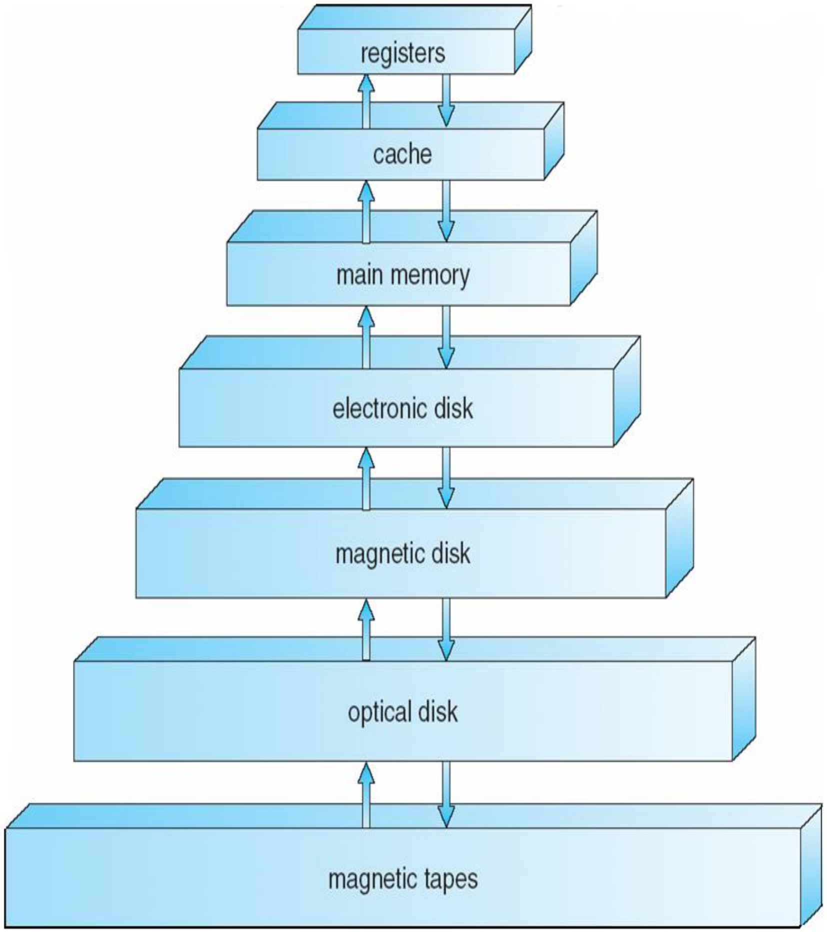
Main Memory (I)

Storage Hierarchy

- ❑Main memory, cache and registers are the only storages that CPU can access directly
- ❑Program must be brought (from disk) into memory and placed within a process image for it to be run



Program building (C Program)

- Preprocessing (Preprocessor)
  - It processes include files, conditional compilation instructions and macros.
  - Command: $cpp hello.c hello.i
  - hello.c is source program, hello.i is ASCII intermediate code
- Compiling (Compiler)
  - It takes the output of the preprocessor and generates assembly source code
  - Command: $cc hello.i –o hello.s

Program building (C Program)

- Assembly (Assembler)
  - It takes the assembly source code and produces an assembly listing with offsets.
  - The assembler output is stored in an object file.
  - Command:$as –o hello.o hello.s
- Linking (Linker)
  - It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file.
  - Linux command for linker is ld

