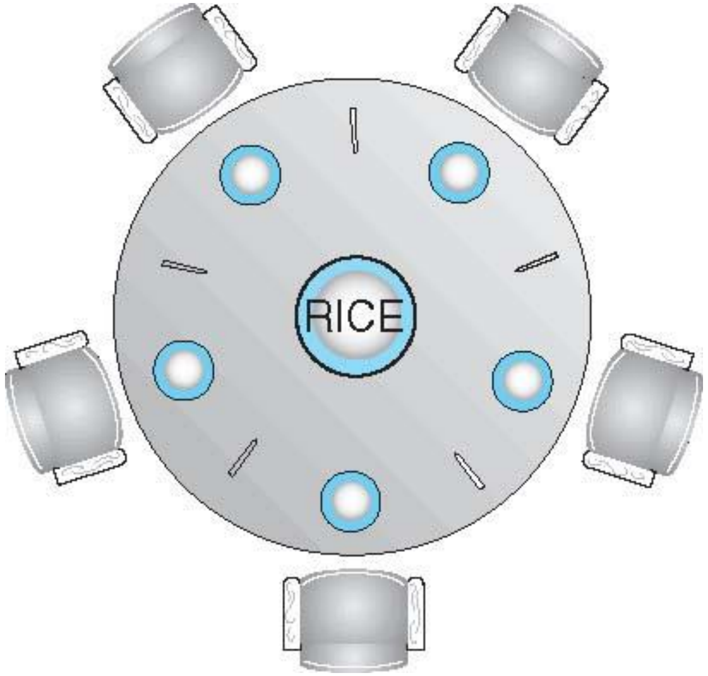
Dining-Philosophers Problem

- Philosophers spend their lives alternating between **thinking** and **eating**
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data analogy:
    - Each philosopher is a thread
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
     wait (chopstick[ (i + 1) % 5] );

