Operating Systems

Memory

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

May, 2022

Why do we need memory?

- Non-volatile memory: Secondary
  – HDD, SDD, Flash Drive, etc.
- Volatile memory: Primary
  – RAM, Registers, Cache, etc.

Non-volatile memory	Source code (prog1.c)		
	↓	Compile	Address binding (Phase I)
	Object code (prog1.o)		
	↓	Link	Address binding (Phase II)
	Executable file (a.out)		
	↓	Load	
Volatile memory	Executable file (a.out)		
		Execute	Address binding (Phase III)

Why do we need to manage memory?

The processes, together with the data they access, must be (at least partially) in main memory during execution.

For a better performance, we must we must share memory (i.e., keep many processes in memory). Memory management schemes control the way a shared memory is managed alongside the other tasks.

Selection of a memory-management scheme for a system depends on the system's hardware design.

The association between CPU and memory

The CPU fetches instructions from memory according to the value of the program counter.

Machine instructions can take memory addresses as arguments but not disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.

Note: If the data is not in memory, they must be moved there before the CPU can operate on them.

Looking at the hardware

Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly.

Built-in registers are generally accessible within one cycle of the CPU clock. But completing a memory access may take many cycles of the CPU clock. This is because the main memory is accessed via a transaction on the memory bus.

While accessing main memory, the processor normally needs to *stall*, since it does not have the data required to complete the instruction that it is executing. To manage this efficiently, we can add a relatively faster type of memory between the CPU and main memory, typically on the CPU chip for quick access. Such memories are known as *cache*.

Memory address space

We need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

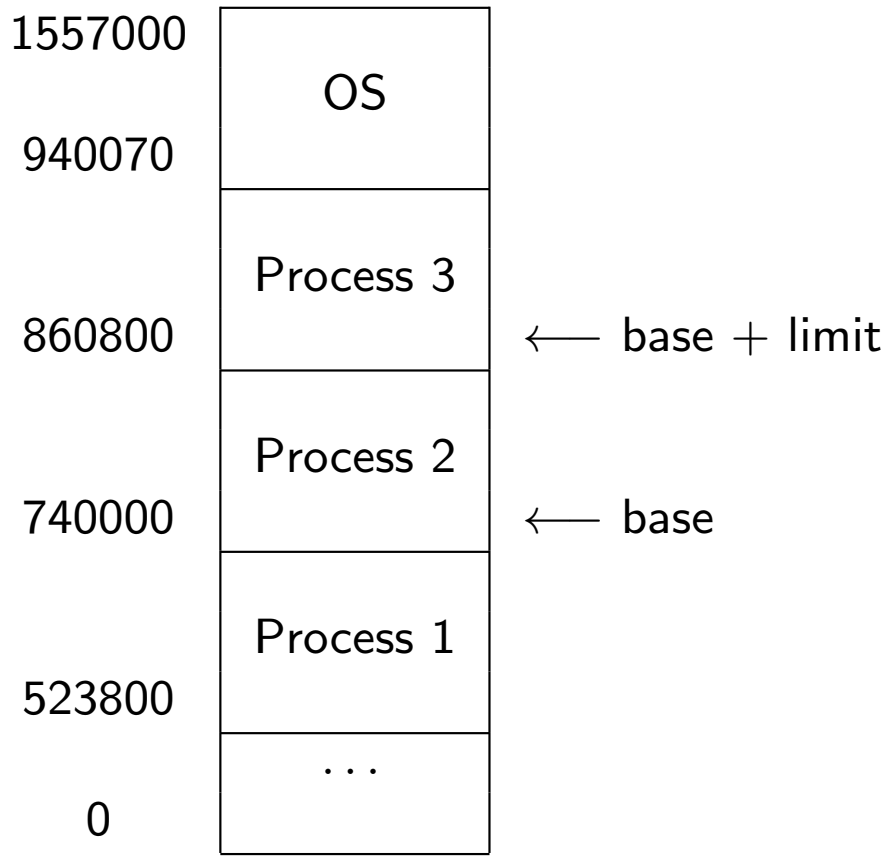
We can provide this protection by using the following two registers.

- The *base register* holds the smallest legal physical memory address.
- The *limit register* specifies the size of the range.

For example, if the base register holds 100780 and the limit register is 140200, then the program can legally access all addresses from 100780 through 240199 (inclusive).

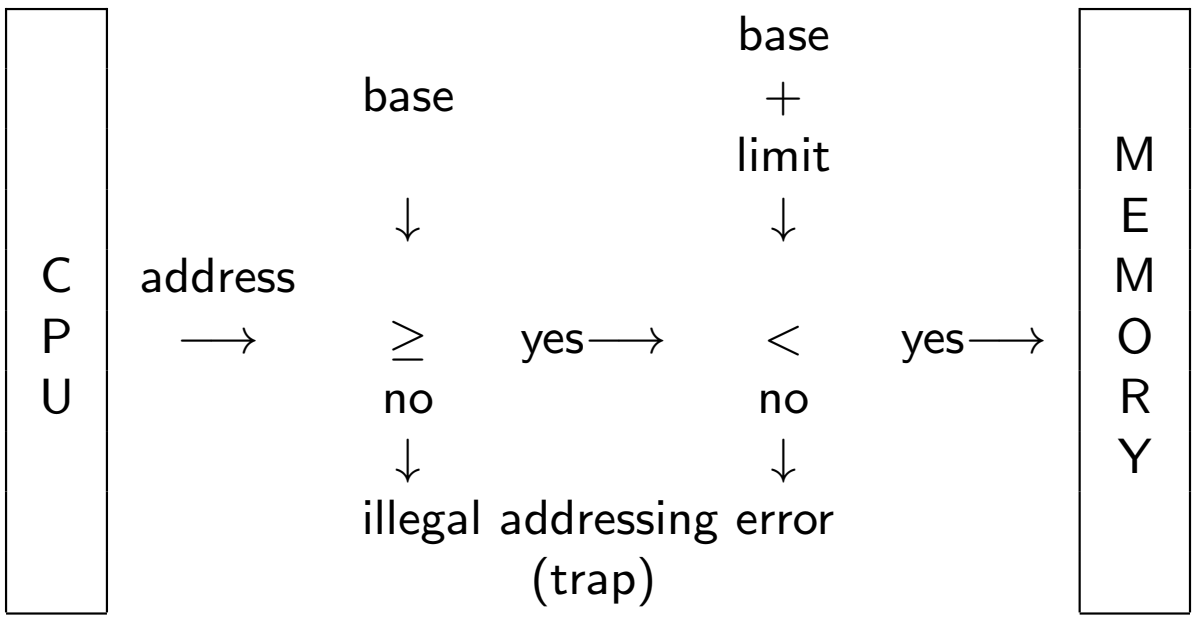
Memory address space

Memory consists of a large array of bytes, each with its own address. A base and a limit register define a logical address space.



How can we protect the memory space?

Protection of memory space can be achieved by having the CPU hardware compare every address (generated in user mode) with the registers. Any attempt by a program executing in user mode to access the OS memory or other users' memory results in an error (known as trap).



Address binding

High-level code:

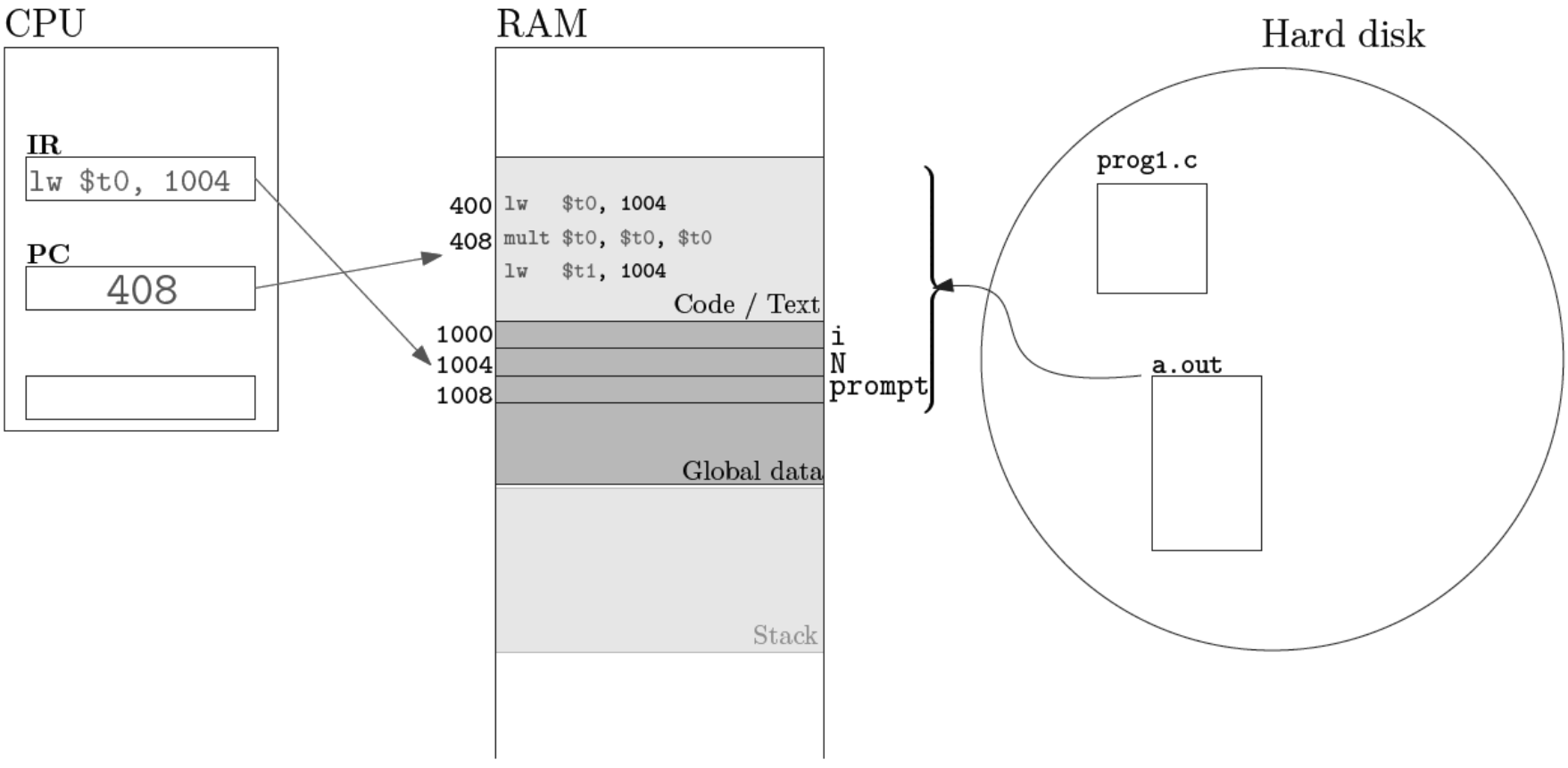
```
char prompt[] = "i";
int N = 20;
int i;
i = N*N + 3*N;
```

$\Longrightarrow$

MIPS machine instructions:

```
lw $t0, 1004 # fetch N
mult $t0, $t0, $t0 # N*N
lw $t1, 1004 # fetch N
ori $t2, $zero, 3 # 3
mult $t1, $t1, $t2 # 3*N
add $t2, $t0, $t1 # N*N + 3*N
sw $t2, 1000 # i = N*N + 3*N
```

Address binding



Address binding

Usually a program resides on a disk as a binary executable file. For execution, the program is brought into memory. Most systems allow a user process to reside in any part of the physical memory. Hence, the address space does not necessarily start at 00000.