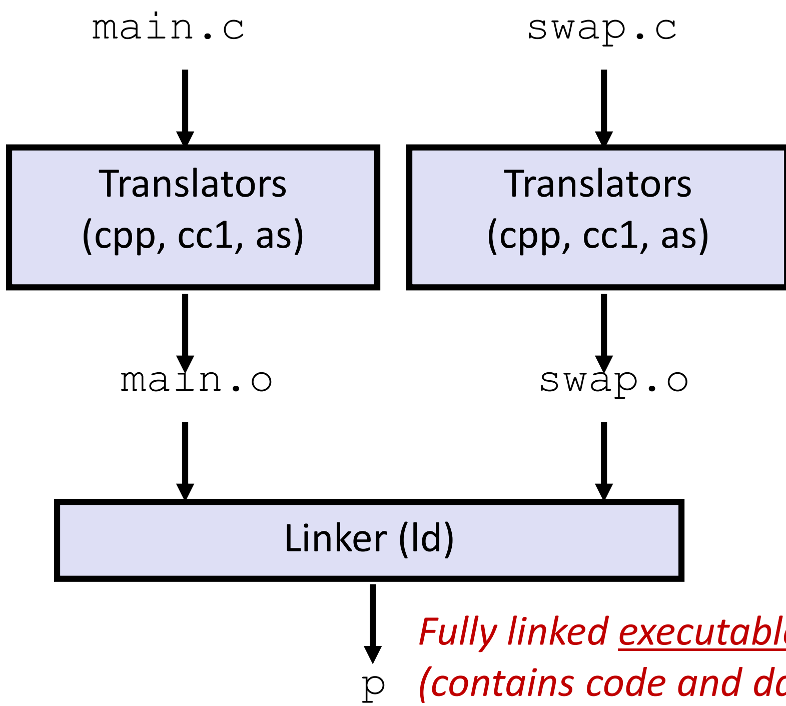
Example C Program

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

Int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



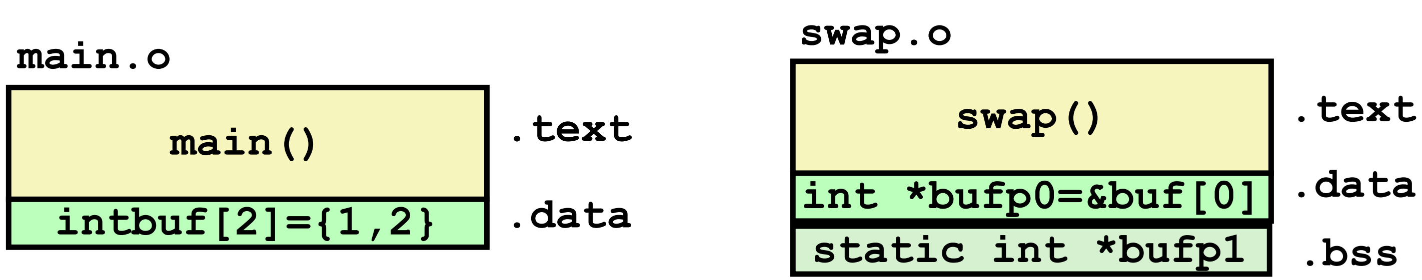
Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` and `swap.c`)

Relocatable Object Files

- A re-locatable object file consists of code and data sections.
  - Code section contains read-only instruction binary
  - Data section contains initialized global variables and uninitialized global variables
  - Code and data sections start at address zero



Linking

- Unix ld program takes as input a collection of relocatable object files as command line arguments and generate as output a fully linked executable object file that can be loaded.

Building Executable involves two tasks

- Linker performs two main tasks to build executable
  - Symbol Resolution
    - Object files define and reference symbols.
    - Symbol resolution associates each symbol reference to its definition.
  - Relocation
    - Compiler and Assembler generate code and data sections that start at address zero.
    - Linker relocates these sections by associating a memory location with each symbol definition.