     //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

     //  think

} while (TRUE);
```

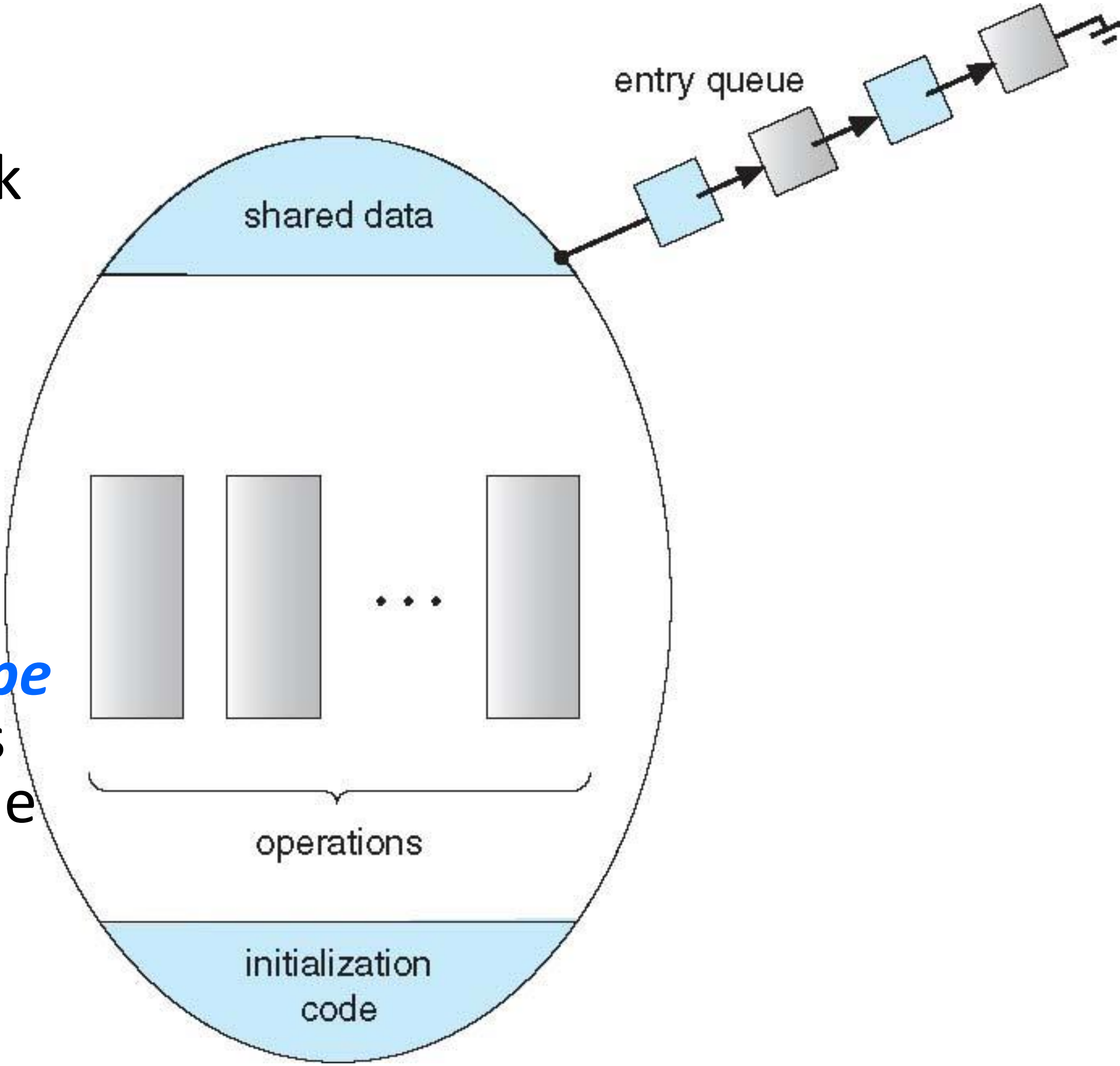
- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table of 5 chopsticks.
  - Use an asymmetric solution
    - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
    - An even-numbered philosopher picks up first the right chopstick and then the left chopstick.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section) → we may use **monitors** to implement this method.