The binding of instructions and data to memory can happen at any time toward the execution.

1. **Compile time:** The *absolute code* is generated and is bound to actual physical addresses.
2. **Load time:** The *relocatable code* is generated and is bound to relative addresses.
3. **Execution time:** The *absolute code* and *relocatable code* are bound together.

Address binding during compilation

- Location of program in physical memory must be known at compile time
- Compiler generates absolute code
  - Compiler binds names to actual physical addresses
- Loader copies executable file to appropriate location in memory
- If starting location changes, program will have to be recompiled

Address binding during loading

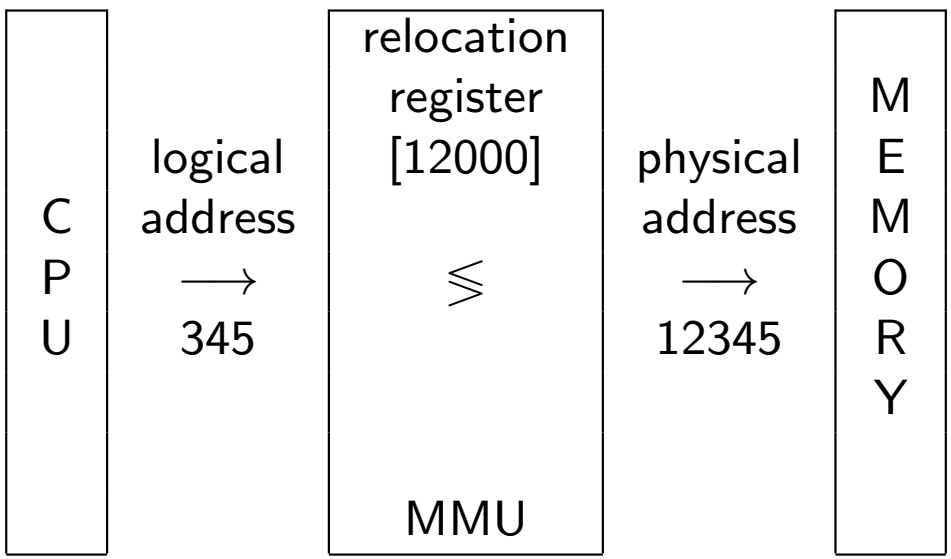
- Compiler generates relocatable code
  - Compiler binds names to relative addresses (offsets from starting address)
  - Compiler also generates relocation table
- Linker resolves external names and combines object files into one loadable module
- Loader converts relative addresses to physical addresses
- No relocation allowed during execution

Address binding during execution

- Programs/compiled units may need to be relocated during execution
- CPU generates relative addresses
- Relative addresses bound to physical addresses at run-time based on location of translated units
- Suitable hardware support required

Dynamic relocation

The user program never accesses the real physical addresses. The memory management unit (MMU) maps the logical address to a physical address.



Dynamic relocation

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the address binding that happens during execution time results in different logical and physical addresses.

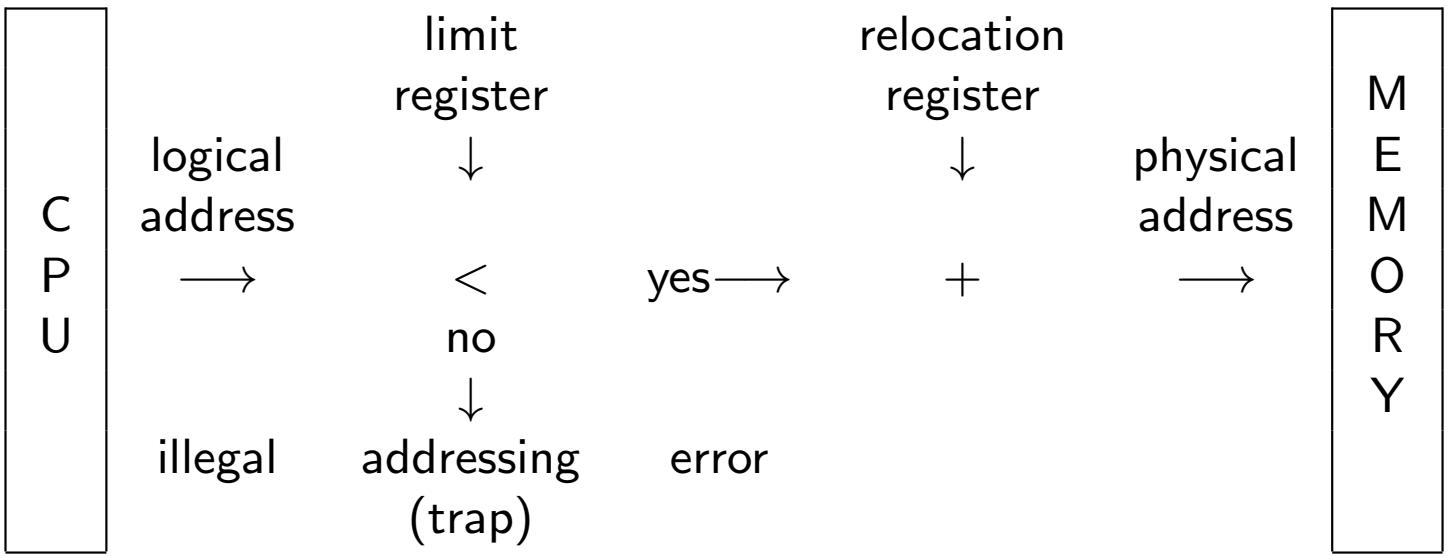
Dynamic loading

All routines are kept on disk in a relocatable load format. With dynamic loading a routine is not loaded until it is called.

When the main program is loaded into main memory during execution, it checks whether the called routine has been loaded. If not, the relocatable linking loader is employed to serve the purpose.

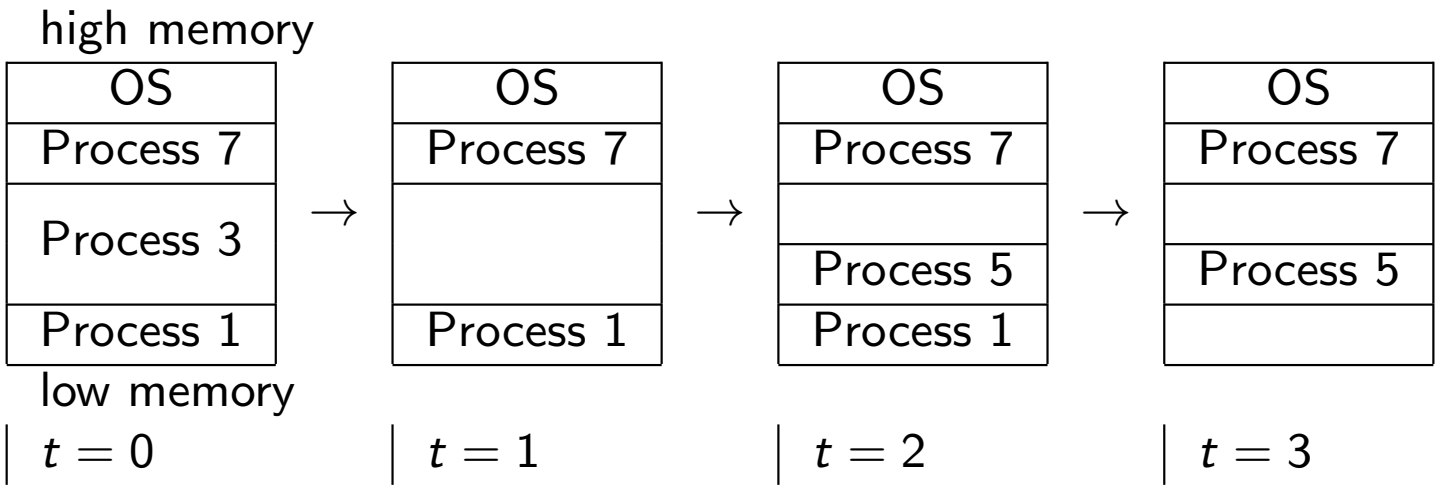
Protection of memory space

The MMU maps the logical address dynamically by adding the value in the relocation register.



Dynamic allocation of memory

Processes are dynamically assigned memory partitions of variable sizes on arrival in memory. As soon as a finishing process leaves memory, it might result into contiguous or non-contiguous holes.



Dynamic allocation of memory

It is a challenging task to allocate a memory partition of size n to a process from the free memory space (i.e., a list of free holes).

The following are some popular strategies to tackle this problem.

- **First fit:** Allocate the first hole that is large enough by searching either at the beginning of the set of holes or at the location where the previous first-fit search ended.
- **Best fit:** Allocate the smallest hole that is large enough by searching the entire list, unless the list is ordered by size.
- **Worst fit:** Allocate the largest hole by searching the entire list, unless it is sorted by size.

Note: Both the first fit and best fit perform better in terms of decreasing time and storage utilization.

Internal and external fragmentation

- **Internal Fragmentation:** This is caused by the unused memory that is internal to a partition.
- **External Fragmentation:** This is caused by the freed memory that is non-contiguous.

How does dynamic memory allocation affect fragmentation?

Both the first fit and best fit strategies suffer from *external fragmentation*. As processes arrive and leave memory, they turn the free memory space broken into little pieces.

The 50-percent rule

Even in the optimized version of the first fit strategy, given N allocated memory blocks, another $0.5N$ blocks will be lost due to fragmentation.

