Operating Systems — Memory Management

ECE_3TC31_TP/INF107

Stefano Zacchiroli

2024

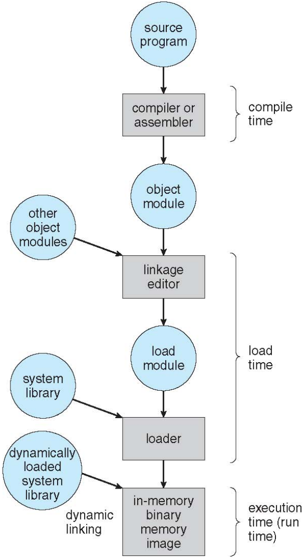
Background

Background

- Program must be brought (from disk) into memory for execution
- The only types of storage CPU can access directly are: (1) registers, (2) main memory
- Memory hardware is “dumb”, it only sees a stream of:
  - address + read request, or
  - address + data and write requests
- Performances:
  - Register access is done in one CPU clock cycle (or less)
  - Main memory can take many cycles, causing a **stall** (i.e., process blocked waiting for memory)
  - **Cache** sits between main memory and CPU registers to avoid/mitigate stalls
- **Protection** of memory is required to ensure correct OS operation

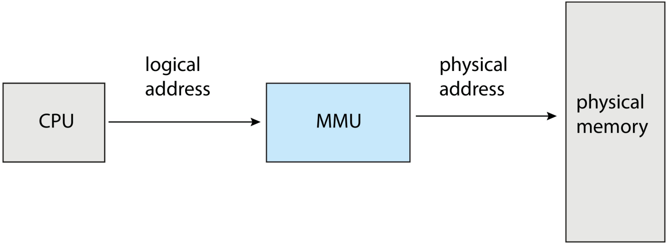
Address Binding

- Addresses are represented differently throughout program's life cycle
  - Source code addresses usually **symbolic**
  - Compiled code addresses bind to **relocatable addresses**
    - E.g., "14 bytes from beginning of this module"
  - Linker or loader bind relocatable addresses to **absolute addresses**
    - E.g., `0x74014`
  - Each binding maps one address space to another
- Address binding of instructions and data to memory addresses can happen at different stages
  - **Compile time:** If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
  - **Load time:** Must generate *relocatable code* if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one segment to another
    - Need hardware support for address maps (e.g., relocation register)



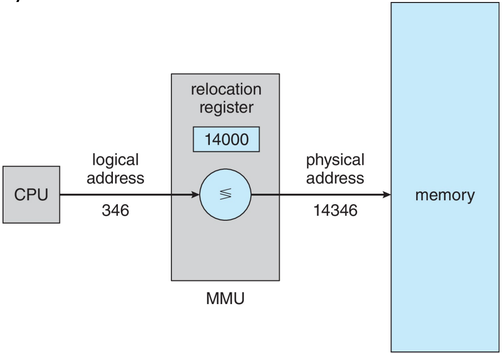
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address** – generated by the CPU (also referred to as **virtual address**)
  - **Physical address** – address seen by the memory unit
- Logical *address space* is the set of all logical addresses generated by a program
- Physical address space is the set of all physical addresses generated by a program
- **Memory-Management Unit (MMU)**: *hardware* device that at run time maps logical to physical address



Memory-Management Unit (MMU)

- At hardware level, *CPU must check every memory access* generated in user mode for validity
  - Two registers used for this: relocation & limit
- Each *logical address* is verified to be between 0 and a maximum (logical) address stored in the **limit register** (not shown in the picture)
- The value in the **relocation register** is *added to every address* generated by a user process at the time it is sent to memory



Contiguous Memory Allocation

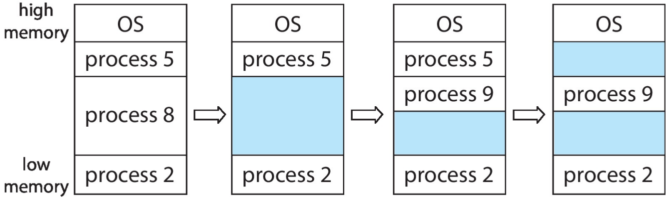
Memory Allocation Problem

- When loading a program for execution[1] we need to decide **where to put it** in physical memory
  - More precisely: where (= which physical address) to map each of its logical addresses to
- This is the **memory allocation problem**
- The simplest memory-allocation schemes are based on the idea of **contiguous memory allocation**
  - We only decide the base *starting physical address* for a given program
  - Subsequent addresses (both logical and physical) will follow increasingly from there, "contiguously"

[1] *at least* when loading; we will see later that there are other situations in which we will need to re-decide this

Variable Partition Allocation

Variable partition allocation is a contiguous memory allocation scheme where each program is loaded into a memory partition of the same size of the program



- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- OS maintains information about: (a) allocated partitions, and (b) free partitions (hole)
- When a process starts: it is allocated memory from a *hole large enough* to accommodate it
- When a process terminates: OS frees its partition, adjacent free partitions are combined

Dynamic Storage-Allocation Problem

How to satisfy an allocation request for a partition of size n from a list of free holes?

- **First-fit:** Allocate the *first hole* that is big enough
- **Best-fit:** Allocate the *smallest hole* that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest hole*
  - Must search entire list, as before
  - Produces the largest leftover hole

Dynamic Storage-Allocation Problem

How to satisfy an allocation request for a partition of size n from a list of free holes?