5.8 Monitors

- The dining philosophers deadlock may be solved using monitors.
- A monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor is an ***abstract data type*** *(i.e. an **object**)*, internal variables only accessible by code within the procedure
- Only one process may be **active** within the monitor at a time.



5.8 Monitors

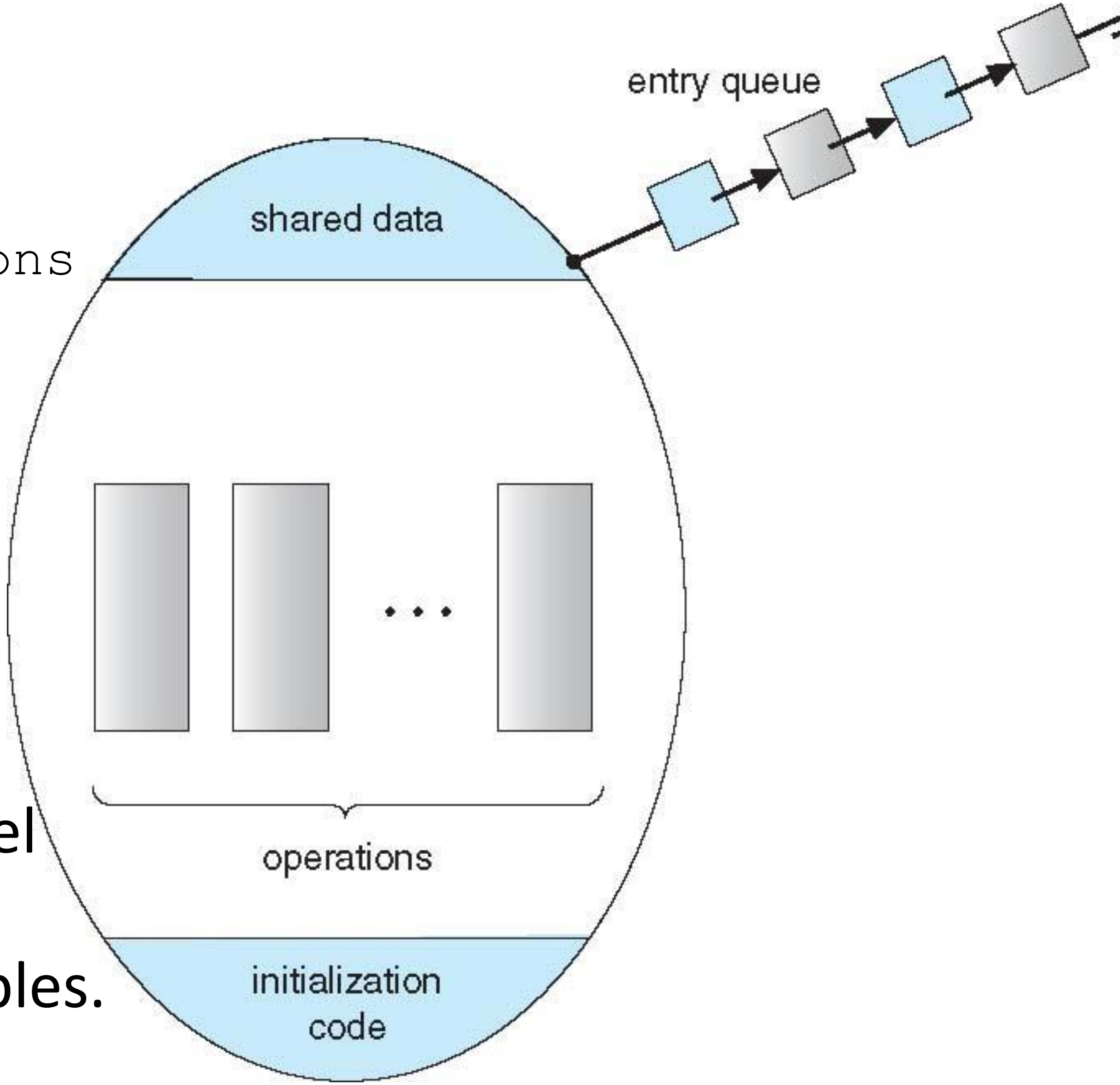
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

 Initialization code (…) { … }

}
```