Addressing the fragmentation problem

The internal fragmentation can be solved by assigning space to the process via *dynamic partitioning*. It allocates only the amount of space requested by the process.

The following strategies help to solve the problem of external fragmentation.

1. Through *compaction*, we can shuffle the memory contents such that all free memory get placed together in one large block.
2. Through *paging*, we can allow the logical address space of processes to be non-contiguous, thereby allocating physical memory wherever such memory is available.

Basics of paging

Paging is a memory management scheme that allows a process's physical address space to be non-contiguous.

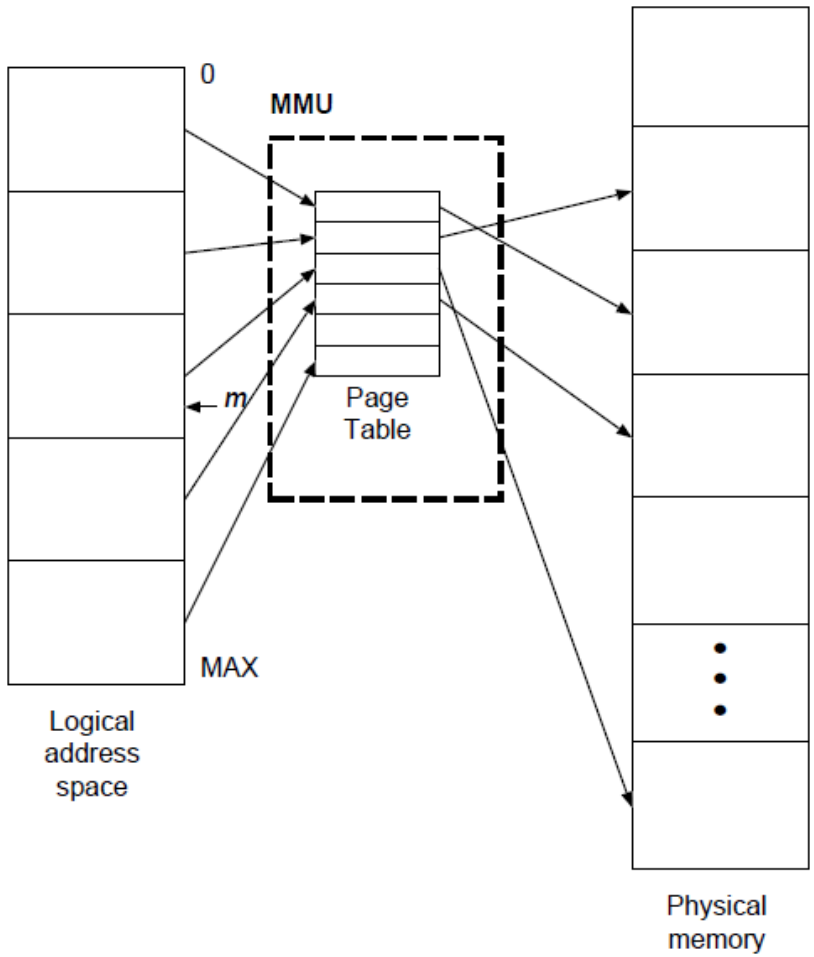
Paging involves breaking physical memory into *frames* (fixed-sized blocks) and breaking logical memory into *pages* (blocks of the same size as frames).

The frame size has the following characteristics:

- It is defined by the hardware.
- It should be power of 2.
- It typically consists of 4 KB - 1 GB.

Basics of paging

When a process with n pages has to be loaded, n free frames have to be found. The kernel keeps track of free frames and the page table translates logical page numbers to physical frame addresses.



Address translation

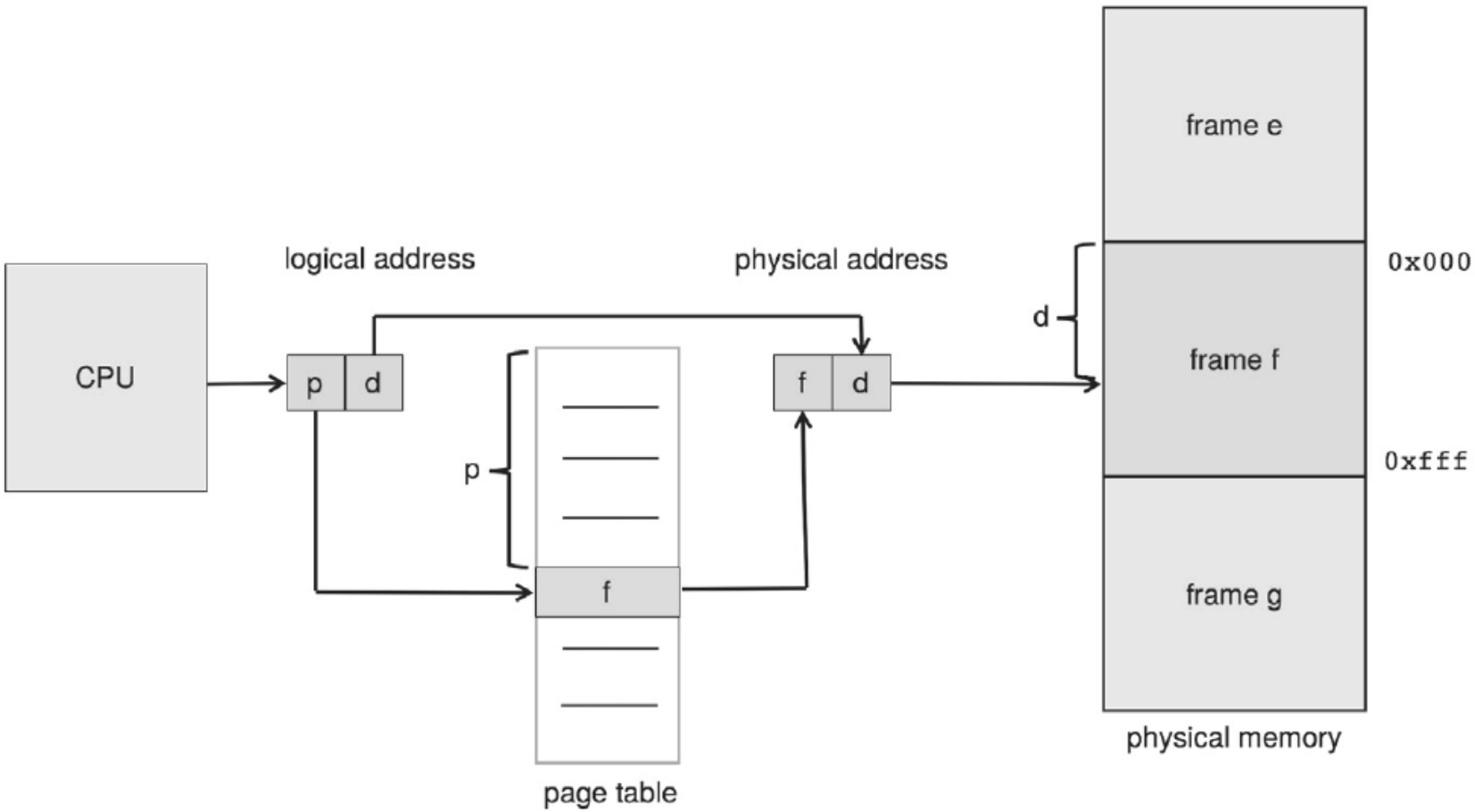
As the page table (PT) accommodates an array of frame addresses, the following properties hold:

- If the logical address space of process i has k pages (say $p_0, p_1, \ldots, p_{k-1}$), then it must have k valid entries in PT.
- $PT[i]$ contains the starting physical address of frame in which the process i is stored.
- If m is the logical address, then the entry $\lfloor m/s \rfloor$ in PT should be consulted, where s denotes the page size.
- If m is the logical address, then the byte $m \mod s$ within the frame pointed by $PT[i]$ should be accessed.

Implementation of paging

Every address generated by the CPU is divided into two parts.

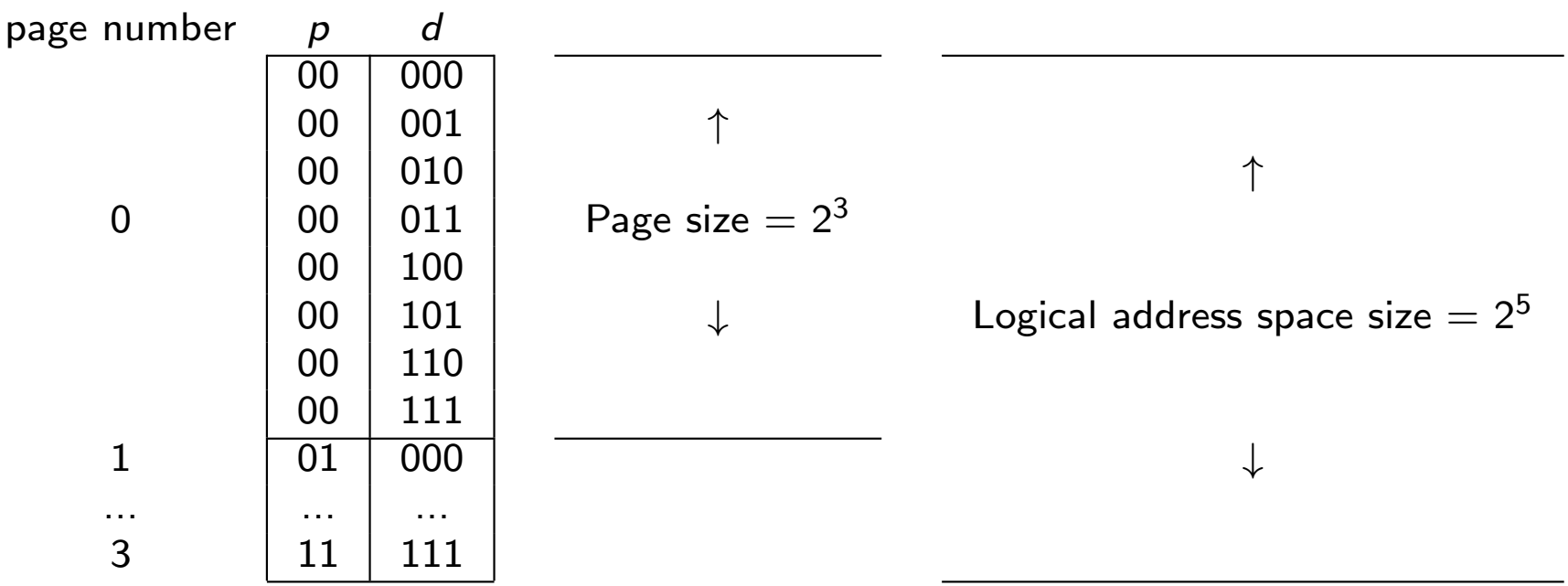
1. A page number (p)
2. A page offset (d)



Implementation of paging

Let the size of logical address space is 2^m and the size of page is 2^n, then

1. p denotes the $m - n$ higher order bits of logical address
2. d denotes the n lower order bits of logical address



Implementation of paging

Page table:

- part of process context
- during context switch, saved page table is used to reconstruct hardware page table
- may be used by some system calls to translate logical addresses to physical addresses in software

Frame table:

- maintained by kernel
- contains 1 entry per physical page frame
  - whether free or allocated
  - allocation information (PID, page)

Advantages and disadvantages

- Memory protection is automatic and process cannot address memory outside its own address space.
- No external fragmentation happens.
- Internal fragmentation can happen on half a page per process, on average.
- Employing small frames causes less fragmentation
- Employing large frames causes page table overhead but I/O is more efficient.