- **First-fit:** Allocate the *first hole* that is big enough
- **Best-fit:** Allocate the *smallest hole* that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest hole*
  - Must search entire list, as before
  - Produces the largest leftover hole

Experimental evaluation results

- First-fit and best-fit better than worst-fit in terms of speed and memory utilization
- No clear winner between first-fit and best-fit in terms of memory utilization
- But **first-fit** faster than best-fit (hence overall winner)

Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Intrinsic problem to any allocation scheme with granularity larger than 1 address (= 1 byte)
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Intrinsic problem to contiguous allocation
- First-fit analysis reveals that given N blocks allocated, $0.5 \cdot N$ blocks are lost due to external fragmentation
  - $\frac{1}{3}$ may be unusable $\rightarrow$ **50% rule** (proven by Knuth)
- We can *mitigate* external fragmentation with **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
- Issues:
  - Could require copying memory from/to mass storage (slow) if memory is really tight
  - Takes a lot of time! And involved processes are blocked in the meantime

Fragmentation

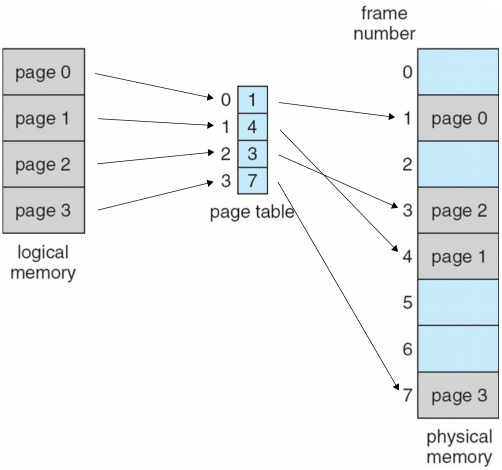
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Intrinsic problem to any allocation scheme with granularity larger than 1 address (= 1 byte)
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Intrinsic problem to contiguous allocation
- First-fit analysis reveals that given N blocks allocated, $0.5 \cdot N$ blocks are lost due to external fragmentation
  - $\frac{1}{3}$ may be unusable $\rightarrow$ **50% rule** (proven by Knuth)
- We can *mitigate* external fragmentation with **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
- Issues:
  - Could require copying memory from/to mass storage (slow) if memory is really tight
  - Takes a lot of time! And involved processes are blocked in the meantime

Can we do better?

Paging

Paging

- Main idea: make the **physical address space non-contiguous** (logical space still contiguous)
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-size blocks called **frames**
  - Size is power of 2, usually between 512 bytes and 16 MiB, dictated by hardware
- Divide logical memory into blocks of the same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses



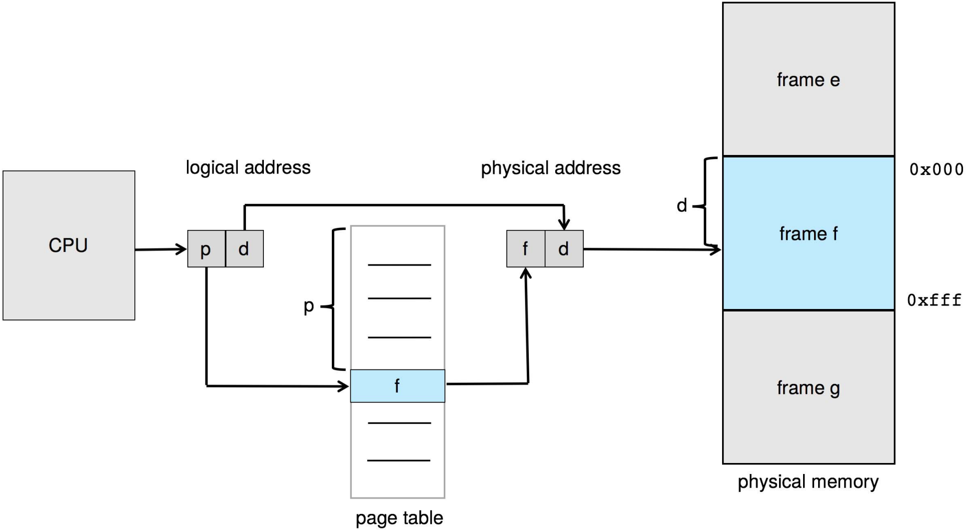
Address Translation Scheme

- Physical address generated by CPU is divided into:
  - **Page number** (p) – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset** (d, in bytes) – combined with base address to define the physical memory address that is sent to the memory unit
- Same split (number+offset) applied to logical addresses

page number	page offset
p	d
m -n	n

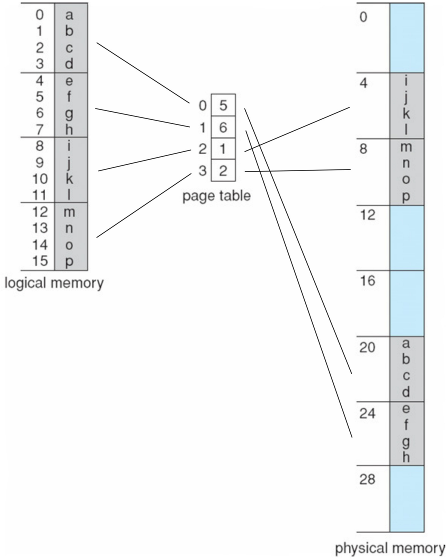
For given logical address space 2^m and page size 2^n

Paging Hardware



Paging (Example)

- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)
- (Logical) page 0 starts at logical address 0 and is associated by the page table to (physical) frame 5, starting at physical address 20 (decimal).
- Logical address 2—corresponding to page 0 and offset 2—contains byte c and is stored in memory at physical address 22 (decimal).



