the OS must decide which existing page to remove or replace to make room for the new page.
- This decision-making process is known as page replacement.

basic page replacement

- If no frame is free, we find one that is not currently being used and free it.
- A frame can be freed by writing the contents to *swap space* and changing the page table to indicate that the page is no longer in memory.
- This free frame can hold the page for which the process faulted.
- So, we are going to change what happens when a page fault happens.

- Find the location of the desired page in the disk (secondary str)
- Find a free frame
  - If there is a free frame, use it.
  - If there is no free frame, use a page-replacement algorithm to select a **victim frame**
  - Write the victim frame to disk, change the page and frame tables accordingly.
- Read the page into the newly freed frame. Change the page and frame tables.
- Continue the user process from where the page fault occurred.
- If no frames are free, we require **two** page transfers (in and out). It doubles the page-fault service time and increases the time.

modify bit

- However, this overhead can be reduced by using a **modify bit** or a **dirty bit**
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
  - Modify bit for page is set by the hardware whenever any byte in the page is written into after being loaded from storage, which indicates that the page is modified.
- So, when we select a page for replacement, we examine the modify bit.
  - If it is set, we know that it has been modified since it was read in from the disk.
  - If so, we must write the page to disk.
- If the modify bit is not set, it means the page is **not** modified since it was read into memory.
  - We need not write the memory page to disk, it is already there.

modify bit

- Indicator of modification
  - It indicates the data in a memory page has been modified since it was read from the disk or main memory.
- Modify bit is crucial for maintaining data integrity and consistency.
- So, when a page with a set modify bit is to be replaced or removed, system known this data has been changed and must be written back to disk or main memory to make sure that data is not lost.
- If the modify bit is not set, system can avoid unnecessary write operation back to storage because the data in memory is the same as the stored version.
  - This optimization reduces write operations (which are slower and bad for SSD)
- In short, it helps OS decide which pages to swap out.
  - A page that was modified is a higher priority for writing back to disk.

modify bit

- Suppose a program reads data from a file into memory.
- Modify bit is initially 0.
- If the program changes the data, system sets the bit to 1.
- Later, if OS needs to free up memory and chooses this page to swap out, it checks the modify bit.
- Since it indicates modification, OS writes the changes back to disk before removing the page from memory.
    - Otherwise it will be lost.

reference string

- To implement **demand paging** we need to solve two problems
  - Frame-allocation algorithm
    - If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  - page replacement algorithm
    - When page replacement is needed, we must select the frames to be replaced.
- These algorithms are important because disk I/O is expensive.
  - Even slight improvements yields large results.
- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.
  - The string of memory references is called a **reference string**
  - This string is a sequence of memory addresses that a process references in the order they occur.
    - Represented as a sequence of page numbers that a process accesses during execution.

reference string

- Theoretical tool used to model and optimize memory management
  - Especially in demand paging systems
- Helps in understanding how different page replacement algorithms will perform in various scenarios.
- In real world, reference string for a process is not known in advanced.
  - Determined by the program’s execution path, input data, etc.

reference string

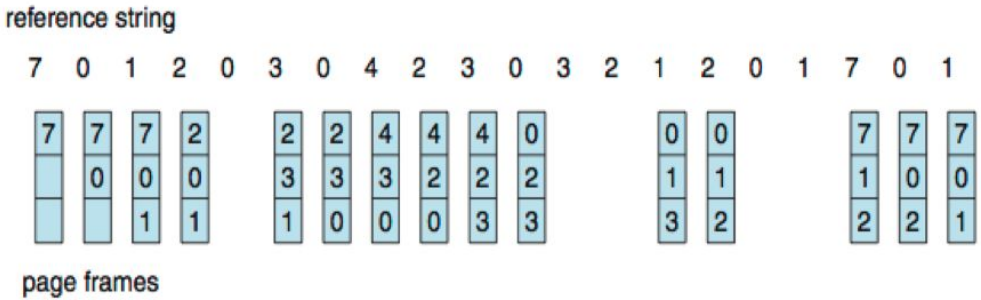
- we can generate reference strings artificially.
  - By using a random number generator
- Or we can trace a given system and record the address of each memory reference.
  - This produces a large number of data.
  - To reduce it, we use 2 facts:
- 1. For a given page size, we need to consider only the page number rather than the entire address.
- 2. If we have a reference to a page **p**, then any references to page p that **immediately** follow will never cause a page fault.
- Page *p* will be in memory after the first reference, so the immediately following references will not fault.