Resolving Symbols

Global

Global

External

Local

```
int buf[2] = {1, 2};

int main()
{
   swap();
   return 0;
}
```

main.c

External

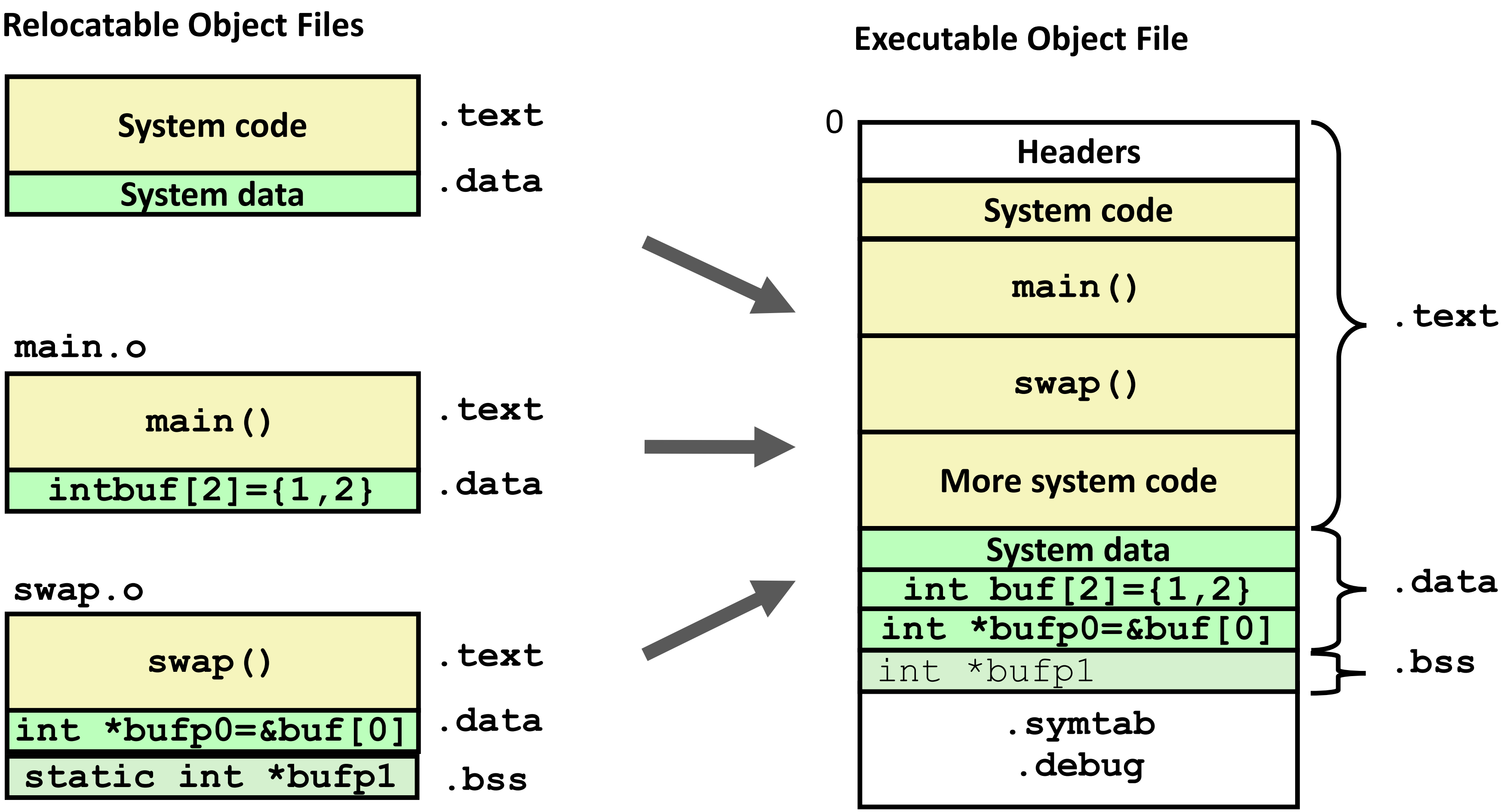
```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

Global

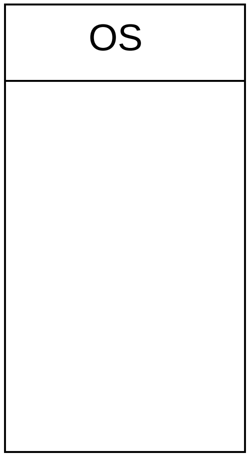
Linker knows nothing of temp

swap.c

Relocating Code and Data



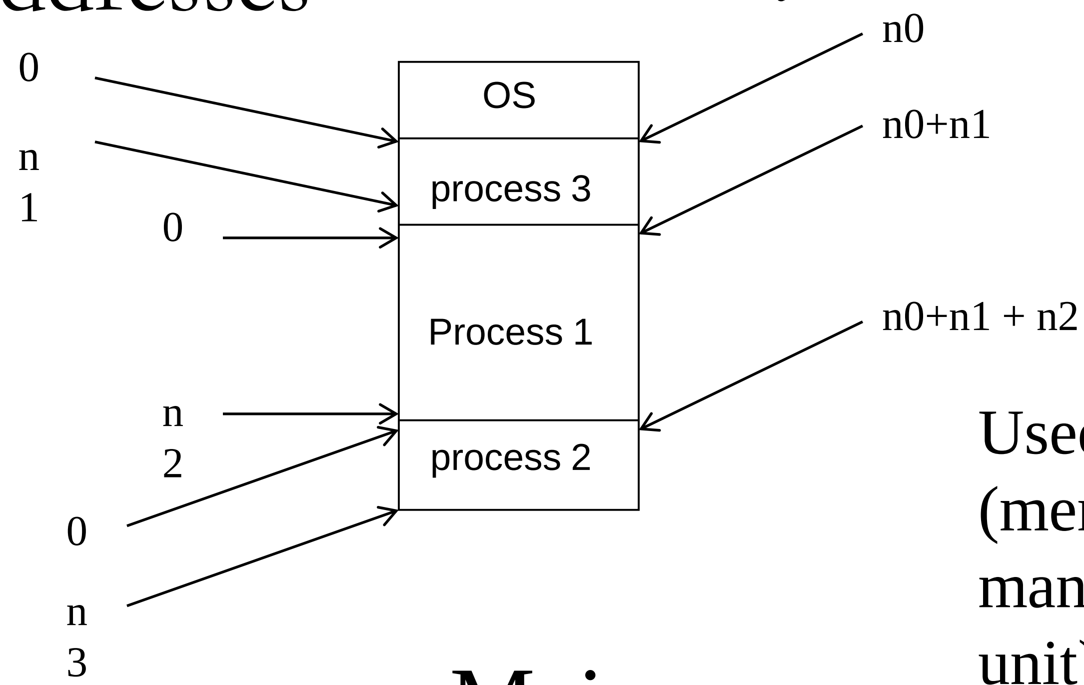
- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory



Contiguous Allocation

Logical Addresses

Physical Addresses



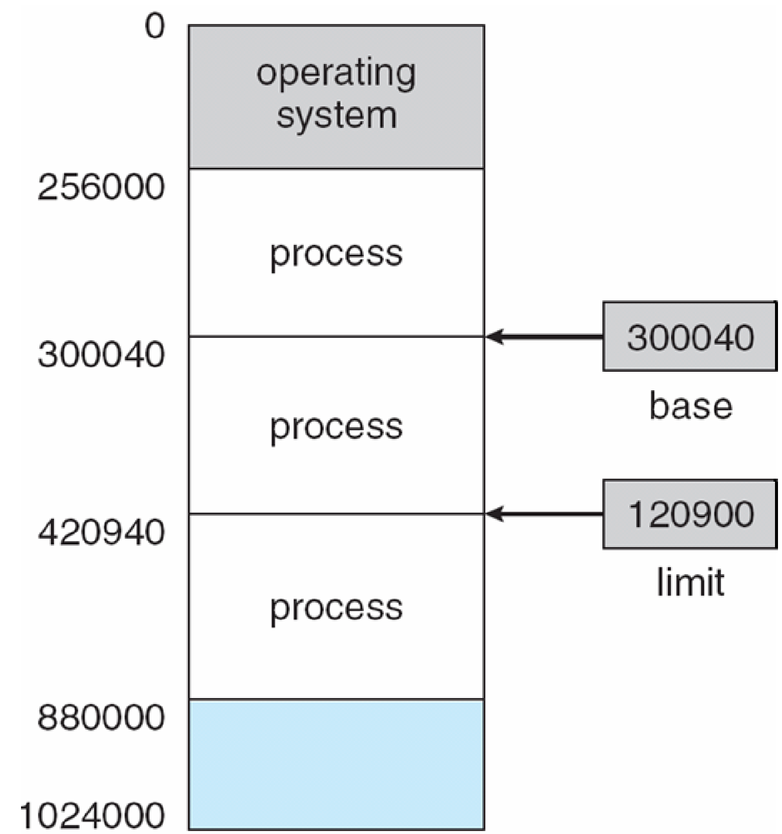
Used in code, by CPU

Used by MMU (memory management unit)

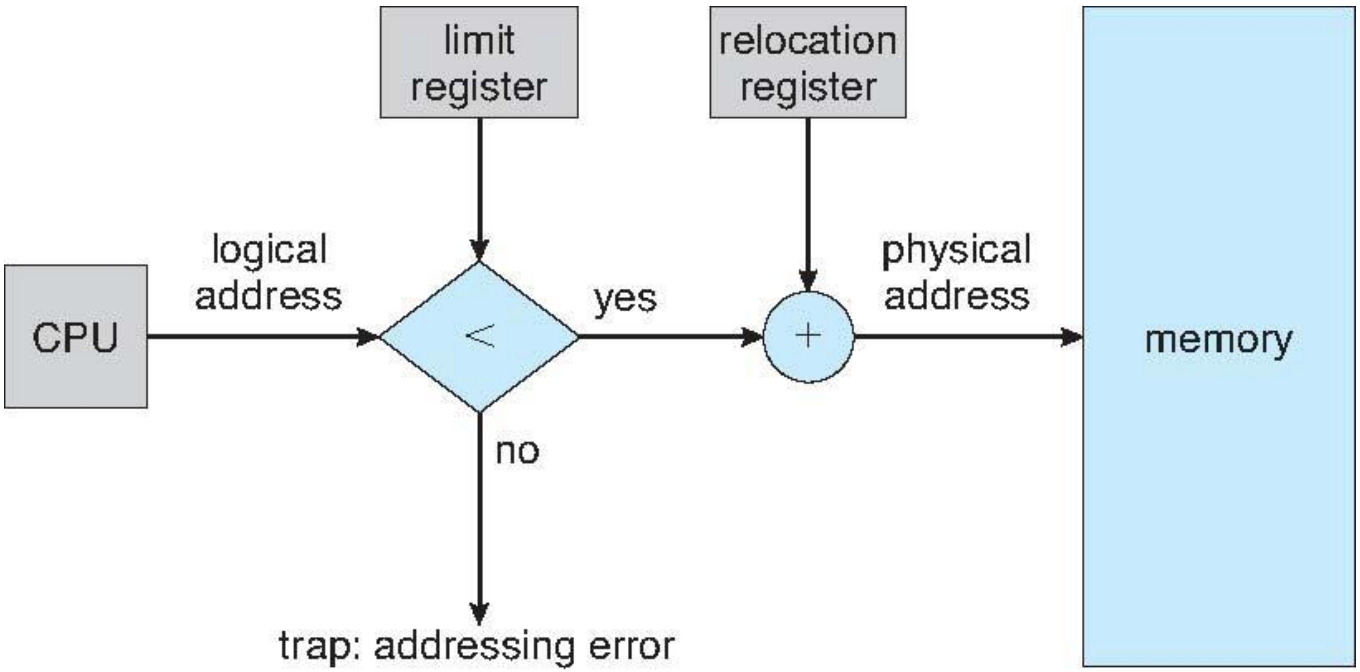
Main Memory

Hardware Support