- But not powerful enough to model some synchronization schemes
- Thus we may use condition variables.

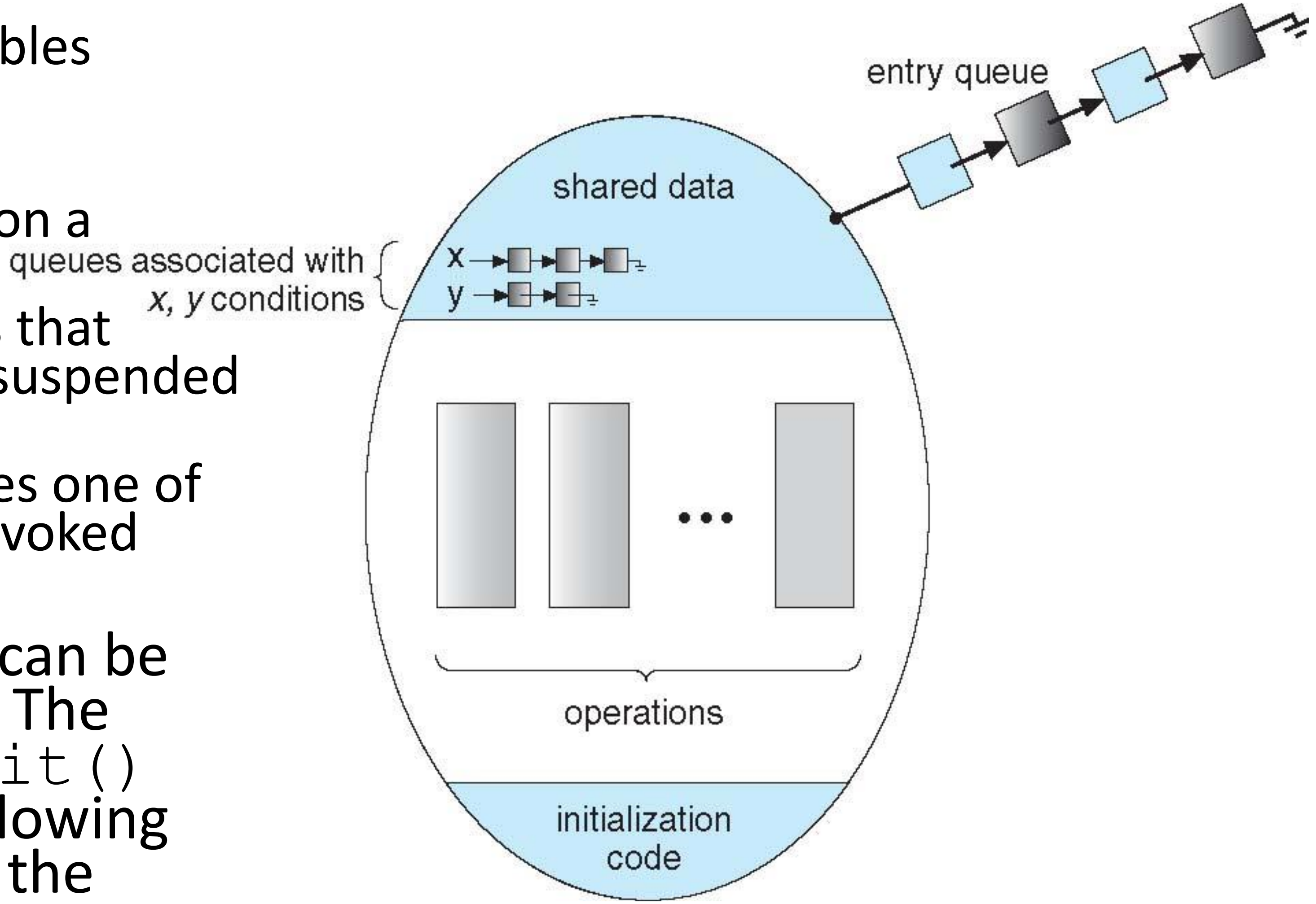


Condition Variables

- Condition variables are variables declared inside a monitor:

  `condition x, y;`

- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
- Only one thread/process can be active inside the monitor. The process that issues `x.wait()` becomes inactive, thus allowing others to be active inside the monitor.



Condition Variables – cont.

- Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore:
  - Unlike semaphores, if there are no threads/processes that are waiting on the condition variable `x`, then calling `x.signal()` does not affect the condition variable.
    - If you recall calling `signal(my_semaphore)` increments the semaphore whether there is someone waiting on it or not.

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q was suspended in `x.wait()`, what should happen next?
    - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
    - **Signal and wait** – P waits until Q either leaves the monitor or Q blocks waiting on another condition (i.e. immediate effect).
    - **Signal and continue** – P continues till it leaves the monitor or blocks waiting on another condition, and then Q may become active (i.e. deferred effect).
    - Both have pros and cons – language implementer can decide

Monitor solution to dining philosophers

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5] ;
  condition cond[5];

  void pickup (int i) {
         state[i] = HUNGRY;
         test_and_signal(i); /* if successful: my state becomes EATING + signal
                              myself which is wasted cause I am not waiting*/
         if (state[i] != EATING) cond[i].wait(); /* test(i) did not succeed */
  }

  void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test_and_signal((i + 4) % 5);
    test_and_signal((i + 1) % 5);
  }
```

Monitor solution to Dining Philosophers (Cont.)