Internal Fragmentation (Example)

- Paging addresses external fragmentation, but not internal fragmentation
- Example:
  - Page size = 2048 bytes
  - One process of size = 72766 bytes
  - Requires: 35 pages (x 2048 = 71680 bytes) + 1086 bytes
    - 1 full page (the 36th) allocated to cover what remains
  - Internal fragmentation: 2048 - 1086 = 962 bytes (1.32% of process size)
- Worst case fragmentation = 1 full frame allocated for just 1 byte in a very small process
- Average fragmentation = ½ frame size

- So are *small frame sizes* desirable?

Internal Fragmentation (Example)

- Paging addresses external fragmentation, but not internal fragmentation
- Example:
  - Page size = 2048 bytes
  - One process of size = 72766 bytes
  - Requires: 35 pages (x 2048 = 71680 bytes) + 1086 bytes
    - 1 full page (the 36th) allocated to cover what remains
  - Internal fragmentation: 2048 - 1086 = 962 bytes (1.32% of process size)
- Worst case fragmentation = 1 full frame allocated for just 1 byte in a very small process
- Average fragmentation = ½ frame size

- So are *small frame sizes* desirable?
- But the page table takes memory too!
- Historical trend in OSes: *increase* page sizes over time (to reduce page table sizes). E.g.:
  - Solaris supports two page sizes – 8 KiB and 4 MiB
  - Linux supports standard pages of 4 KiB and **"huge pages"** up to 1 GiB

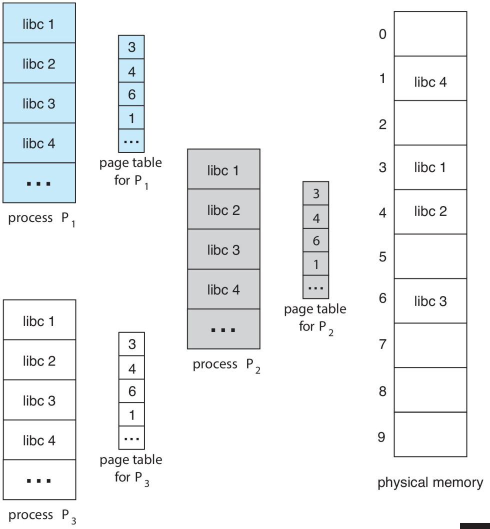
Shared Pages

- Paging adds an *indirection level* between logical memory (seen by the processes) and physical memory (seen by the hardware)
- This indirection can be exploited to **share memory pages** between processes

- Use case: **shared code**
  - One copy of read-only code shared among processes (i.e., text editors, compilers, libraries)
  - Similar to multiple threads sharing the same program instructions
- Use case: **shared data**
  - Also useful for interprocess communication if sharing of read-write pages is allowed
  - Similar to multiple threads sharing the same address space

Shared Pages (Example)

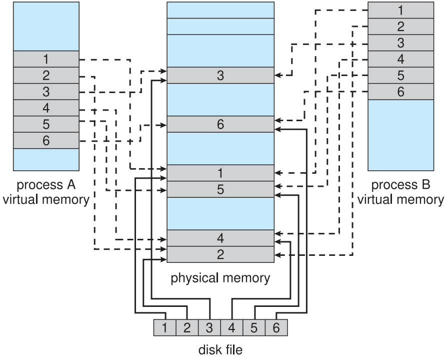
- **Shared libraries** are generally shared among all processes linked against (the same versions of) them
- The C standard library `libc` is often shared by most of the processes executing on a system
  - Significant memory saving!
- But the principle is applicable to any other shared library and object



Memory Mapping

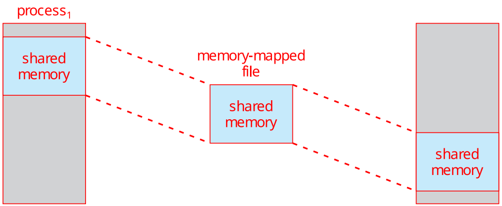
Memory-Mapped Files

- **Memory mapping** is a direct file access method, alternative to sequential access, which leverages the page table to improve file I/O performances
- Idea: "map" a (part of a) file to a set of memory pages—e.g., a fixed-length `void *` buffer
  - Read from memory buffer → data read from file to memory via *demand paging* (more on this later)
  - Write to memory buffer → write data to file
- After setup, **no syscalls needed** for I/O operations
  - Lower context-switch overhead
  - No seek/file pointer needed! All accesses are direct, with byte granularity
  - I/O still takes time to happen, of course, but can be lazy and is cached transparently by the OS



Sharing Memory via Memory Mapping

- Memory mapping can also be leveraged as a **shared memory** IPC mechanism
- If multiple processes memory map the same file, the OS will make them share mapped pages
  - Reading/writing memory will result in *both* updating the mapped file and sharing the updated data with all participant processes
- If only memory sharing is desired (and not file access), some OSes allow to map regions of **anonymous memory**, which are not backed by a file and can be shared by *related* processes, e.g., across `fork()`



Memory Mapping — UNIX Example

- UNIX provides the `mmap()` syscall to setup memory mappings

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
    // Returns starting address of mapping on success, or MAP_FAILED on error
    //
    // prot -> memory protection, e.g., whether it is r/w or r/o
    // flags -> MAP_SHARED for memory that (1) will be reflected to the FS and
    //          (2) (potentially) shared among processes; MAP_PRIVATE otherwise
```

  - It maps a region of an open file (which can be the entire file) to a memory buffer whose address is returned by the syscall
  - Note that via mmap access you can change the content of a file but **not resize** it; write/lseek/truncate are needed for that

- The matching `munmap()` syscall shuts down an existing memory mapping

```
int munmap(void *addr, size_t length);
    // Returns 0 on success, or -1 on error
```

mmap — Example

```c
#include <sys/mman.h>
/* ... include list trimmed for space ... */

#define errExit(msg) { fprintf(stderr, "%s\n", (msg)); exit(EXIT_FAILURE); }

int main(int argc, char *argv[]) {
    char *addr;
    int fd;
    struct stat finfo;

    if (argc != 2)  errExit("Usage: mmap FILE");
    fd = open(argv[1], O_RDONLY);  /* open input file */
    if (fstat(fd, &finfo) == -1)   /* retrieve file info, as we need its size */
        errExit("fstat failed");

    addr = mmap(NULL, finfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED)  errExit("mmap failed");

    if (write(STDOUT_FILENO, addr, finfo.st_size) != finfo.st_size)
        errExit("incomplete file read/write");
    exit(EXIT_SUCCESS);
}
```

mmap — Example (cont.)