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

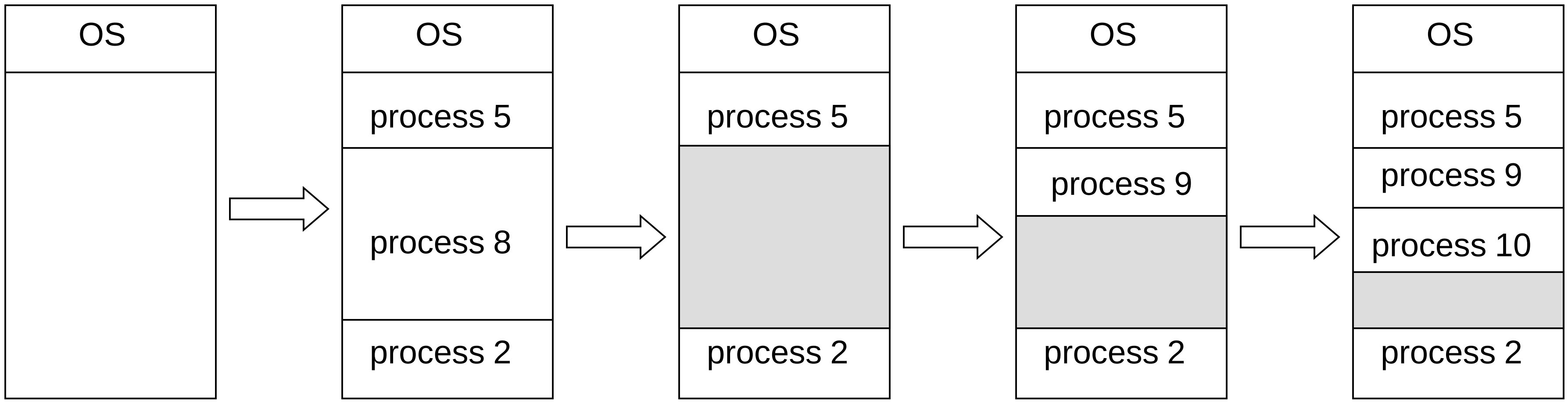


Hardware Support



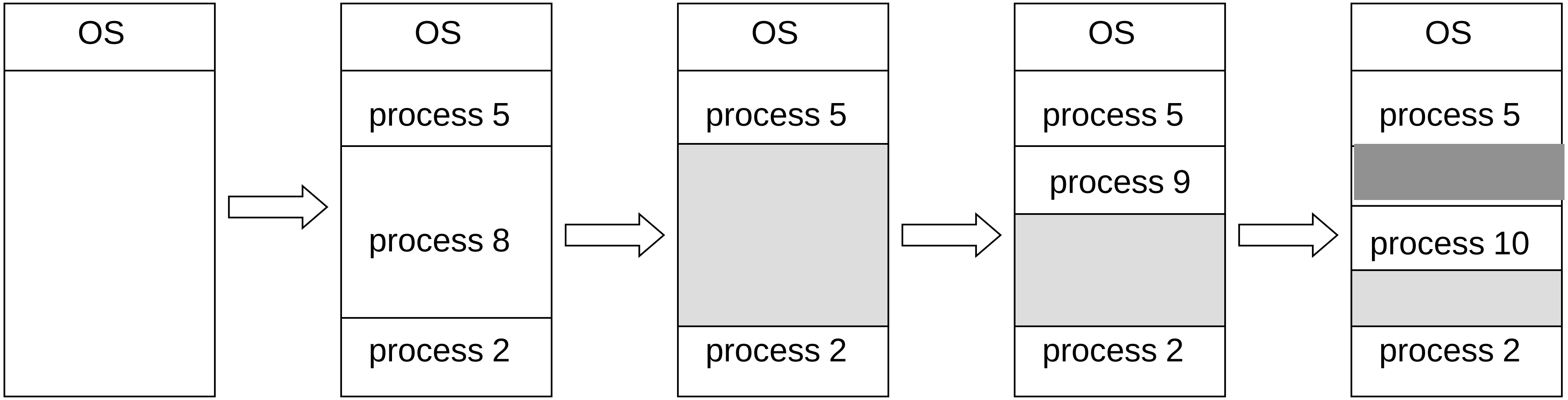
Holes

- Hole – block of un-occupied contiguous memory space
- At the beginning, there is a single hole in the memory: the whole space for user processes



Holes

- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  a) allocated partitions   b) free partitions (holes)
- Holes of various sizes may be generated later on and are scattered throughout memory



Contiguous Allocation Policies

How to satisfy a request 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 