Paging hardware – With special purpose registers

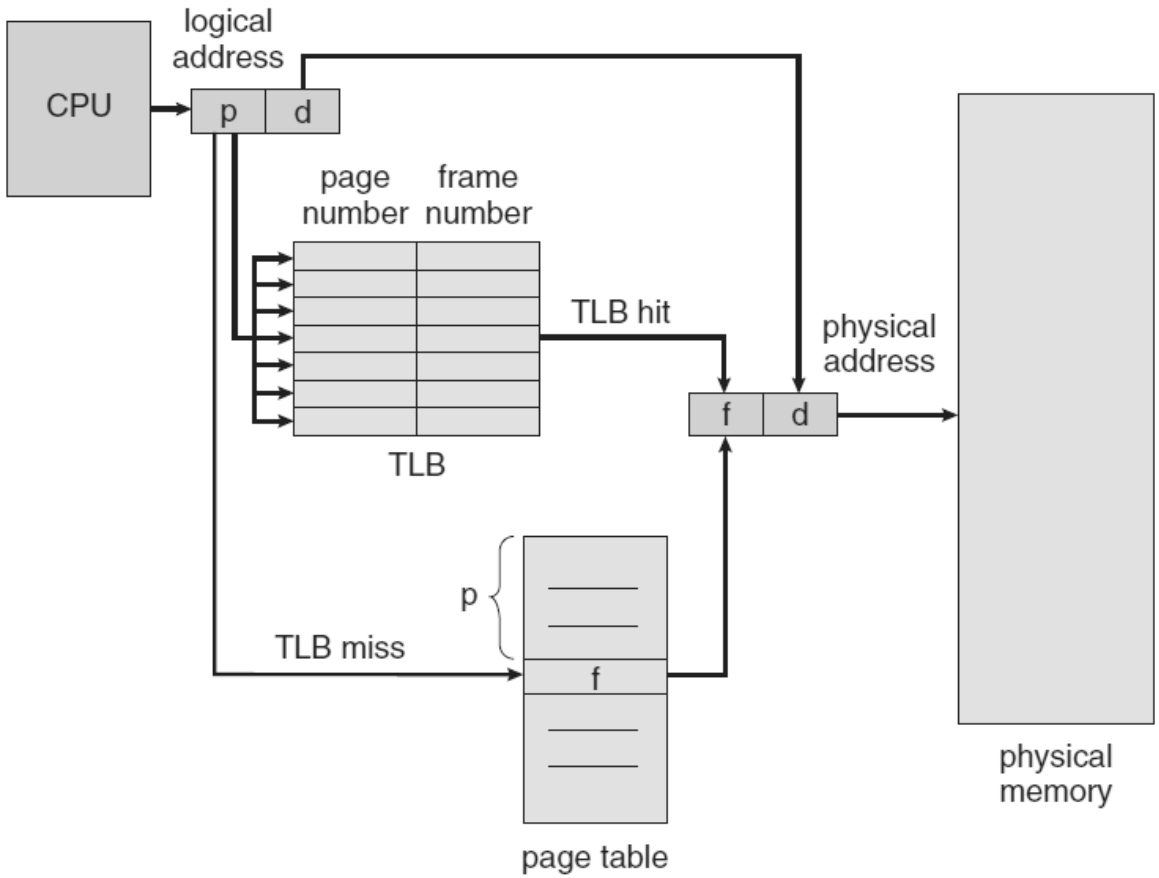
- Page table is stored in a set of dedicated, high-speed registers only
- Instructions to load/modify page table registers are privileged
- Not an acceptable solution for large address space

Paging hardware – With RAM + PTBR + PTLR

- Page table is stored in main memory
- Base address of the page table is stored in page table base register (PTBR)
- Length of the page table is stored in page table length register (PTLR)
- Context switch involves changing PTBR and PTLR but the page table does not need to be saved/restored
- Two physical memory accesses are needed per user memory access, thereby slowing down memory access by a factor of 2

Paging hardware – With RAM + TLB

Translation look-aside buffers (TLBs) are costly, small, but fast-lookup hardware cache (built using high-speed memory).



Note: TLB has to be flushed on context switch.

Using Translation look-aside buffer (TLB)

The percentage of times a page number of interest is found in the TLB is called the *hit ratio*. A hit ratio of 0.8 (or 80%) signifies that the desired page number can be found in the TLB 80% of the time.

Suppose it takes 10 ns to access memory, then a mapped-memory access takes a 10 ns when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 ns) and then access the desired byte in memory (10 ns), for a total of 20 ns.

Hence, the *effective memory-access time* turns out to be:
$0.80 * 10 + 0.20 * 20 = 12$ ns.

Note: We are assuming that a page table lookup takes only one memory access, but it can take more in reality.

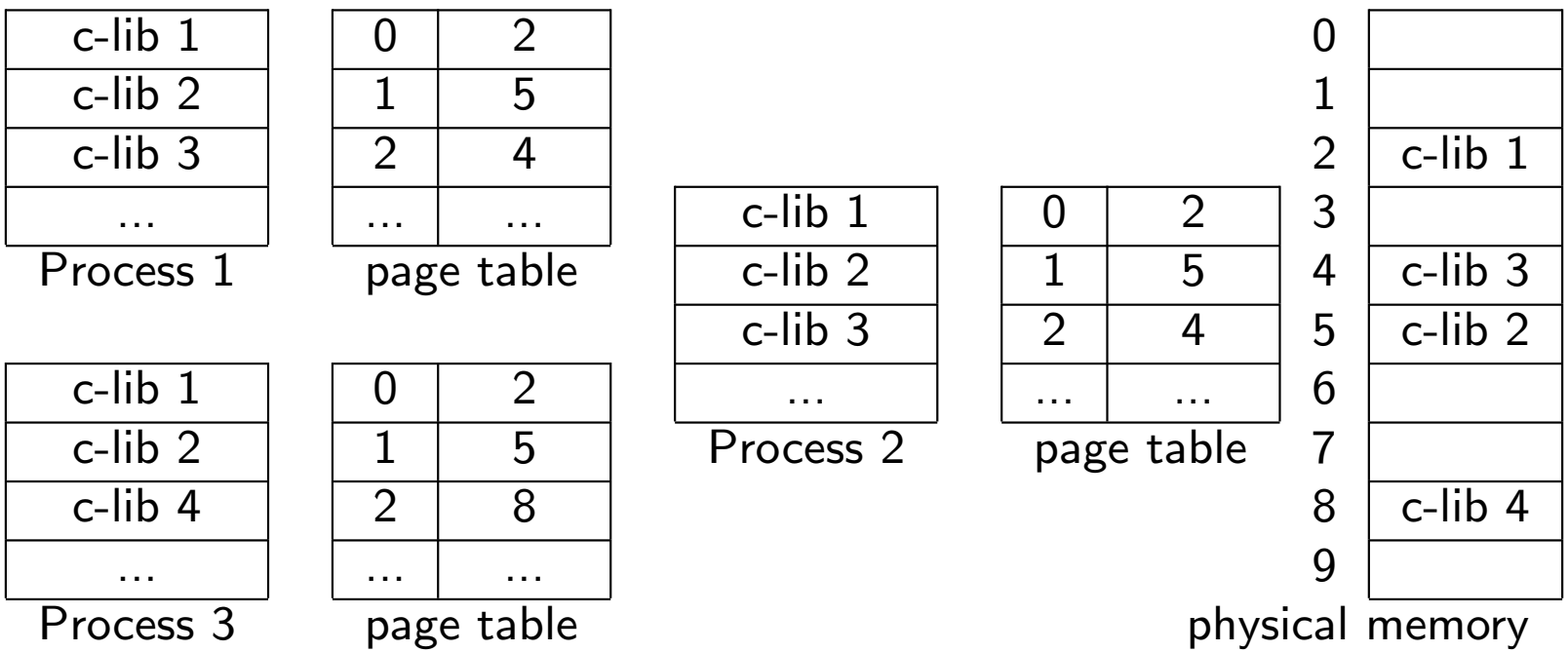
Protection of memory space in paging

Memory protection in a paged environment is accomplished by *protection bits* associated with each frame. One additional bit, known as *valid-invalid bit*, is generally attached to each entry in the page table.

- When the protection bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the protection bit is set to invalid, the page is not in the process's logical address space.

Sharing pages

Paging enables the sharing of common code, which is of practical demand in multiprocessor environments. This is primarily used for sharing *reentrant* (read-only) codes associated with heavily used programs.



Here, we consider that "c-lib i" denotes a standard C library.