- e.g. if we trace a particular process, we might record the following address sequence.

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 01015

- At 100 bytes per page, this sequence is reduced to :
  - 1,4,1,6,1,6,1,6,1,6,1
- To determine the # of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of available page frames.

- If the number of available frames increase, number of page fault decreases.
  - Adding physical memory increases the number of frames.
- For the future slides where we talk about page replacement algorithms, we will use the reference string:
  - 7,0,1,2,0,3,4,2,3,0,3,2,1,2,0,1,7,0,1
  - for a memory with three frames.



Common Page Replacement Alg.

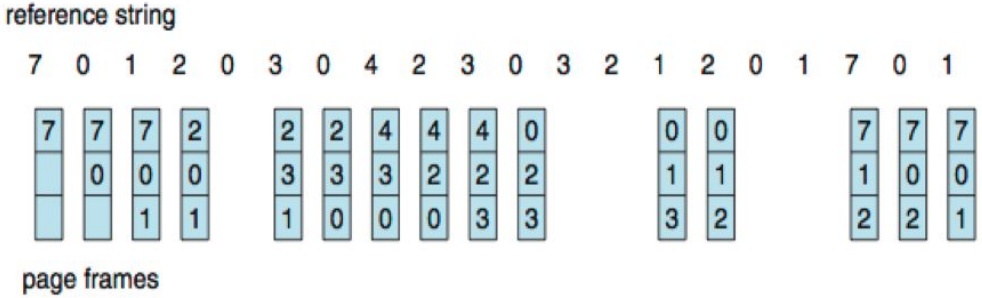
- FIFO
  - Oldest page in memory is the first to be replaced. Simple but can replace pages that are heavily used. (Not smart)
- LRU
  - Replaces the page that has not been used for the longest time.
  - More complex to implement but better.
- Optimal page
  - Theoretical.
  - Replaces the page that will not be used for the longest time in the future.
  - Not implementable in practice because you need to know the future.

FIFO

- When a page must be replaced, the oldest one is chosen.
- A FIFO queue can be used to hold all pages in memory.
- We replace the page at the head of the queue.
- When a new page is brought to memory, we insert it to the tail.

FIFO

- Our three frames are initially empty.
- First three refs (7 0 1) cause page faults and are brought into empty frames.
- Next reference, 2 replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and is already in memory, we have no fault.
- Next we have 3, it replaces 0 since it is now the first in line.



FIFO

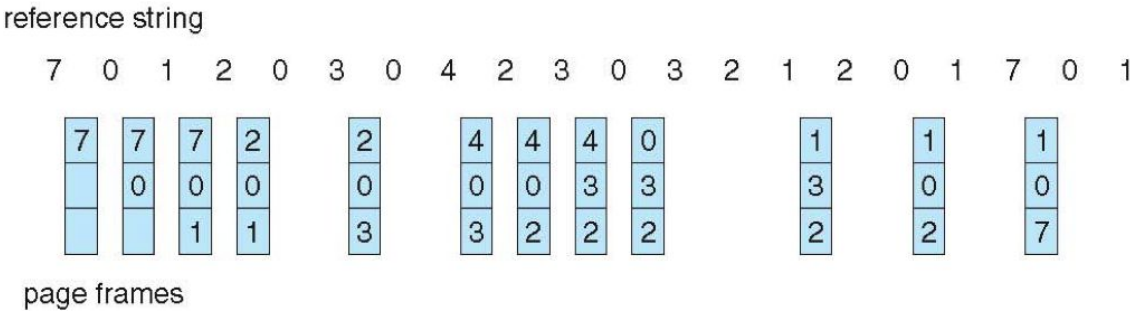
- Easy to understand and program
- Performance is not always good.
- On one hand, the page replace may be an initialization module and maybe we don’t need it.
    - However, on the other hand, it is possible that it contains heavily used variables etc. (important)
- It is also possible that adding more frames can cause more page faults:
    - Belady’s Anomaly.