also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Limitation: Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- Reduce external fragmentation by **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

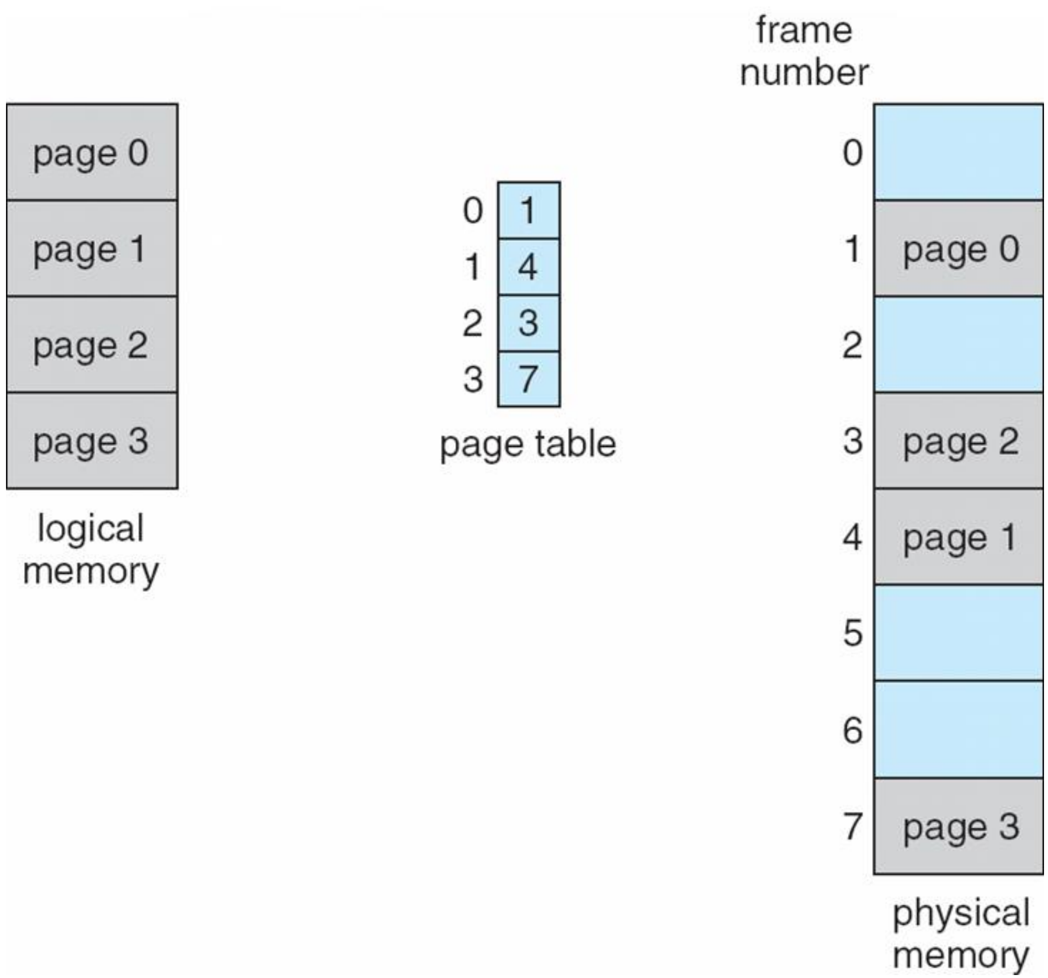
Paging – Memory management Strategy adopted by modern OSes

- Objective:
  - Logical address space of a process remains contiguous but the physical address space of it needs not be contiguous
  - Process is allocated physical memory whenever the latter is available

Paging: Key Ideas

- Divide physical memory (user memory part) into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory space of a process into blocks of same size called **pages**