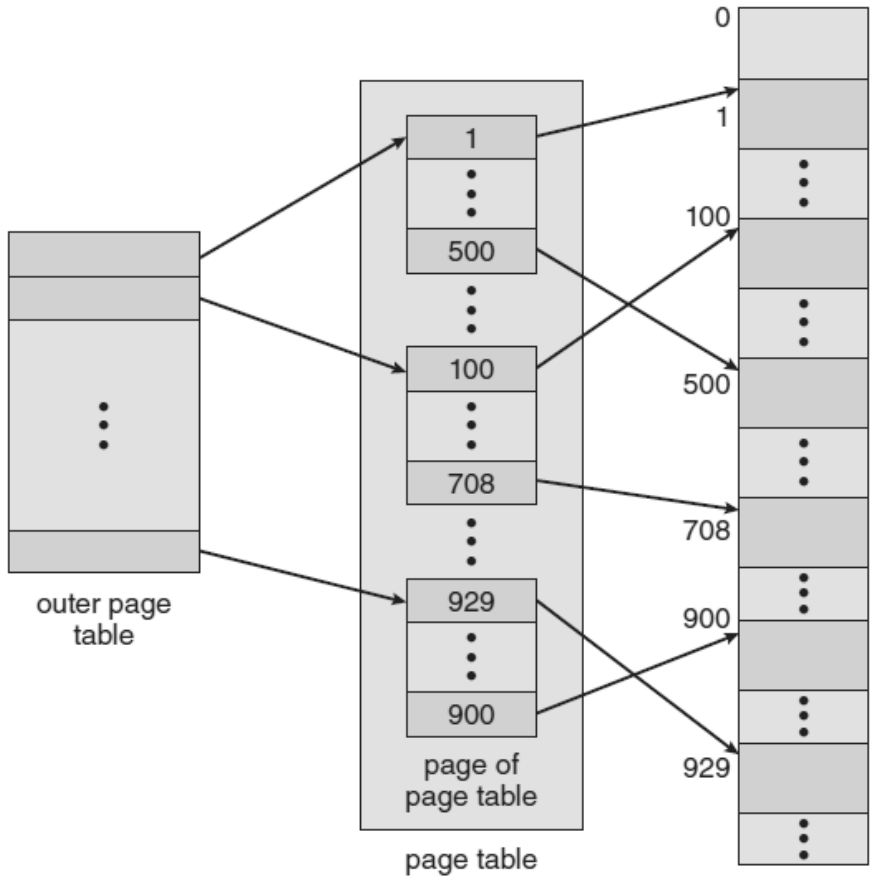
Configuring a page table

The structure of a page table can be configured in different ways as listed below.

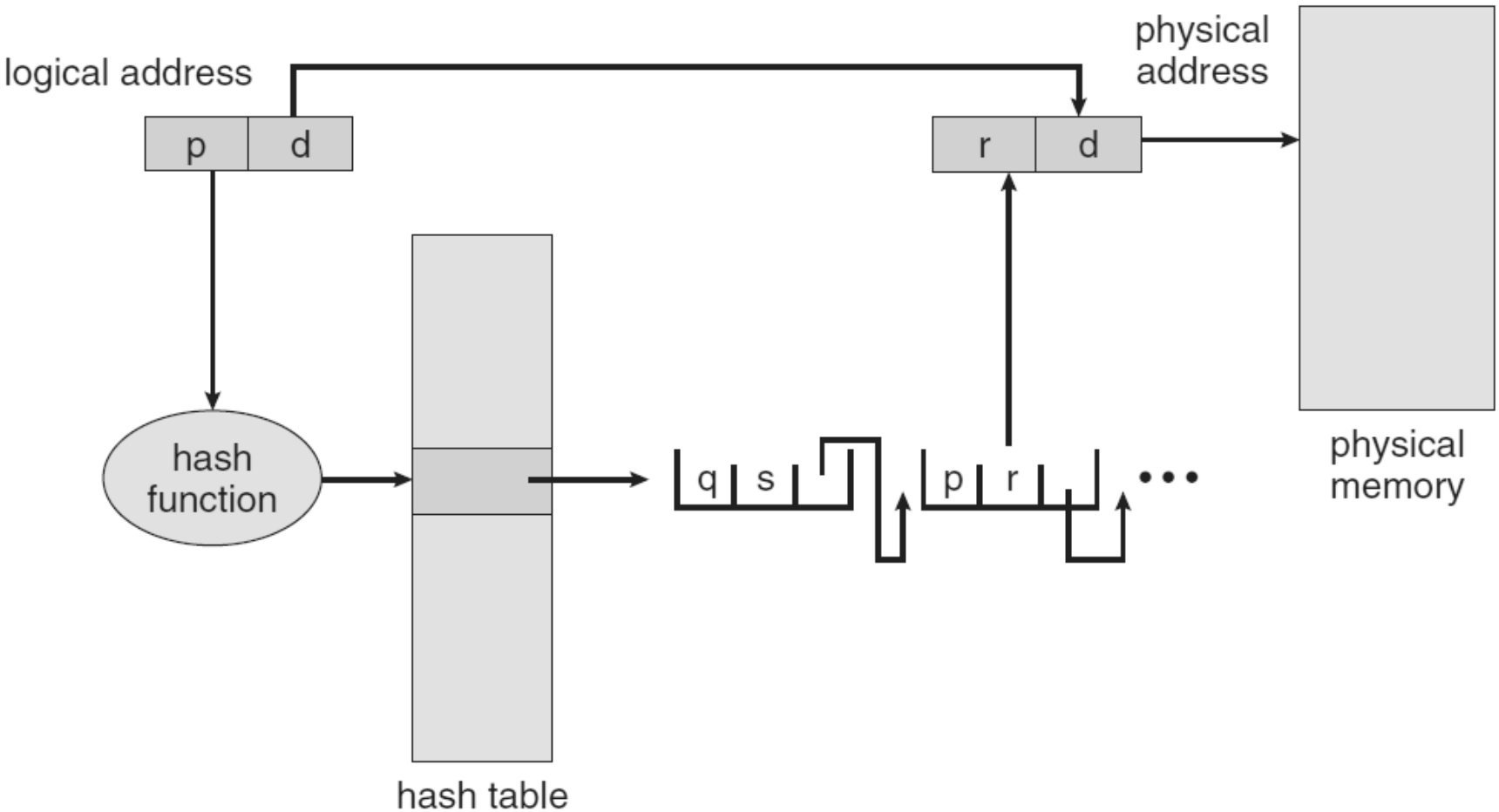
- Hierarchical page table
- Hashed page table
- Inverted page table

Hierarchical page table

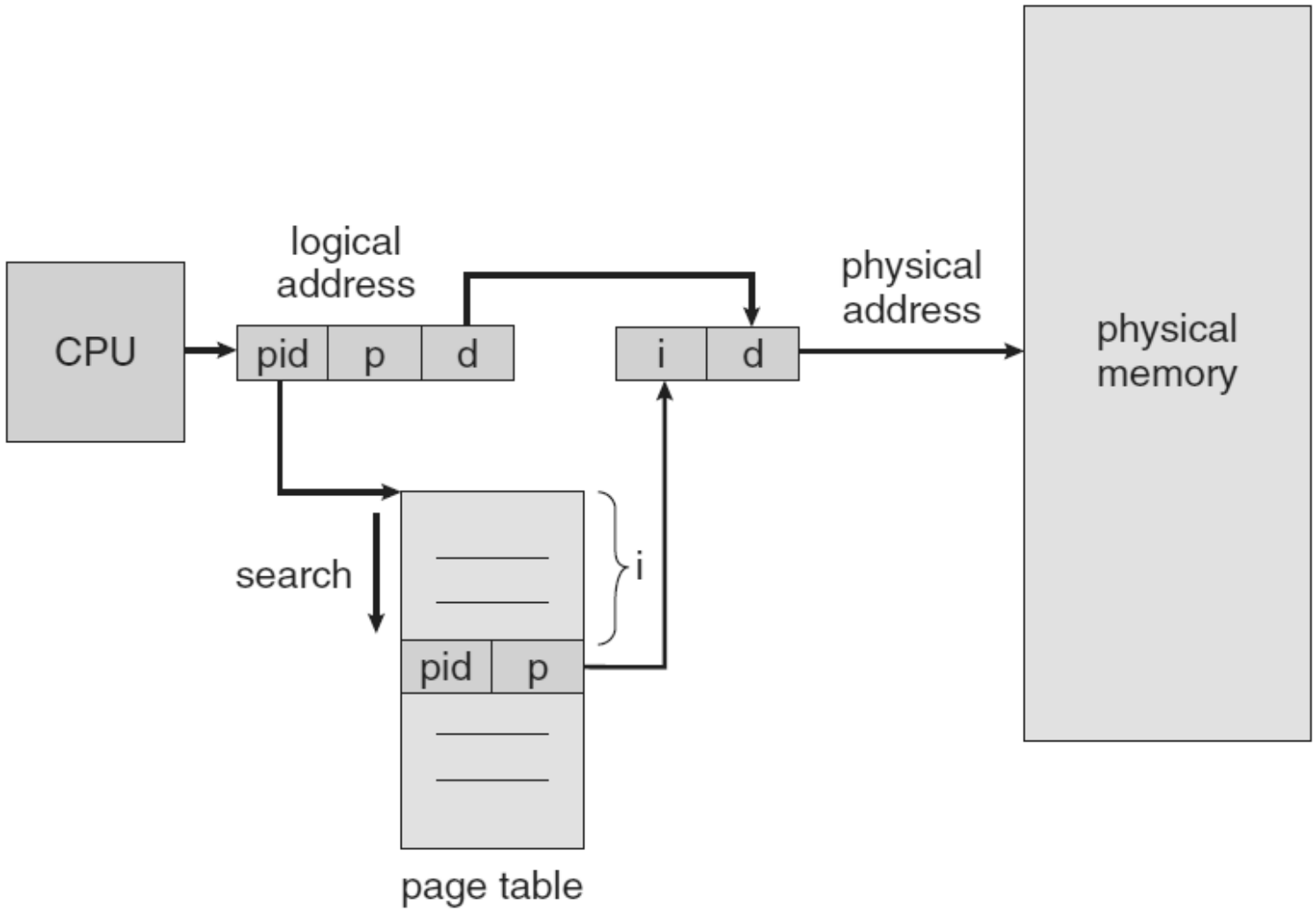
If the page table is large, we might require a secondary (outer) page table to keep track of the pieces of primary (inner) page table. However, many levels of paging may be required for state-of-the-art architectures.



Hashed page table



Inverted page table



Basics of swapping

Motivation:
Consider that a set of processes are resident in memory and occupy all available physical spaces. Now one of the processes forks to create a child.

Swapping allows a process (or a portion of the process) to be temporarily swapped out of memory to a *backing store*, to release memory for another process, and then swapped back in memory for continued execution.

Approaches of swapping

1. Round-robin
   - When a process's quantum expires, it is swapped out, another process is swapped into freed memory.
   - Scheduler allocates next quantum to some other process in memory.
2. Priority-based (roll out, roll in)
   - When higher priority process arrives, lower-priority process is swapped out.
   - When higher priority process finishes, lower priority process can be swapped in.

Note: The time quantum is same for every process.

Performance issues

Problem:
Swapping increases the time of context switch because it involves disk transfer is involved. Consider the example where: Process size = 100K, Transfer rate = 1Mbps.
Hence, swap-out time + swap-in time = 2 * (100K / 1 Mbps) = 200ms.
Note that this will also include some operational time.