```
$ gcc -Wall -g -o mmap mmap.c
$ ./mmap /etc/passwd | head -n 5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

- Exercise: try it also under `strace` to check if `read`/`write` syscalls are happening and why.

Swapping

Swapping

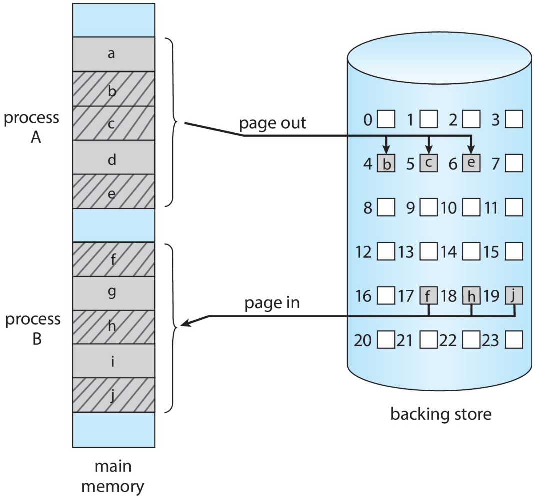
- When memory becomes tight, (parts of) processes can be **swapped temporarily out of memory** to a backing store, and brought back into memory later for continued execution
- **Total logical memory** space of processes can **exceed physical memory**
  - The degree of multiprogramming increases
- **Backing store** – fast disk large enough to accommodate copies of all address spaces of all processes
- Major part of swap time is I/O transfer time; total transfer time is directly proportional to the amount of memory swapped

Swapping

- When memory becomes tight, (parts of) processes can be **swapped temporarily out of memory** to a backing store, and brought back into memory later for continued execution
- **Total logical memory** space of processes can **exceed physical memory**
  - The degree of multiprogramming increases
- **Backing store** – fast disk large enough to accommodate copies of all address spaces of all processes
- Major part of swap time is I/O transfer time; total transfer time is directly proportional to the amount of memory swapped
- Does the swapped out process need to swap back in to *same physical addresses*?
  - Generally not, but it is complicated™ if swapping during pending I/O requests
- Modified versions of swapping are found on most non-mobile OS (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Swapping Pages

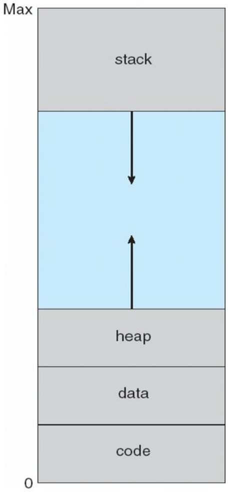
- With memory paging, we can implement swapping at **page granularity**
  - Obsolete alternative: swap in/out *entire processes*
- When memory is tight, swap out individual pages
- When memory becomes available again, or swapped out pages are needed, swap them back in



Virtual Memory

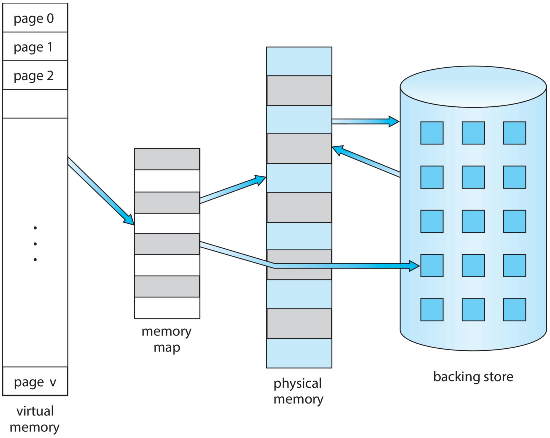
Virtual Address Space

- The addresses and address space seen by programs are "virtual", in the following sense
- Usually design logical address space for *stack* to start at **max logical address** and grow "down" while *heap* grows "up" from 0
  - Maximizes address space use
  - (Lot of) unused address space between the two is a **hole**
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse address spaces** with holes left for growth, dynamically linked libraries, etc.



Virtual Memory

The union of virtual addresses and paging provides the illusion that the memory available to any process is very large, generally much larger than physical memory → **virtual memory**.