- Pages are mapped to frames one-by-one; process-specific page table records the mapping and facilitates the logical to physical address translation
- OS keeps track of free frames and allocates frames to new/swap-in processes

Paging Model and Page Table



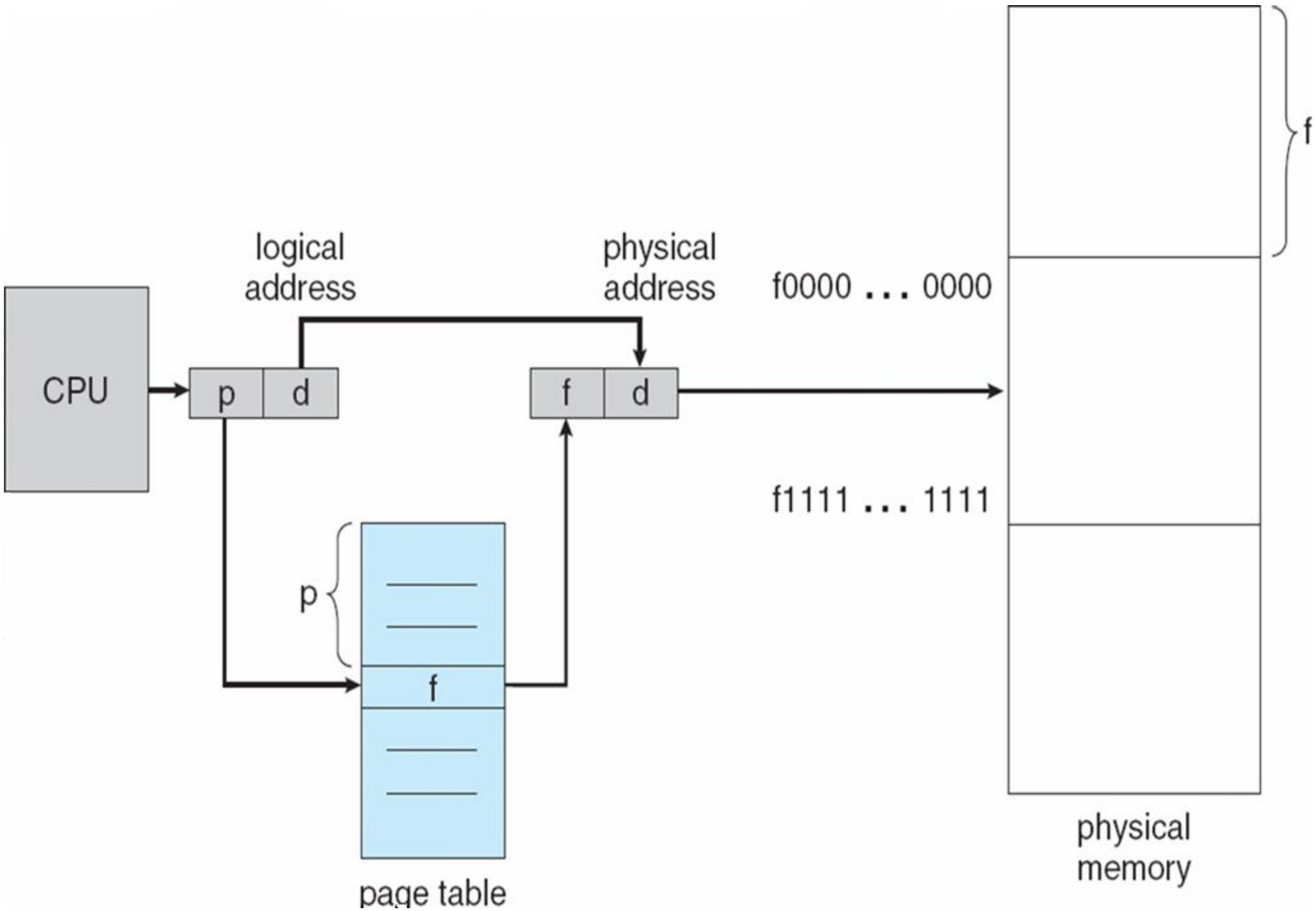
Address Translation Scheme

- Logical 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)** – combined with base address to define the physical memory address that is sent to the memory unit

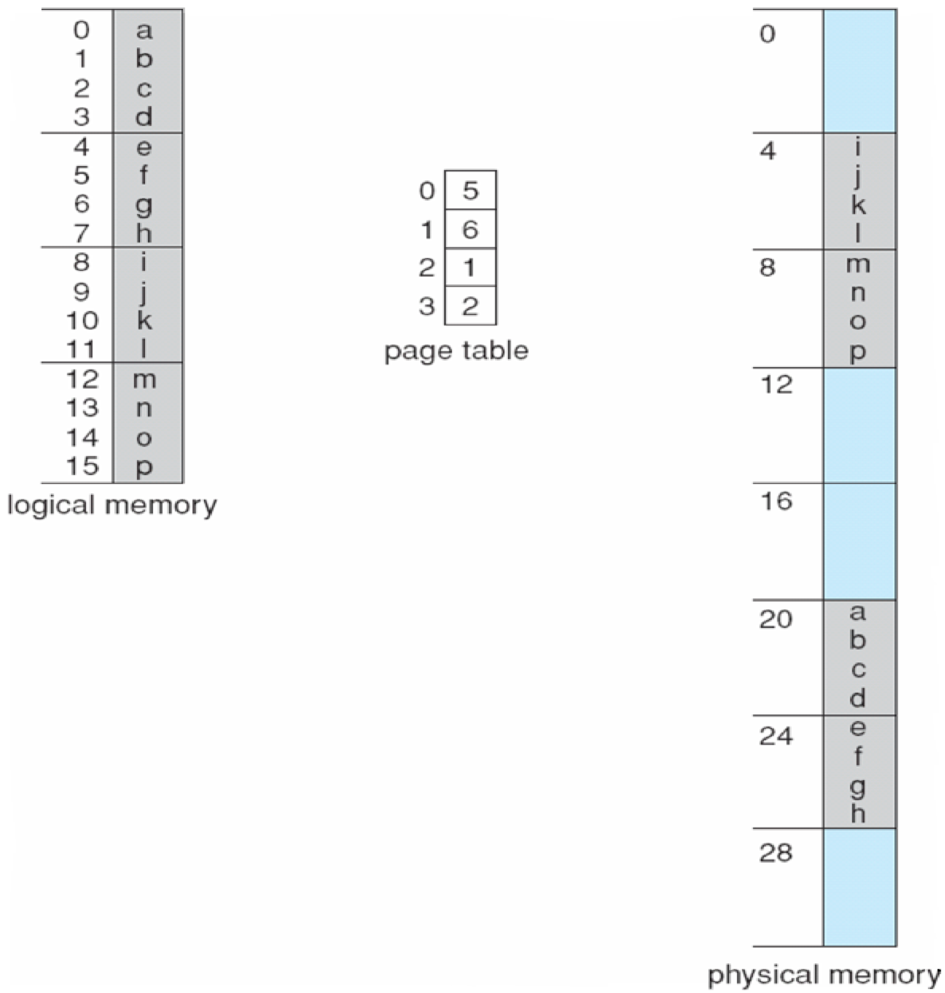
page number	page offset
p	d
$m - n$	n

  - For given logical address space 2^m, *page size* 2^n *and maximum number of pages per process* 2^{m-n}

Paging Hardware



Paging Example



32-byte memory and