```
  void test_and_signal(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ){
      state[i] = EATING;
      cond[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
  }
} /* end of monitor */
```

Monitor solution to dining philosophers (Cont.)

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

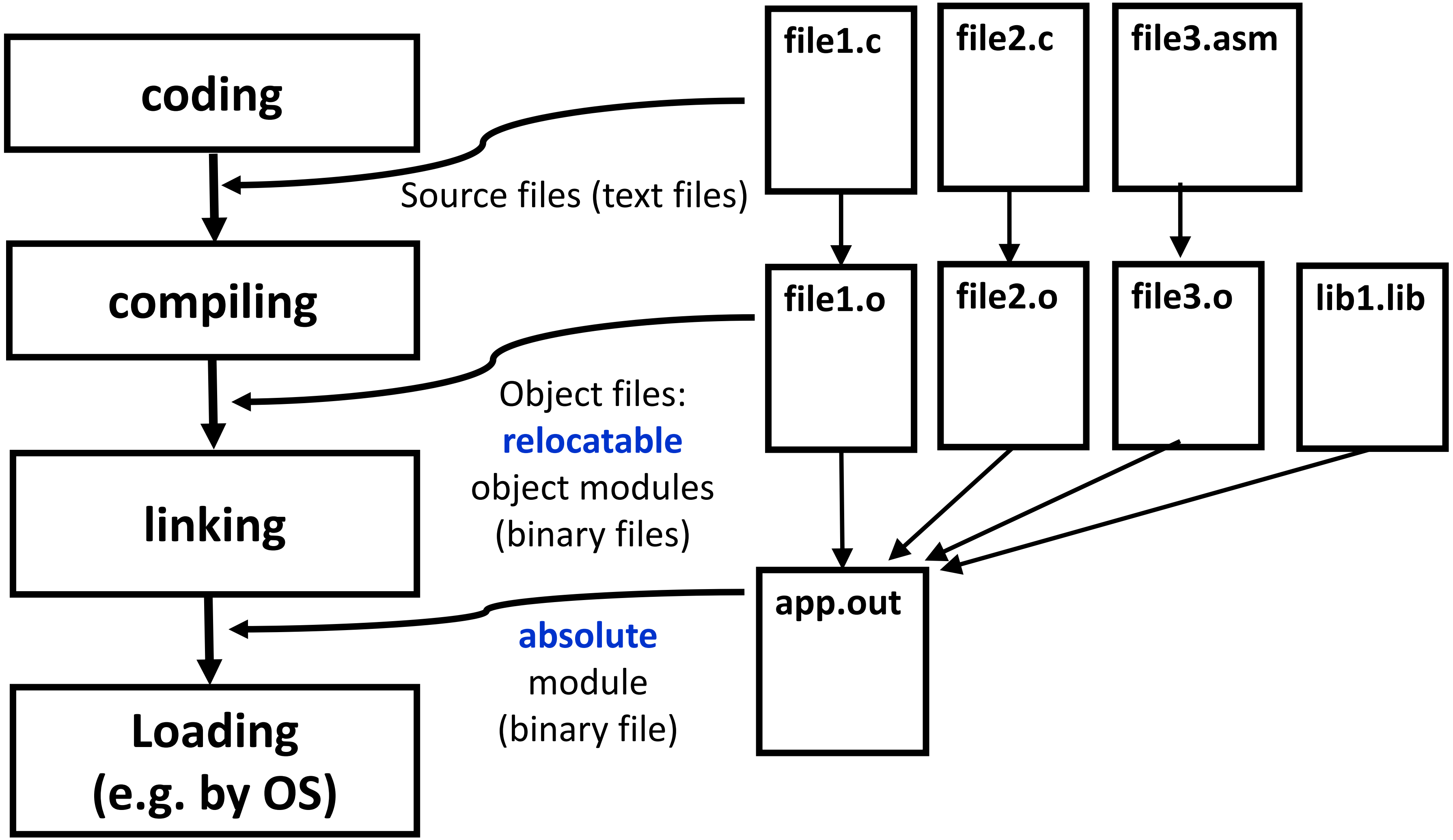
```
DiningPhilosophers.pickup(i);
EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Lecture 6: Memory Management

- Program translation
- Program relocation
- H/W support for program relocation
- Memory partitioning
- Memory swapping
- Virtual memory
  - Memory segmentation
  - Memory paging
- Virtual Memory management
  - Static virtual memory management and demand paging
  - Dynamic virtual memory management