Solution:
The time quantum should be large compared to swap time for a good utilization.

Performance issues

Problem:
If a process is swapped out while waiting for input into buffer in user memory, addresses used by I/O devices may be wrong.

Solution:
The processes with pending I/O should never be swapped.

OR

I/O operations are always done using OS buffers (data can be transferred from OS to user buffer when a process is swapped in).

Performance issues

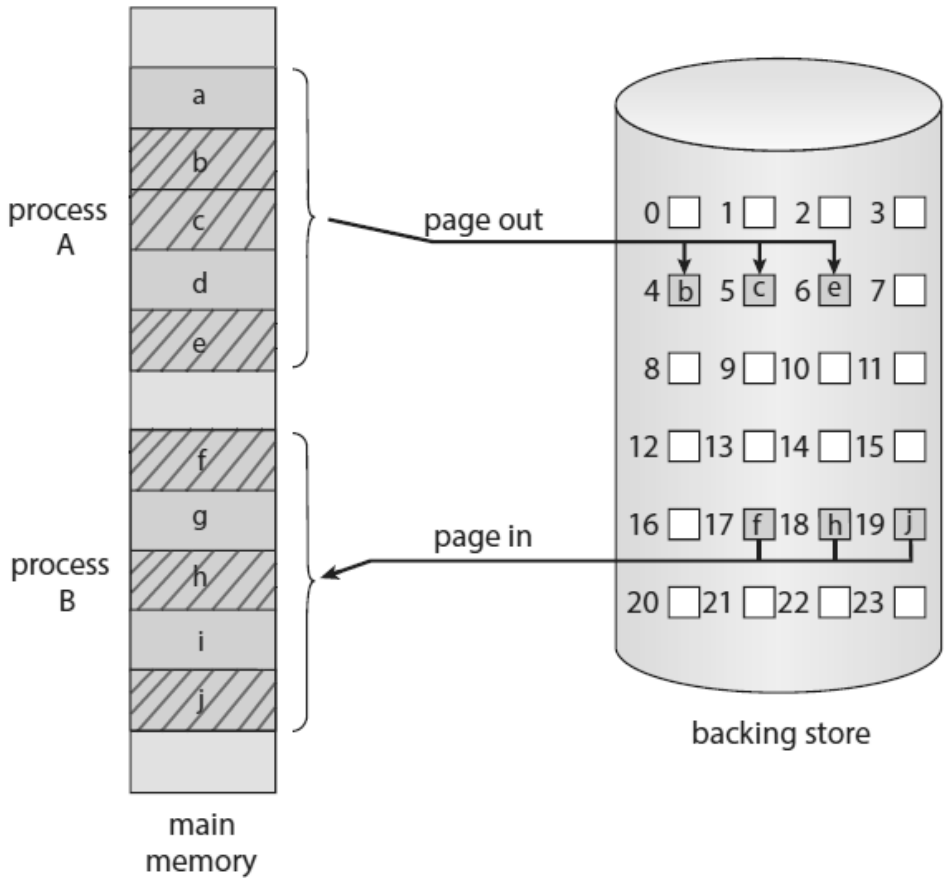
Problem:
Swapping might cause external fragmentation.

Solution:
Apply compaction as follows.

1. Processes which have to be moved are swapped out
2. Memory is compacted by merging holes
3. Swapped-out processes are swapped in to different memory locations to minimize fragmentation

Swapping with paging

Recent OSs employ a variation of swapping in which pages of a process, rather than an entire process, can be swapped. This may allow physical memory to be oversubscribed, but avoids the cost of swapping the entire processes.



Background

High-level code:

```
switch(operator){
    case '+':
        // Statement(s)
        break;
    case '-':
        // Statement(s)
        break;
    case '*':
        // Statement(s)
        break;
    case '/':
        // Statement(s)
    default:
      // Default statement(s)
}
```

$\Longrightarrow$

MIPS machine instructions:

???

Background

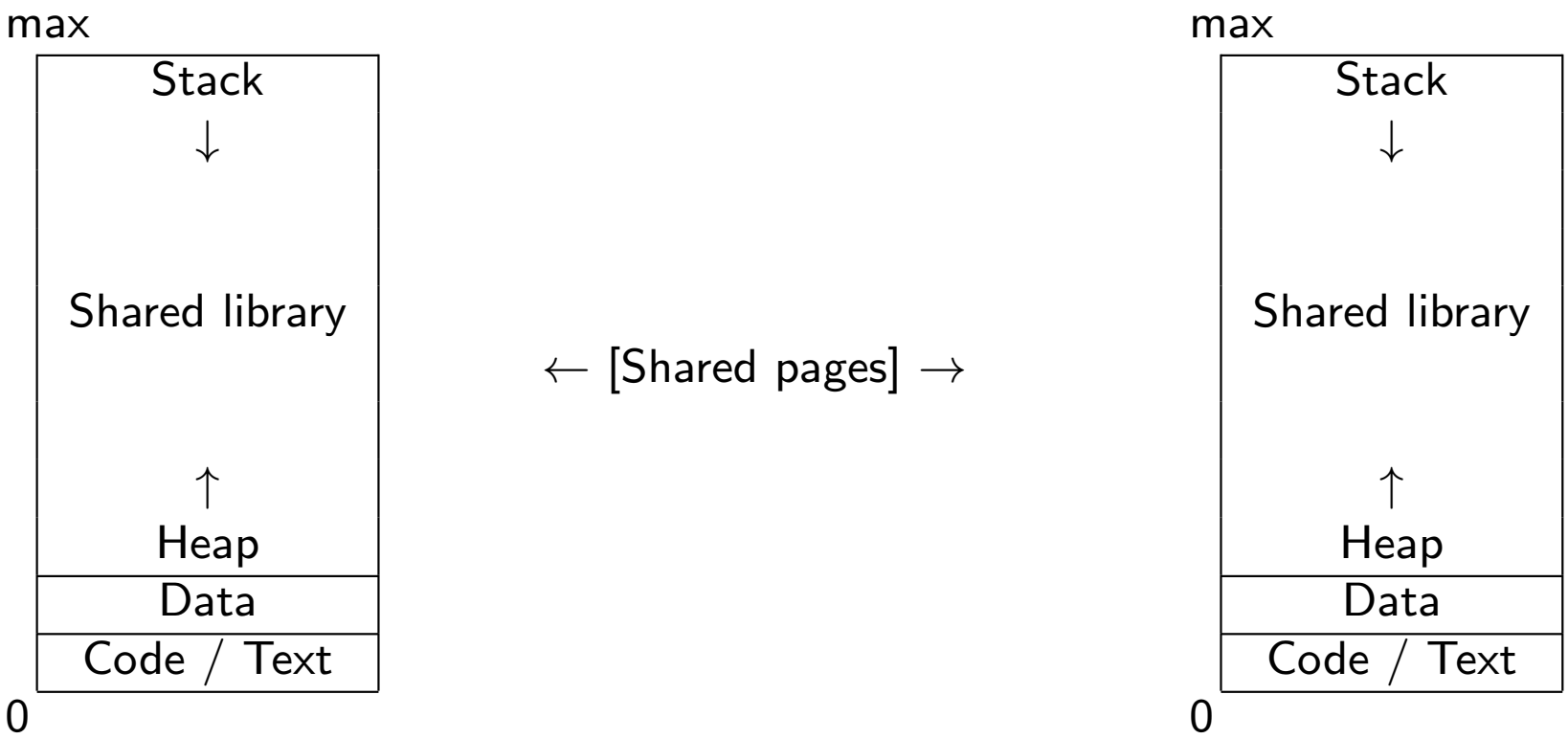
Entire logical address space does not have to be loaded into memory.

- Some code may be executed rarely (e.g., error handling routines for unusual error conditions, code implementing rarely used features).
- Arrays/tables may be allocated more memory than required.

Virtual memory is a mechanism to allow execution of processes without requiring the entire process to be in memory.

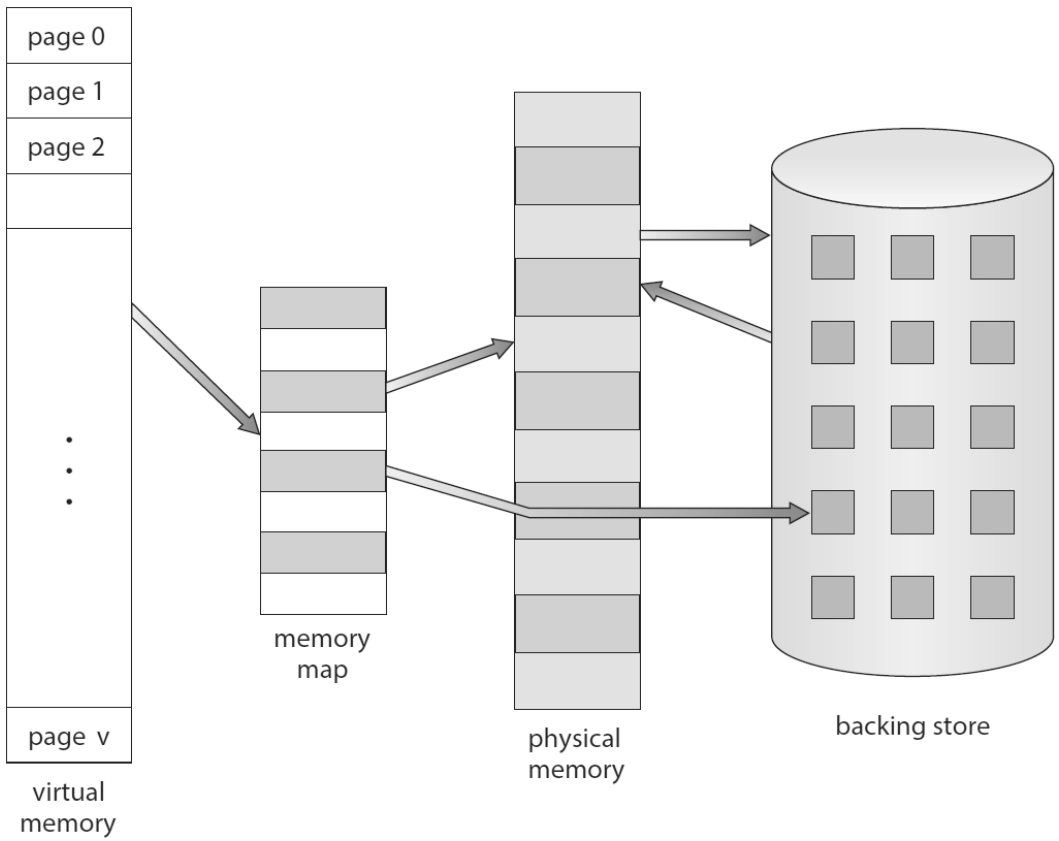
Virtual address space

The heap is allowed to grow upward in memory as it is used for dynamic memory allocation. Similarly, the stack is allowed to grow downward in memory through successive function calls.