Demand Paging

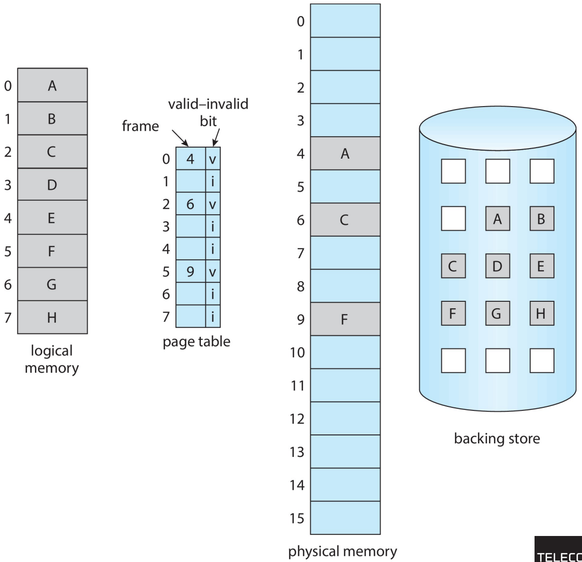
- When executing a program we can bring entire process into memory at load time
- Or, with virtual memory, we can *bring a page into memory only when it is needed* → **demand paging**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster startup time

Demand Paging

- When executing a program we can bring entire process into memory at load time
- Or, with virtual memory, we can *bring a page into memory only when it is needed* → **demand paging**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster startup time
- To implement demand paging we need **MMU support** for determining:
  - If pages needed are already *memory resident* → no difference from non demand-paging
  - If page needed and not memory resident → need to detect and load the page into memory from backing storage
    - Without affecting program runtime behavior (*user transparent*)
    - Without programmer needing to change code (*developer transparent*)

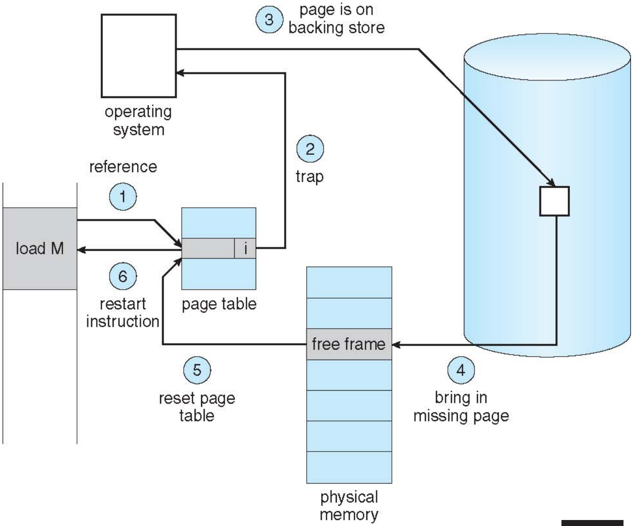
Valid-Invalid Bit

- Associate to each entry in the page table a **valid–invalid bit**
  - v → page in memory
  - i → page not in memory
- Initially set to i for all entries (*demand* paging!)
- During MMU address translation, if an invalid page is requested → **page fault**



Handling a Page Fault

1. Access invalid page → page fault
2. Trap to the OS
3. Find free frame
4. Swap page into frame (disk I/O)
5. Set valid bit to v
6. Restart the instruction that caused the page fault



Performances of Demand Paging

- Three major activities
  - Handle the interrupt – just several hundred instructions with careful coding
  - Read the page – lot of time (I/O)
  - Restart the process – again just a small amount of time
- **Page fault rate** $0 \leq p \leq 1$
  - $p = 0 \rightarrow$ no page faults; $p = 1$ every memory reference is a fault
- **Effective Access Time (EAT)** =

  $(1-p) \times$ memory access + $p \times$ (page fault overhead + swap page out + swap page in)

Performances of Demand Paging (Example)

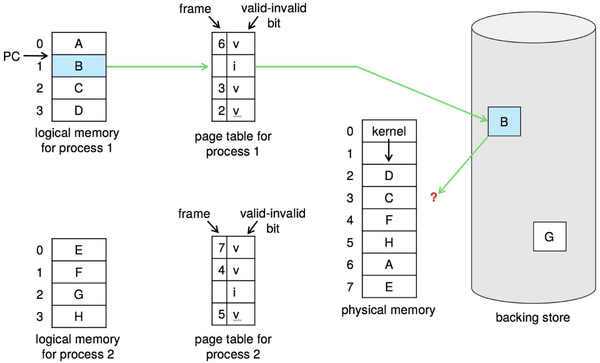
- Memory access time = 200 ns
- Average page-fault service time = 8 ms
- EAT = $(1-p) \times 200 + p \times (8\,ms)$
  = $(1-p) \times 200 + p \times 8\,000\,000$
  = $200 + p \times 7\,999\,800$
- If one access out of 1000 causes a page fault, then EAT = 8.2 microseconds (µs).
  - This is a slowdown by a factor of 40 !
- If we want performance degradation < 10%
  - $220 > 200 + 7\,999\,800 \times p$
  - $20 > 7\,999\,800 \times p$
  - $p < .0000025$, i.e., less than one page fault in every $400\,000$ memory accesses

(Spoiler: yes, it is achievable, because program execution exhibits strong **locality of reference**.)

Page Replacement

What if There are no Free Frames?

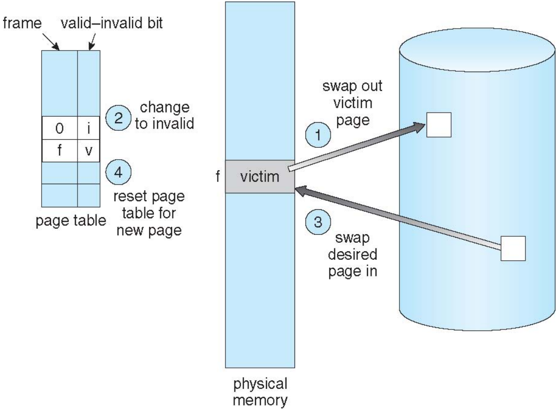
- Virtual memory is nice, but pages are ultimately stored in physical frames
- Memory frames used by: process pages, kernel pages, I/O buffers
- Q: **What if there are no free frame** when a page fault happens?
  - We have **over-allocated memory**, i.e., allocated more virtual memory than available physical memory
  - It is fine!, because it increases multiprogramming, but we need to handle it