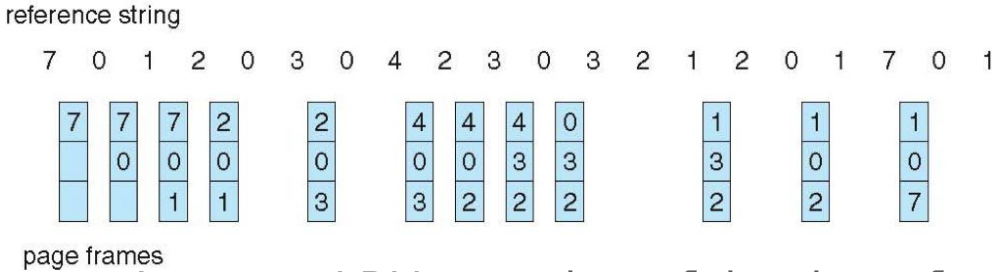
Optimal Page Replacement

- The algorithm that has the lowest page-fault rate of all algorithms and never suffer from Belady's anomaly.
- It exists and called OPT or MIN.
  - Replace the page that will not be used for the longest period of time.
- Use of this algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.
- Difficult to implement because it requires future knowledge of the reference string.
  - It is used mainly for comparison studies.
  - We can grade an algorithm as: *It is within 12.3 percent of optimal at worst*

LRU Page Replacement

- We saw that the implementation of optimal algorithm is not possible
  - Maybe an approximation is possible ?
- If we use the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest period of time.
- Least Recently Used
- Associates with each page the time of that page's last use.
  - When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.





- When the reference to 4 occurs, LRU sees that of the three frames in memory, 2 was used least recently.
  - Thus it replaces 2.
  - However, it doesnt know 2 will come back just later.
  - It faults for 2 next, LRU replaces 3 now because it is now the least recently used of those in memory.
  - So, there are problems but it is still better than FIFO.
- Often used as a page-replacement algorithm and considered to be good.
- The problem is **how to** implement it.
  - Requires hardware assistance.
  - Problem is to determine an order for the frames defined by the time of last use.

Implementation of LRU

- Two implementations are feasible
  - Counters
  - STack
- Counter
  - We associate with each page-table entry as time-of-use field and addd to the CPU a logical clock or counter.
  - Clock is incremented for every memory reference.
  - Whenever a ref to a page is made, contents of the clock register are copied to the time of use field in the page table entry for the page.
  - So, we always have the time of the last reference to each page

- Stack
  - Keep a stack of page numbers
  - Whenever a page is referenced, removed from the stack and put on top
  - So, most recently used page is always at the top of the stack and LRU is always at the bottom.
  - Because entries must be removed from the middle of the stack, best to implement this approach by using a doubly linked list with *head pointer* and *tail pointer*.
- LRU does not suffer from Belady's anomaly
  - Just like optimal

counting based page replacement

- We can keep a counter of the number of references that have been made to each page and develop the following two schemes:
- LFU: Least Frequently Used
  - The page with the smallest count is replaced.
  - Actively used page should have a large reference count.
  - However, its possible that a page is used heavily during the initial phase but never used again.
    - Since that is the case, it will still have a large reference count and it will remain in memory even though it is no longer needed.
- MFU: Most Frequently Used
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
- Neither of these is common.
- Implementation is expensive.

Allocation of frames

- How do we allocate the fixed amount of free memory among various processes?
- If we have 93 free frames and 2 processes, how many frames does each process get?
- The primary goal of frame allocation is to allocate the limited physical memory among various processes efficiently and effectively to optimize performance and avoid issues like *thrashing*.

Allocation strategies

- Fixed allocation
  - Each process is allocated a fixed number of frames.
  - Can lead to under-utilization and over-utilization
- Proportional Allocation
  - Frames are allocated based on the size of the process. Larger processes get more frames.
- Priority Allocation
  - Frames are allocated based on the priority of processes. Higher priority get more.
- Dynamic Allocation
  - # of frames allocated to a process can change dynamically based on its current need and overall system load.