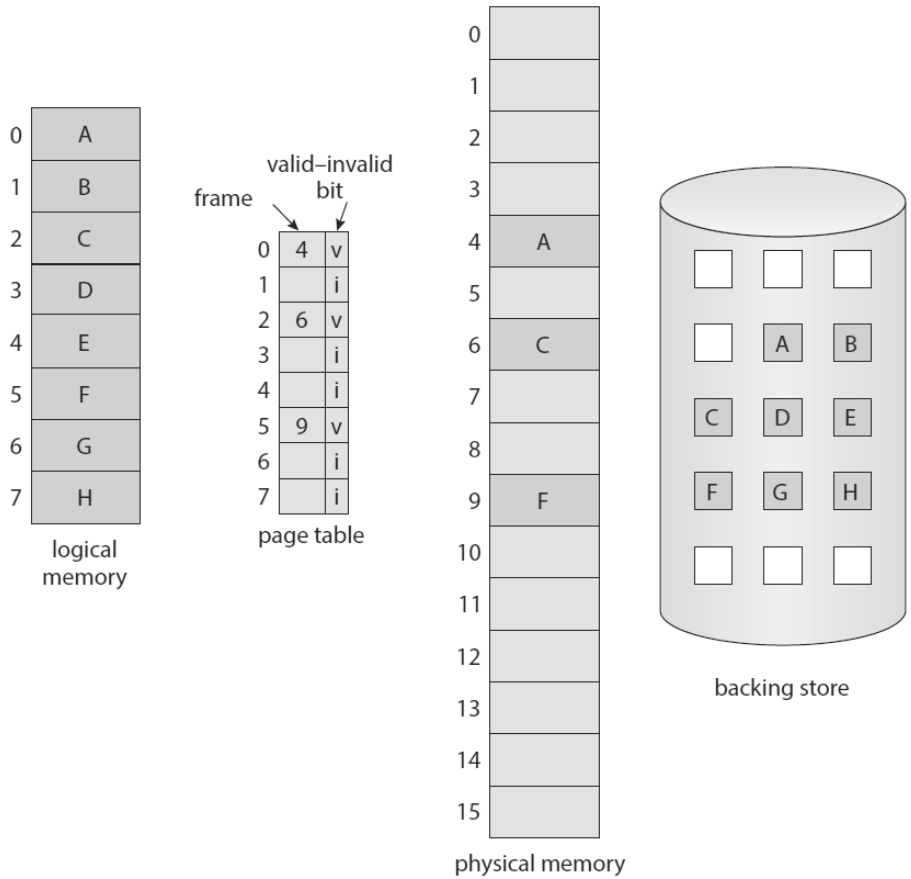
Virtual memory

Virtual memory involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.



Demand paging

With demand-paged virtual memory, pages are loaded in physical memory (primary) only when they are demanded during program execution. Pages that are never accessed stay in the logical memory (secondary).



Demand paging – The implementation

The *valid-invalid bit* is used for distinguishing the pages that are in primary memory and secondary storage.

- A bit is set to *valid*
  – when the associated page is both legal and in memory.
- A bit is set to *invalid*
  – when the page either is not valid (i.e., not in the logical address space of the process)
  – when it is valid but is currently in secondary storage.

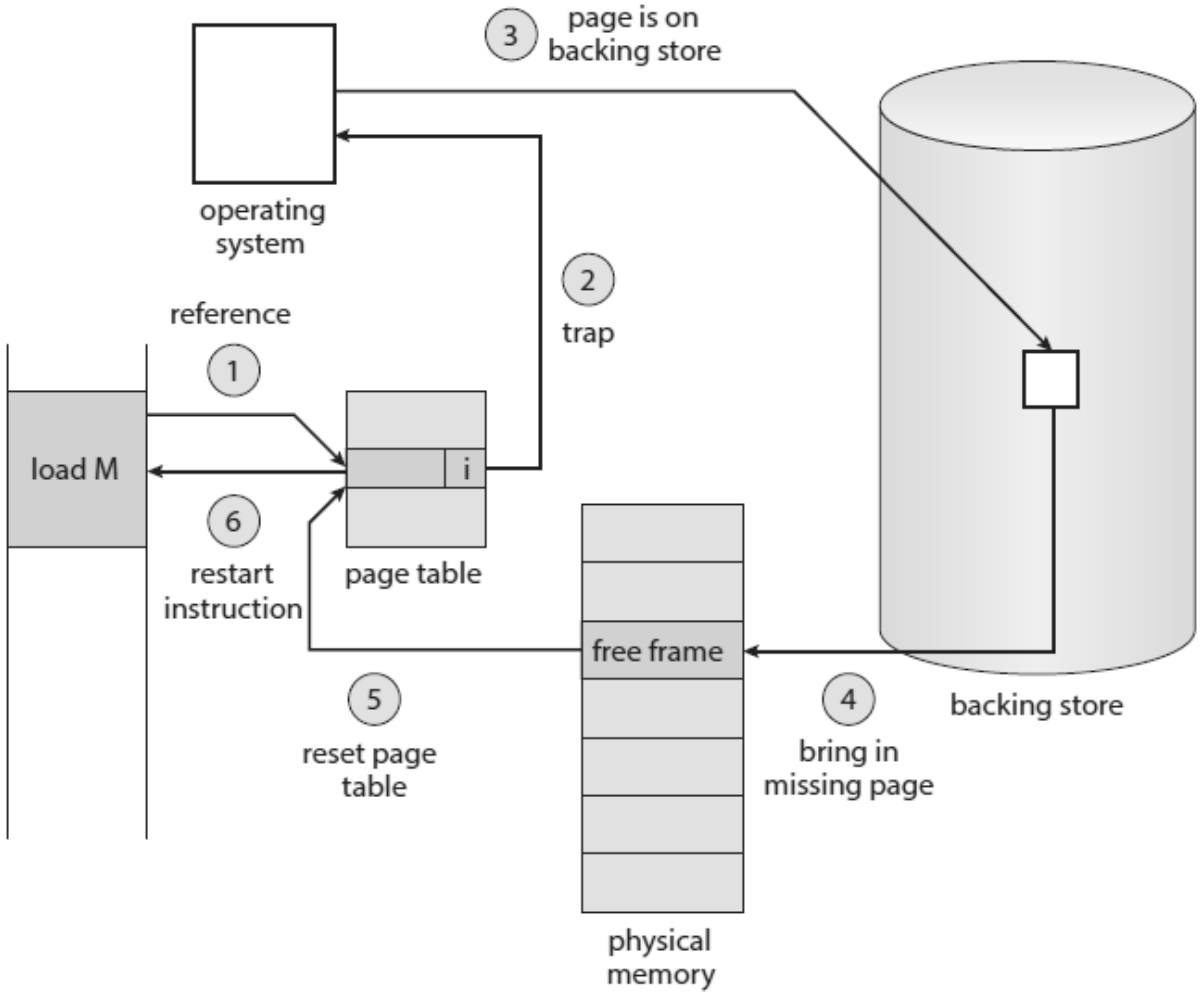
Page fault in demand paging

If a process tries to access a page that was not brought into memory, CPU translates the address through the page table and will notice that the invalid bit is set, causing a trap to the operating system. This is known as page fault.

Note: Marking a page as invalid will have no effect if the process never attempts to access that page.

Page-fault service routine in demand paging

We can adopt the following steps to handle page fault.



Page fault in demand paging

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible.

In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

Pure demand paging

In pure demand paging, we never bring a page into memory until it is required.

Consider an extreme case. We start executing a process with no pages in memory. When the OS sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults.

Free-frame list

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory. To resolve page faults, most operating systems maintain a free-frame list, a pool of free frames for satisfying such requests.

Note: Free frames must also be allocated when the stack or heap segments from a process expand.

Performance of demand paging

As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.

Suppose the memory access time is denoted by t_m. Let us consider p be the probability of a page fault and t_p denotes the time to recover from page fault. Then the effective memory-access time is:

$$(1 - p) * t_m + p * t_p.$$

Note: We expect to have page faults rarely, i.e., p to be close to zero.

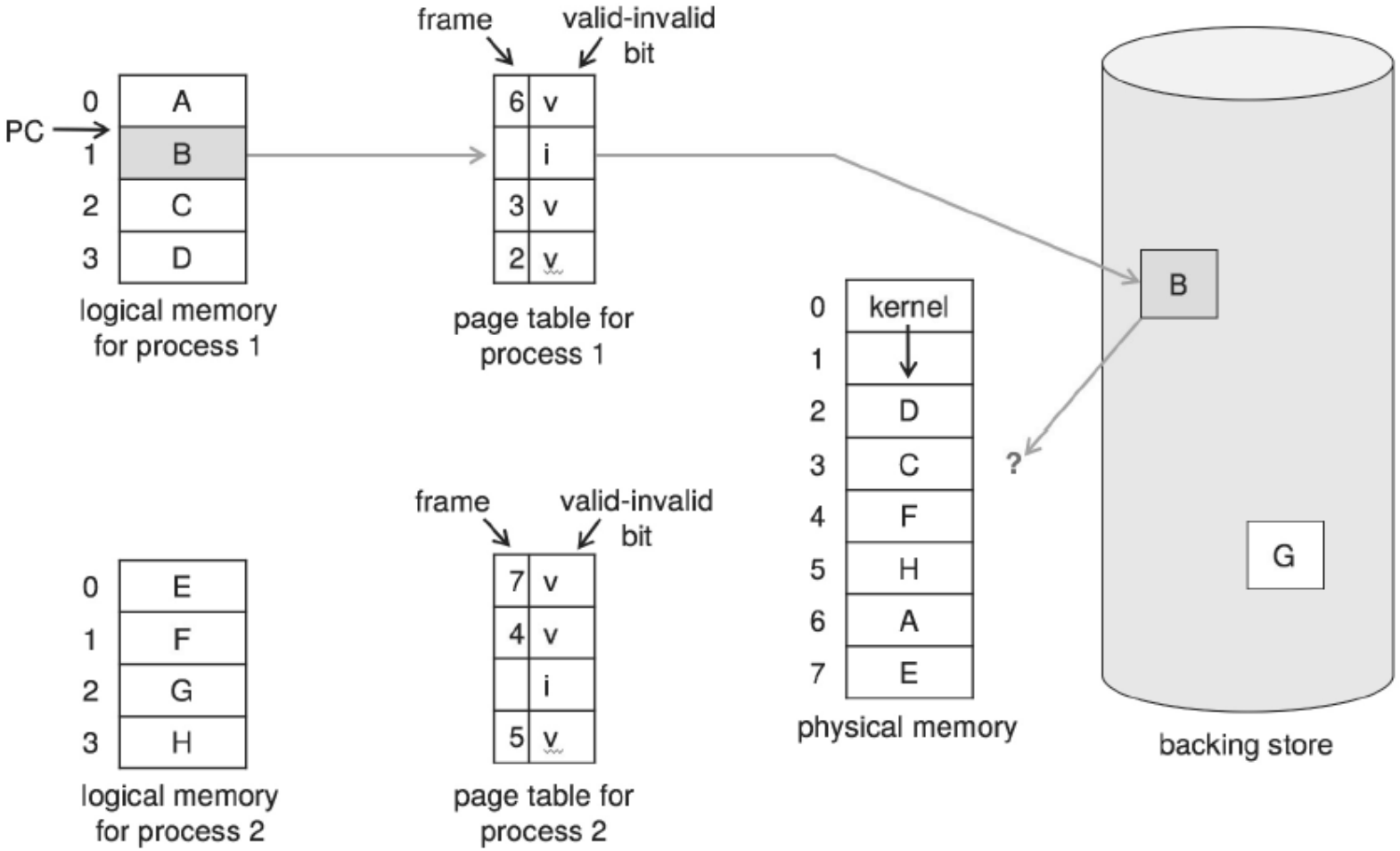
Does demand paging support `fork()`?