- A: **Page replacement** – find some page in memory, but not really in use, page it out
  - A **page replacement algorithm** decides what to do, both with concerned processes (terminate? swap out?) and concerned pages (which ones to page in/out)

Page Replacement Process

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If not, page replacement algorithm **selects a victim frame**
   - Write victim frame to disk *if dirty*
     - If the page was read-only or unmodified, e.g., code, there is no need to write it back to disk
3. Bring the desired page into the (newly) free frame; update page table
4. Continue the process by restarting the instruction that caused the trap



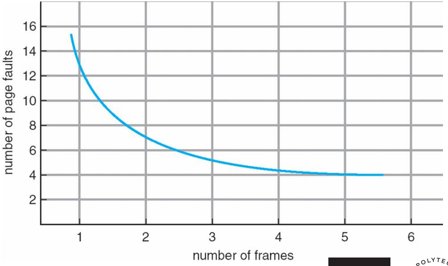
Note: requires up to 2 page transfers per page fault → increasing EAT

Evaluation of Page Replacement Algorithms

- Intertwined sub-problems: **frame allocation** policy (with multiprogramming, how many frames allocate to each process); **page replacement** policy (when memory is full at a page fault, which page to replace)
  - We will focus on evaluating page replacement algorithms
- Goal: **minimize the number of page faults**
  - Both on first and subsequent accesses to a page

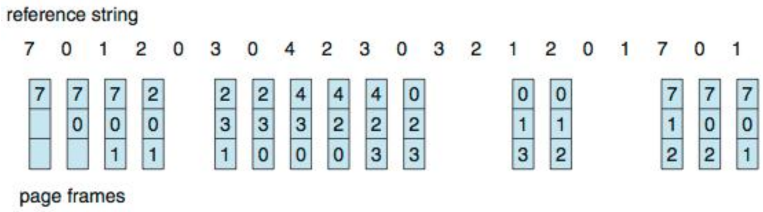
- Evaluate algorithm by running it on a string of memory references (**reference string**) and computing the number of page faults on that string
  - String is just page numbers, not full addresses, e.g., `7,0,1,2,0,3,0,4,2,3,0,…`
  - String can be random, simulated from a model, or a trace recorded from a real system
  - Evaluation results depend on number of frames available. In general, we expect *page faults to decrease when available frames increase* (i.e., adding memory should not make your system *slower*)

Expected trend for good memory replacement policies



First In, First Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames per process, i.e., 3 pages can be in memory at a time per process

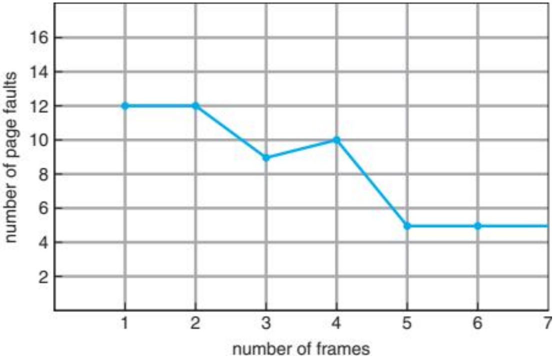


- Total: 15 page faults

- Implementation: How to track ages of pages?
  - Don't. Just use a FIFO queue instead.

Bélády's Anomaly

- Consider a FIFO page replacement algorithm
- Reference string
  `1,2,3,4,1,2,5,1,2,3,4,5`
- The number of page faults for varying amounts of available frames is shown on the right

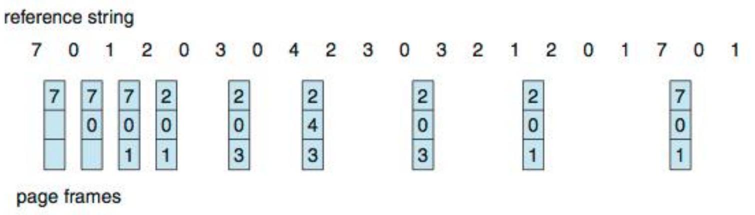


FIFO page replacement exhibits **Bélády's Anomaly**:[2] the page-fault rate may *increase* as the number of available frames increases.

[2] Belady, Nelson, Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. Commun. ACM 12(6): 349-353 (1969)

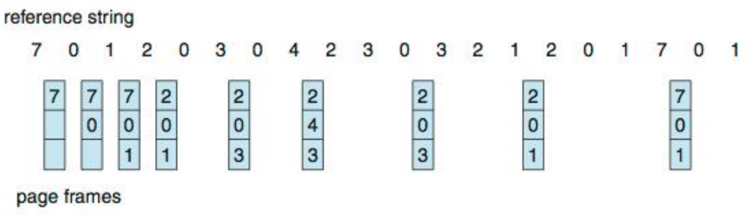
Optimal Page Replacement

- Requirements for the best possible page replacement algorithm:
  1. Has the lowest page-fault rate among all possible algorithms
  2. Does not exhibit Bélády's anomaly when increasing available frames
- Such an algorithm exists and has been called **OPT page replacement** algorithm (also called *MIN*)
  - OPT rule: **Replace the page that will not be used for the longest period of time.**
  - Example (9 page faults in total):



Optimal Page Replacement

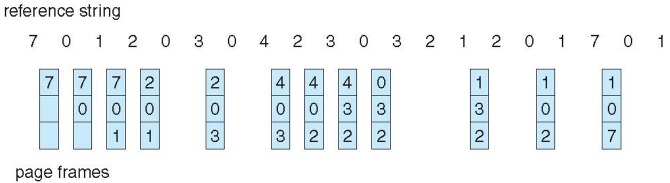
- Requirements for the best possible page replacement algorithm:
  1. Has the lowest page-fault rate among all possible algorithms
  2. Does not exhibit Bélády's anomaly when increasing available frames
- Such an algorithm exists and has been called **OPT page replacement** algorithm (also called *MIN*)
  - OPT rule: **Replace the page that will not be used for the longest period of time.**
  - Example (9 page faults in total):



- There's one "little" problem: we cannot predict the future, so how can we implement OPT?
  - A: we can't, but we can **approximate it**!
  - Also, OPT is a useful *reference benchmark*.