Program translation



coding

compiling

linking

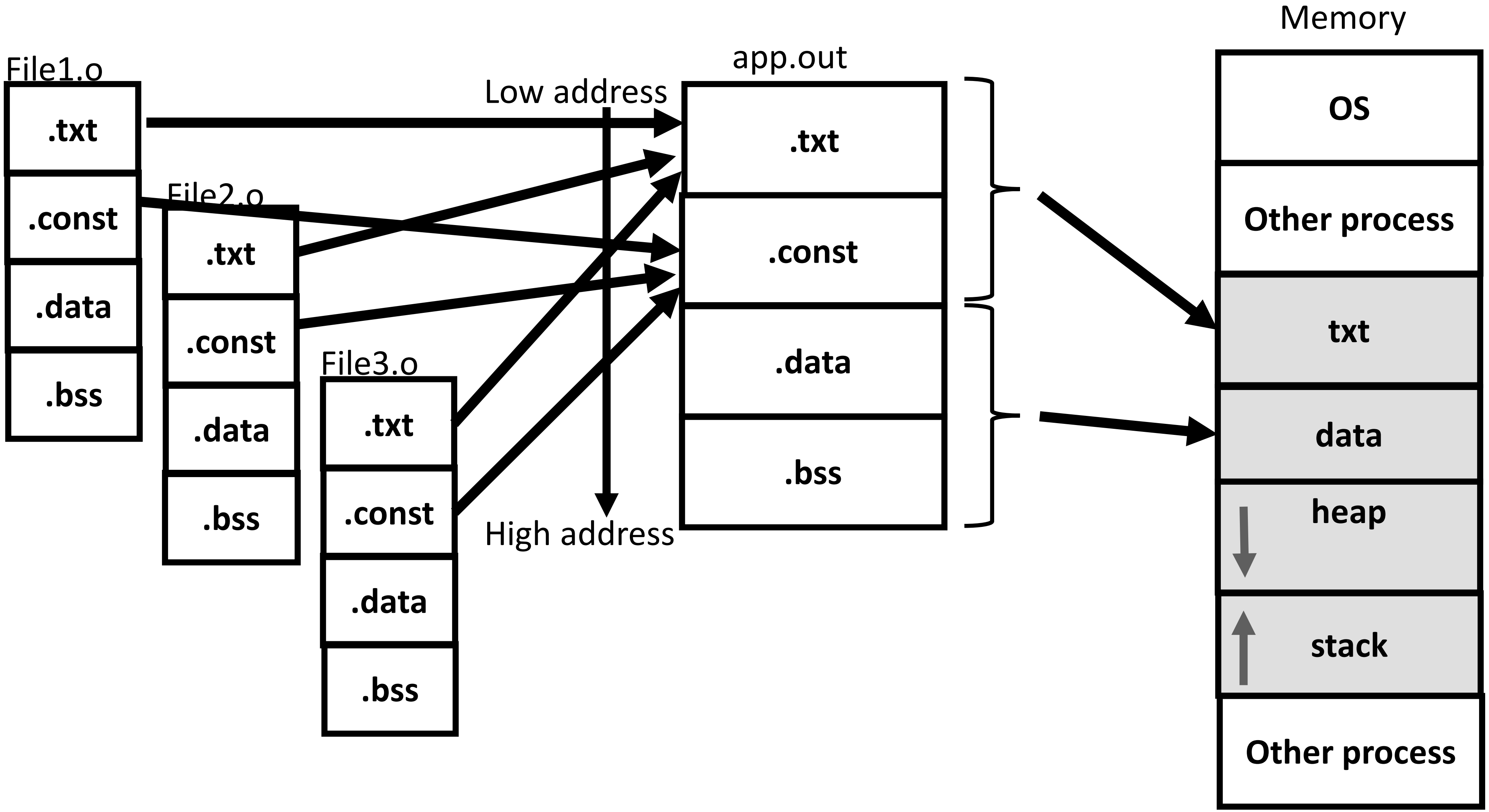
Loading
(e.g. by OS)

Source files (text files)