Process creation using the `fork()` system call does not currently create a copy of the parent's address space for the child, duplicating the pages belonging to the parent. It can initially bypass the need for demand paging by using *copy-on-write* technique, which is similar to page sharing.

The *copy-on-write* technique allows the parent and child processes initially to share the same pages. If either process writes to a shared page, a copy of the shared page is created.

Page replacement

If no frame is free, page replacement helps to find a frame that is not currently being used. It frees the frame by writing its contents to swap space and changing the page table accordingly.



Page-fault service routine with page replacement

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
   - 2.1 If there is a free frame, use it.
   - 2.2 If there is no free frame, use a page-replacement algorithm to select a *victim frame*.
   - 2.3 Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

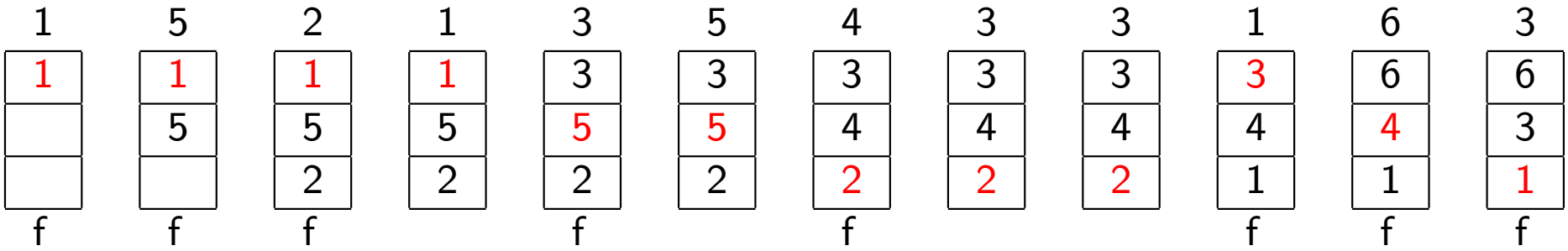
Page replacement algorithms – FIFO

Pages are kept in a FIFO queue.

- When a page is brought into memory, it is added at tail of queue.
- When a page has to be replaced, page at head of queue is selected.

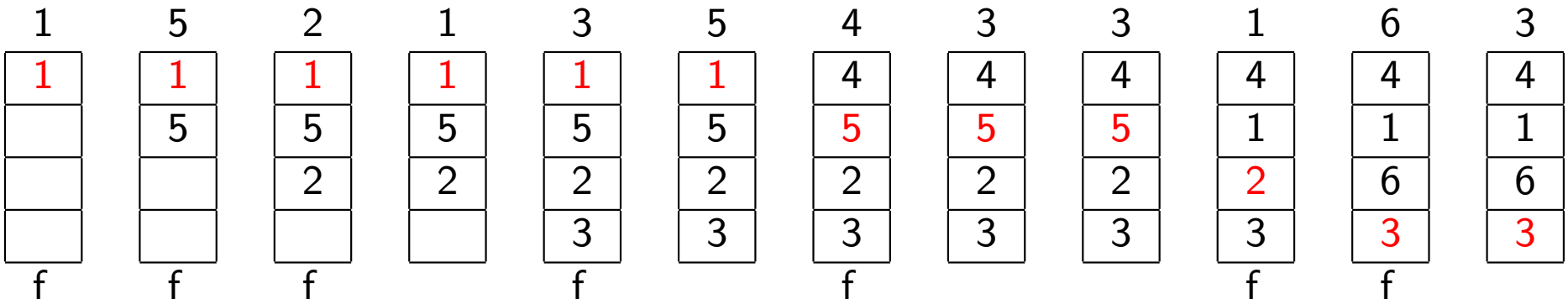
Page replacement algorithms – FIFO

Consider frame size = 3, reference string = 1 5 2 1 3 5 4 3 3 1 6 3



Page faults: 8 (the candidate to replace is shown in red)

Consider frame size = 4, reference string = 1 5 2 1 3 5 4 3 3 1 6 3



Page faults: 7 (the candidate to replace is shown in red)

Page replacement algorithms – Belady's anomaly

Belady's anomaly states that increasing the frame size does not always decrease the number of page faults.

Note: Belady's anomaly will never occur if the pages in memory with n frames is a subset of the pages in memory with $n+1$ frames.

Page replacement algorithms – Belady's anomaly

Consider frame size = 3, reference string = 3 4 5 6 3 4 7 3 4 5 6 7

3	4	5	6	3	4	7	3	4	5	6	7
3	3	3	6	6	6	7	7	7	7	7	7
	4	4	4	3	3	3	3	3	5	5	5
		5	5	5	4	4	4	4	4	6	6
f	f	f	f	f	f	f			f	f	

Page faults: 9 (the candidate to replace is shown in red)

Consider frame size = 4, reference string = 3 4 5 6 3 4 7 3 4 5 6 7

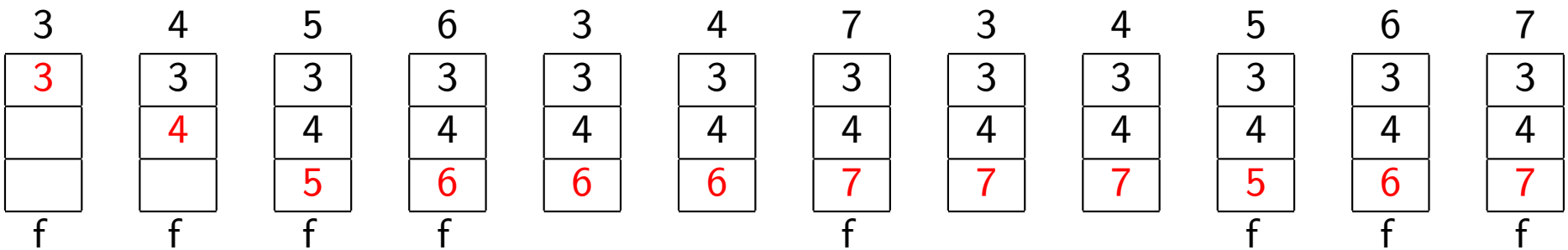
3	4	5	6	3	4	7	3	4	5	6	7
3	3	3	3	3	3	7	7	7	7	6	6
	4	4	4	4	4	4	3	3	3	3	7
		5	5	5	5	5	5	4	4	4	4
			6	6	6	6	6	6	5	5	5
f	f	f	f			f	f	f	f	f	f

Page faults: 10 (the candidate to replace is shown in red)

Page replacement algorithms – Optimal

Page that will not been used for the longest period of time is replaced.

Consider frame size = 3, reference string = 3 4 5 6 3 4 7 3 4 5 6 7

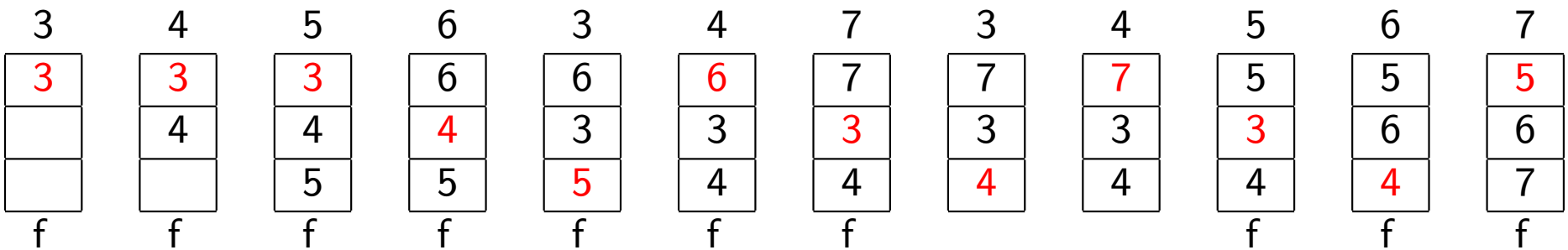


Page faults: 8 (the candidate to replace is shown in red)

Page replacement algorithms – LRU

Pages are kept in a LIFO stack. Page that has not been used for the longest time is replaced.

Consider frame size = 3, reference string = 3 4 5 6 3 4 7 3 4 5 6 7



Page faults: 10 (the candidate to replace is shown in red)

Page replacement algorithms – LRU-Approximation

- **Additional-Reference-Bits Algorithm:** We can gain additional ordering information by recording the reference bits at regular intervals. An 8-bit shift register can contain the history of a page use for the last eight time periods.
- **Second-Chance Algorithm:** When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. This can be done with a circular queue.
- **Enhanced Second-Chance Algorithm:** We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair.

Page replacement algorithms – Counting-based

- **Least frequently used (LFU) page-replacement:** It requires that the page with the smallest count be replaced. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. To address this, we can shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- **Most frequently used (MFU) page-replacement:** It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Thrashing

Consider what occurs if a process does not have enough frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.