Least Recently Used (LRU) Algorithm

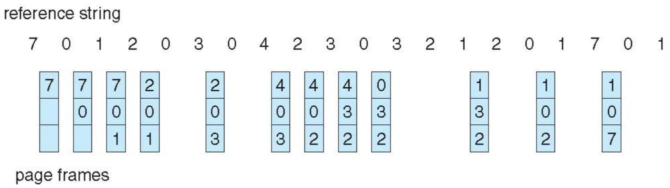
- Idea: use past knowledge, rather than the future, to approximate OPT.
  - Assumption: history repeats itself.
- **Replace the page that has not been used for the longest amount of time**



- 12 faults – better than FIFO but worse than OPT
- LRU is a generally good algorithm and frequently used. Does not exhibit Bélády's anomaly.

Least Recently Used (LRU) Algorithm

- Idea: use past knowledge, rather than the future, to approximate OPT.
  - Assumption: history repeats itself.
- **Replace the page that has not been used for the longest amount of time**



- 12 faults – better than FIFO but worse than OPT
- LRU is a generally good algorithm and frequently used. Does not exhibit Bélády's anomaly.
- But how to implement?
  1. Page **counters** storing last use timestamps → search through all counters to find victim
  2. Keep a **stack** of page numbers in a doubly-linked list
     - A page is referenced → move it to the top (changing 6 pointers max)
     - No search needed for replacement, but each update is more expensive

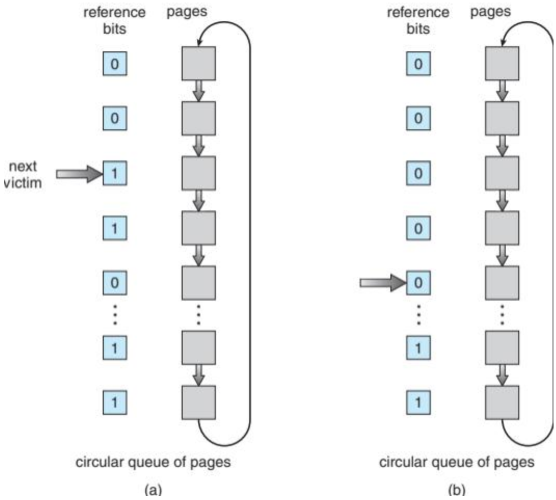
LRU Approximation Algorithms

- LRU needs special support (counters or stack) and is still slow (for updates or victim selection)
- Modern systems provide **hardware support** that can be leveraged by page replacement
- **Reference bit**
  - Associated to each page, initially set to 0 (by OS)
  - When a page is referenced → set bit to 1 (done automatically by hardware)
  - After some time we check all the bits
    - All pages with reference bit = 0 have not been referenced (since last check)
    - We select our victims among these
    - We do not know the order of reference *among them*, though
- Reference bits provide support to implement efficiently algorithms that approximate LRU (which in turn approximates OPT)

Second-Chance Algorithm

Second-Chance Algorithm (also called *clock algorithm*): a widely used LRU approximation based on reference bits

- Basic policy: FIFO replacement
- When a candidate victim is selected we inspect its reference bit
  - If bit = 0 → page not referenced, victim found
  - If bit = 1 → page was referenced
    - "Give page a 2nd chance" and move to next candidate victim
    - Set reference bit to 0
- Eventually we will find a victim with bit = 0



Enhanced Second-Chance Algorithm

- Idea: Improve algorithm by using **reference bit and modify bit** (if available) in concert
- Take **ordered pair** $\langle$reference bit, modify/“dirty” bit$\rangle$
  - (0, 0) neither recently used not modified → best page to replace (no need to swap it out!)
  - (0, 1) not recently used but modified → not quite as good, must write out before replacement
  - (1, 0) recently used but clean → probably will be used again soon
  - (1, 1) recently used and modified → probably will be used again soon and need to write out before replacement
- When page replacement called for, use the second chance scheme but use the four classes and replace page in lowest non-empty class
- Might need to search circular queue several times

Reading List

You should study on books, not slides! Reading material for this lecture is:

- Silberschatz, Galvin, Gagne. Operating System Concepts, Tenth Edition:
  - Chapter 9: Main Memory
  - Chapter 10: Virtual Memory

Credits:

- Some of the material in these slides is reused (with modifications) from the official slides of the book Operating System Concepts, Tenth Edition, as permitted by their copyright note.