4-byte pages

Efficiency Limitation & Solution

- Every data/instruction access requires two memory accesses: One for the page table and one for the data/instruction.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

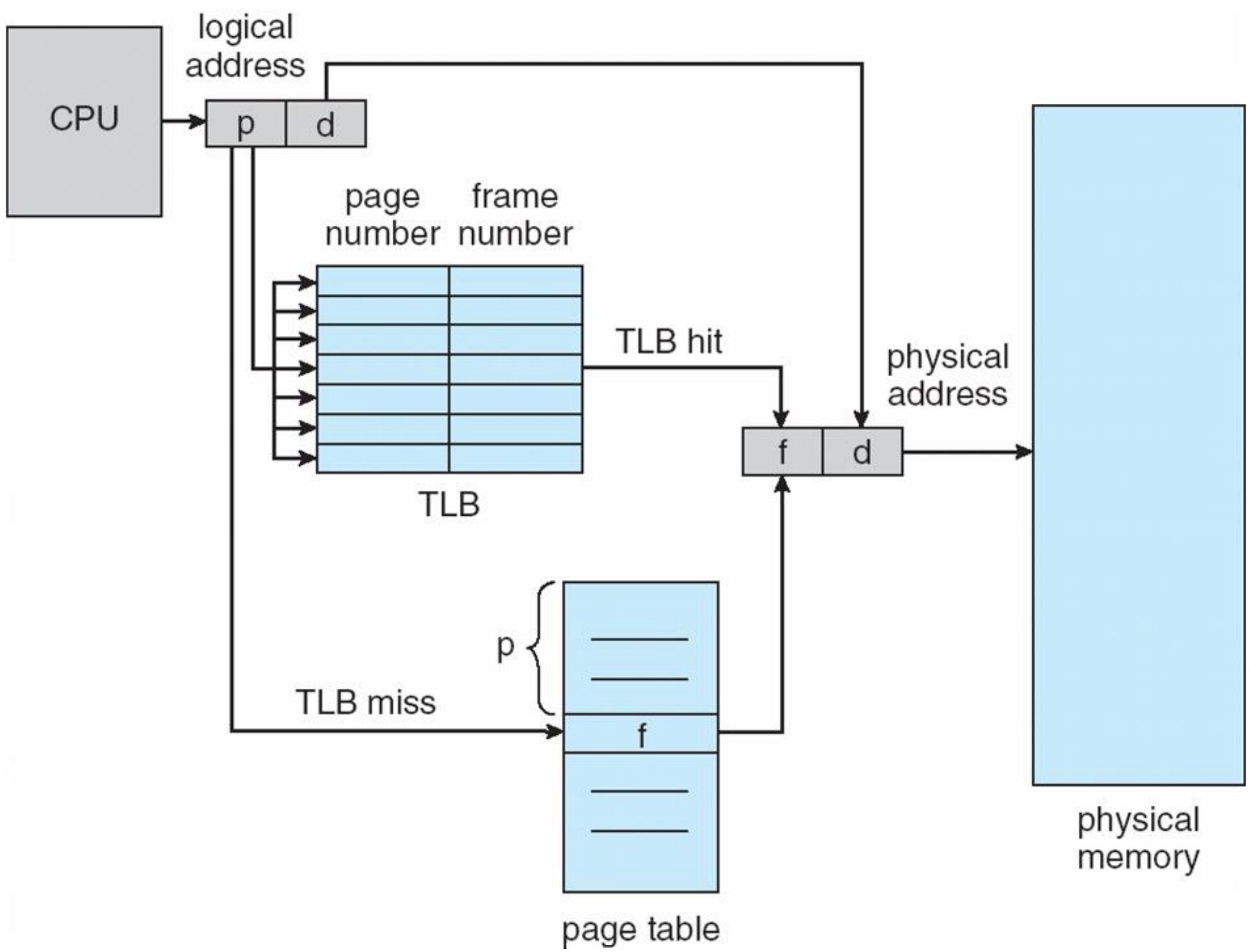
Associative Memory

- Associative memory – parallel search

Page #	Frame #

  - Input page number p
  - If p is in associative register, get frame # out

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε microsecond
- Assume memory cycle time is t microseconds
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$\begin{aligned} EAT &= (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha) \\ &= 2t + \varepsilon - \alpha t \end{aligned}$$

TLB Hardware is shared by multiple processes

- Problem: when a process is swapped out and a new process is swapped in, the content of the TLB becomes outdated

- Solutions:
  - flush the TLB when process switch; or
  - store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Implementation of Page Table

- Page table is kept in main memory (kernel space)
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- Memory protection implemented by
  - PTLR specifies the length of page table (the number of pages for a process)
  - If the page number of an address >= the value of PTLR, the logical address is invalid

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space