Equal Allocation

- Easiest way to split *m* frames among *n* processes is to give everyone an equal share. *m/n* frames.
- If there are 93 frames and 5 processes, each process will get 18 frames.
- Three leftrover frames can be used as a *free-frame* buffer pool.
- Called **equal allocation**

Proportional Allocation

- Processes will need different amounts of memory.
- Consider a system with 1KB frame size.
- If a small process of 10KB and an interactive database of 127KB are the only two processes running in a system with 62 free frames;
  - It is meaningless to give 31 frames to each.
  - Small process do not need more than 10. So 21 are wasted.
- As a solution, we can allocate available memory to each process according to its size.
- s_i = size of process p_i

  $S = \sum s_i$

  m = total number of frames

  a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Global vs Local allocation

- Page replacement algorithms can be classified into two broad categories.
- **Global replacement**
  - Process select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process
    - One process can take a frame from another process.
  - Greater throughput, so more common
  - But process execution time can vary greatly.
- **Local replacement**
  - A process cannot take a frame from another process.
  - Each process select from only its own set of allocated frames
  - Provides more consistent per-process performance
  - But probably underutilized memory

Global allocation

- The page replacement policy can select *any* frame in physical memory for replacement.
  - It doesn't matter if that frame is owned by another process.
  - It means that a process can lose a frame to another process.
- It is good because it is now flexible and can adapt to changing needs of different processes.
- It can *potentially* lead to more efficient use of memory, as a highly active process can now access more frames compared to a less active one
- However, if not managed carefully, it can lead to **thrashing**
  - Processes will continually lose frames to each other and it will lead to excessive paging.
- Requires sophisticated algorithms to manage the global set of frames effectively.
- If your system have processes with varying memory requirements and total memory usage is dynamically changing, this is ideal.

Local allocation

- Each process has a *fixed* number of frames allocated to it.
- The page replacement policy can only select from these frames in case of a page fault.
- Good, because a process has a guaranteed minimum # of frames.
  - Memory requirement is more predictable.
- Processes are less likely to affect each other's performance because they are not competing for frames.
- However, it is inflexible. It may not use memory efficiently.
  - Because some processes will have frames that they don't need.
  - It may lead to underutilization of memory resources, especially if some processes are idle or require less memory.
- Suitable for systems where process are relatively consistent and have predictable memory needs.

Thrashing

- When we spend more time transferring pages of data in an out of memory (paging) than executing actual instructions, we call it **thrashing**.

Causes of thrashing

- Thrashing often occurs when a system does not have enough physical memory to handle all processes.
- If there are many active processes competing for the limited physical memory, the OS might continuously swap pages in an out of memory to try to keep up with demand.
- If memory pages are poorly allocated
  - Either due to suboptimal page replacement algorithms
  - Or bad allocation strategies
- Running too many applications simultaneously or executing processes that require more memory than the system can provide.

Thrashing

- Adding more RAM can alleviate thrashing by reducing the need for paging.
- Managing the number of running processes.
  - Kill unnecessary ones or run them in different times.
- Using more efficient memory management techniques and algorithms.
-

Summary

- Paging is a memory management scheme.
- With paging, we do not need for contiguous allocation of physical memory.
- It divides virtual memory into blocks of fixed size (pages) and do the same in physical memory (same size) (frames)
- The OS maintains a *page table* for each process.
  - This table maps virtual pages to physical frames.
- Paging allows for the physical memory to be non-contiguous.
  - Therefore makes efficient use of available memory and reduce fragmentation.

Summary

- When a program is executed, it’s divided into pages.
  - Not all pages need to be loaded into physical memory initially.
- System reads from and writes to disk where these pages are stored as needed.
  - But when and how many pages to load is not specified in basic paging.
- So, we talk about **demand paging**: a refined version of paging.

Summary

- Now, pages are loaded into memory only as they are required.
- When a process tries to access a page that is not in memory (it is in the disk), **page fault** occurs.
  - It triggers the OS to load the required page into memory.
  - This is also referred to as **lazy loading**
  - Instead of loading all pages to RAM, pages are loaded on demand.
  - This reduces memory usage.
- It can lead to more efficient use of memory because now only the necessary pages are in RAM.
- Can improve system performance and responsiveness by reducing unnecessary I /O and memory usage.

Summary

- Important aspect is handling page faults.
- System must decide which pages to keep in memory and which to swap out
  - Here, we talk about page replacement algorithms
  - LRU, FIFO, etc.
- In simple paging, the strategy for loading pages into memory is not specified.
- The entire process or large parts can be loaded at the start.
- In demand padding, pages are loaded only as they needed.