Appendix

More on the Page Table

Implementation of the Page Table

- Remember: page table is process-specific, needs to be updated upon context switch
- 1st approach: **one register per entry**(!) in the page table
  - Very fast
  - Not scalable unless the page table is very small (e.g., ≤ 256 entries)

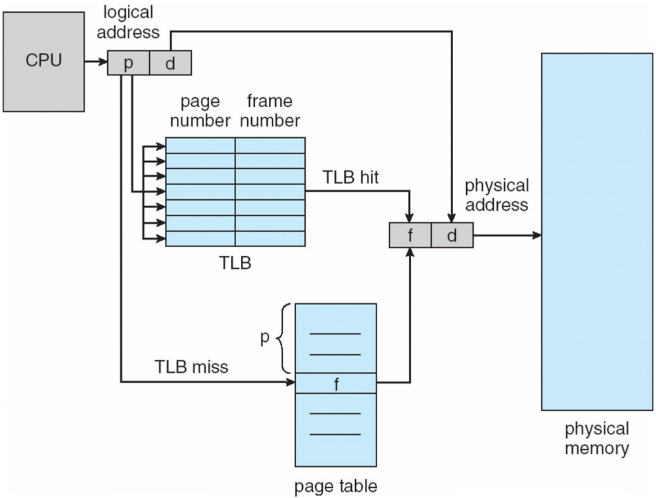
Implementation of the Page Table

- Remember: page table is process-specific, needs to be updated upon context switch
- 1st approach: **one register per entry**(!) in the page table
  - Very fast
  - Not scalable unless the page table is very small (e.g., ≤ 256 entries)
- 2nd approach: page table is kept in **main memory**
  - Registers just *point to it* and are updated upon context switch
  - Usually two: *page-table base register* (PTBR, start address) + *Page-table length register* (PTLR, size)
- Problem: every memory data/instruction access now requires **two memory accesses**
  - One for the page table and one for the data / instruction

Implementation of the Page Table

- Remember: page table is process-specific, needs to be updated upon context switch
- 1st approach: **one register per entry**(!) in the page table
  - Very fast
  - Not scalable unless the page table is very small (e.g., ≤ 256 entries)
- 2nd approach: page table is kept in **main memory**
  - Registers just *point to it* and are updated upon context switch
  - Usually two: *page-table base register* (PTBR, start address) + *Page-table length register* (PTLR, size)
- Problem: every memory data/instruction access now requires **two memory accesses**
  - One for the page table and one for the data / instruction
- 3rd approach: use special fast hardware cache called **translation look-aside buffers (TLBs)**
  - Fast associative memory that can store page table entries; small (64 to 1024 entries)
  - It's a cache with hits and misses; *TLB miss* → fallback to page table in main memory
  - TLB key:
    - just the page number → TLB flushed at each context switch
    - page number + *address-space identifier* (*ASID*) → TLB can store page table entries for multiple processes

Translation Look-aside Buffer (TLB)



More on Swapping

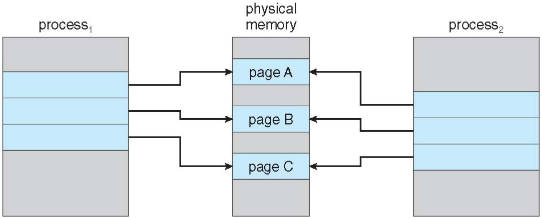
Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS **asks apps** to voluntarily **relinquish allocated memory**
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination by the OS
  - Android **terminates apps** if low free memory, but first **writes application state** to flash for fast restart

More on Virtual Memory

Copy-on-Write

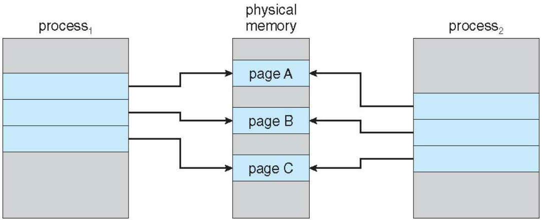
- **Copy-on-Write (COW)** allows parent and child processes to initially share memory pages
- If either process modifies a shared page, only then is the page copied
- COW allows more *efficient process creation* as only modified pages are copied (later)
  - UNIX OSes uses this to implement `fork()` efficiently



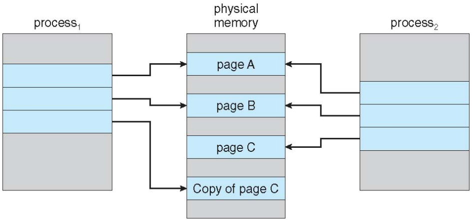
Before Process 1 modifies page C

Copy-on-Write

- **Copy-on-Write (COW)** allows parent and child processes to initially share memory pages
- If either process modifies a shared page, only then is the page copied
- COW allows more *efficient process creation* as only modified pages are copied (later)
  - UNIX OSes uses this to implement `fork()` efficiently



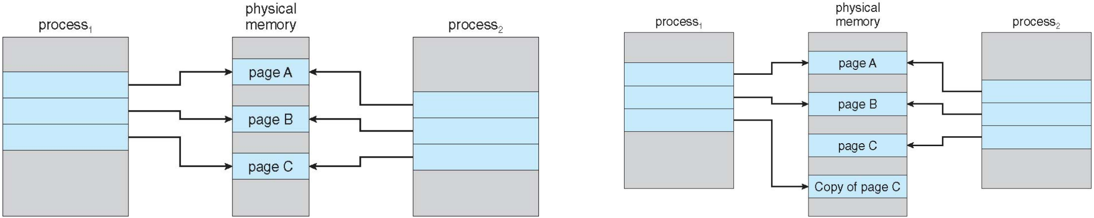
Before Process 1 modifies page C



After Process 1 modifies page C

Copy-on-Write

- **Copy-on-Write (COW)** allows parent and child processes to initially share memory pages
- If either process modifies a shared page, only then is the page copied
- COW allows more *efficient process creation* as only modified pages are copied (later)
  - UNIX OSes uses this to implement `fork()` efficiently



Before Process 1 modifies page C — After Process 1 modifies page C

- Note that we still need to duplicate the page *table* upon `fork()`
  - Which is a waste of time is the child will `exec()` just after
  - `vfork()` is a variant of `fork()` that: (1) does not duplicate the page table, (2) blocks parent process until child exits or `exec()`-s