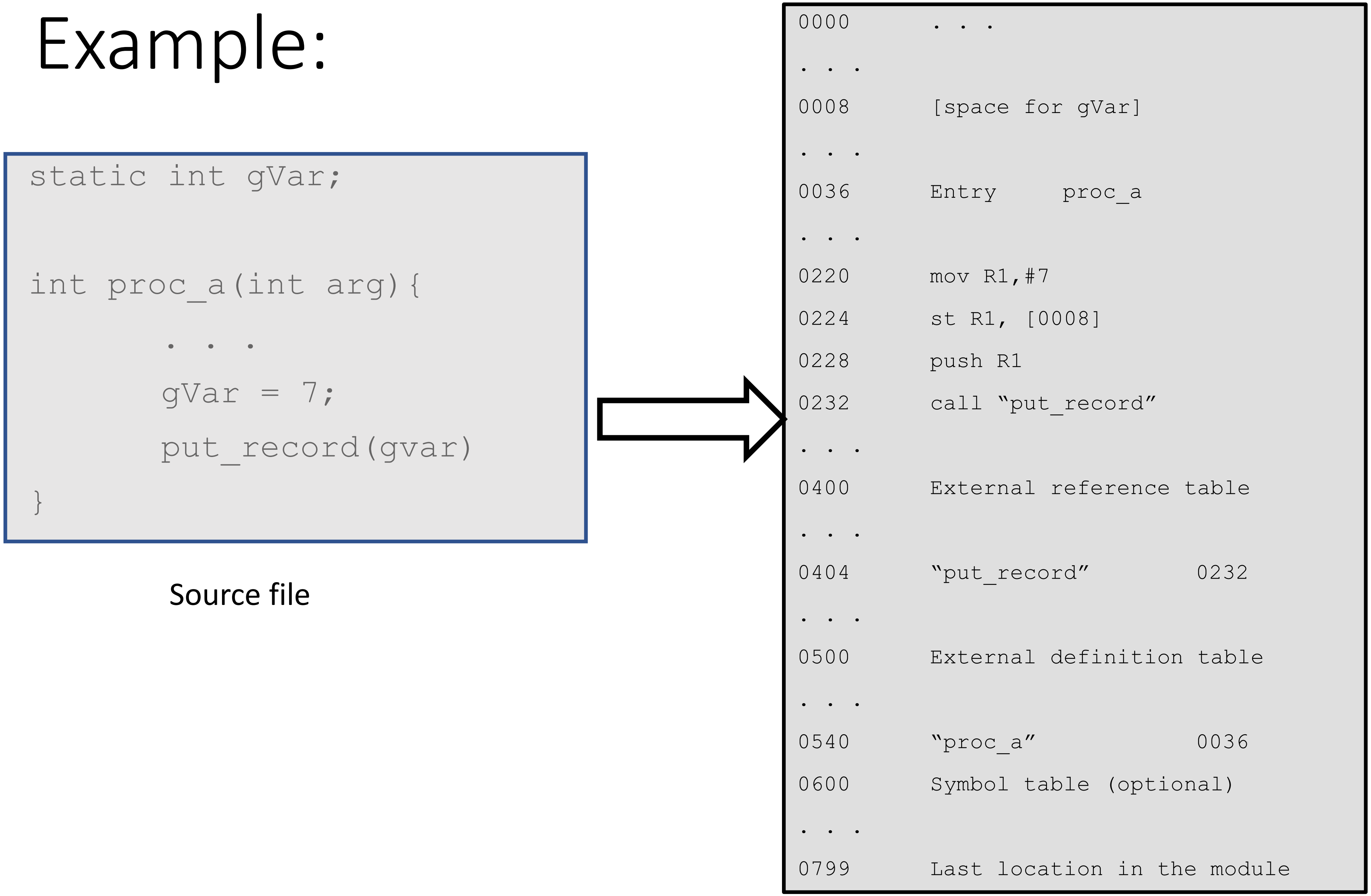
Object files:
relocatable
object modules
(binary files)

absolute
module
(binary file)

Program translation – linking + loading



Example:



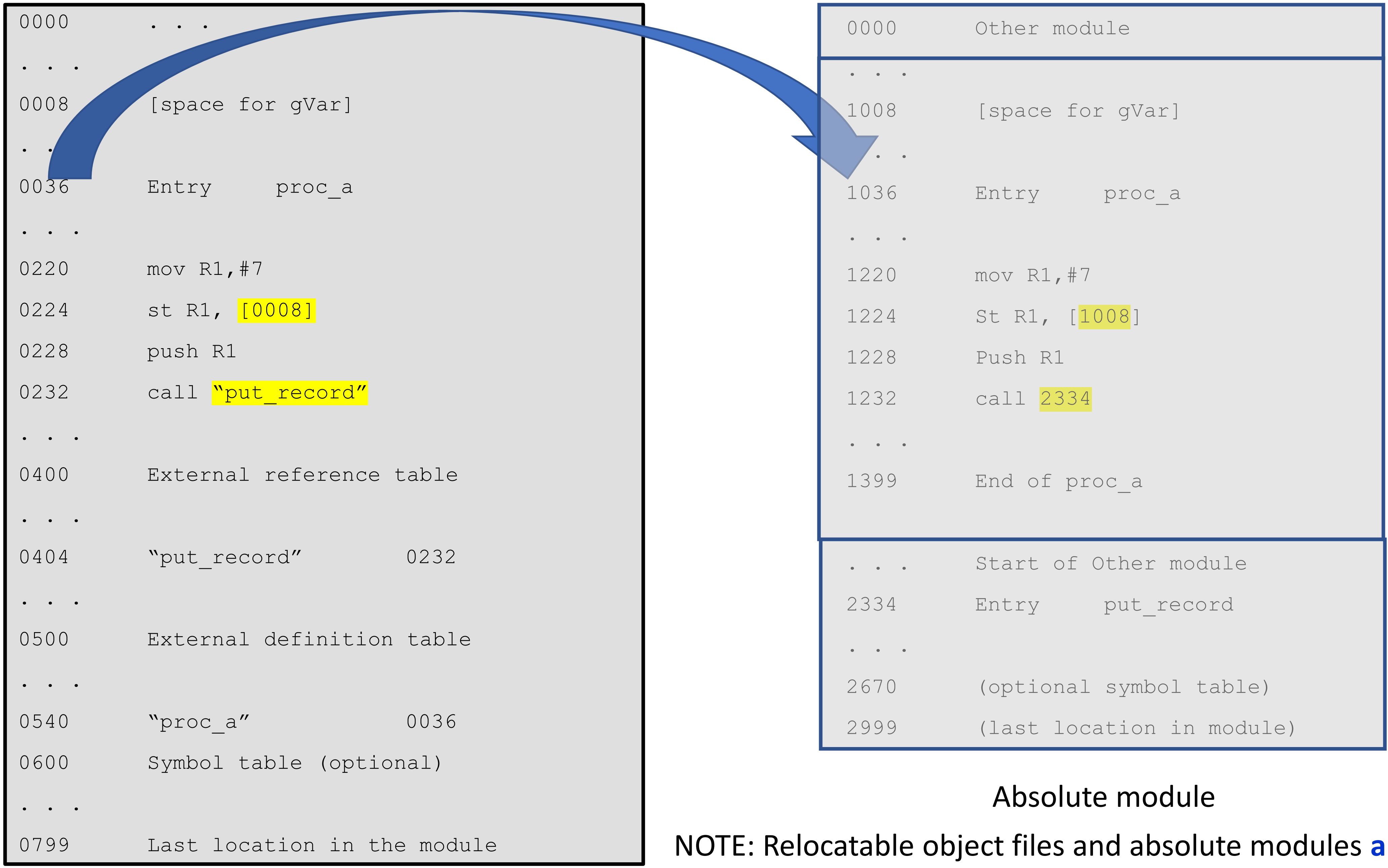
```
static int gVar;

int proc_a(int arg){
        . . .
        gVar = 7;
        put_record(gvar)
}
```

Source file

```
0000        . . .
. . .
0008        [space for gVar]
. . .
0036        Entry      proc_a
. . .
0220        mov R1,#7
0224        st R1, [0008]
0228        push R1
0232        call "put_record"
. . .
0400        External reference table
. . .
0404        "put_record"        0232
. . .
0500        External definition table
. . .
0540        "proc_a"            0036
0600        Symbol table (optional)
. . .
0799        Last location in the module
```

Relocatable object file



```
0000      . . .
. . .
0008      [space for gVar]
. . .
0036      Entry     proc_a
. . .
0220      mov R1,#7
0224      st R1, [0008]
0228      push R1
0232      call "put_record"
. . .
0400      External reference table
. . .
0404      "put_record"       0232
. . .
0500      External definition table
. . .
0540      "proc_a"           0036
0600      Symbol table (optional)
. . .
0799      Last location in the module
```

Relocatable object file

```
0000      Other module
. . .
1008      [space for gVar]
. . .
1036      Entry     proc_a
. . .
1220      mov R1,#7
1224      St R1, [1008]
1228      Push R1
1232      call 2334
. . .
1399      End of proc_a
. . .     Start of Other module
2334      Entry     put_record
. . .
2670      (optional symbol table)
2999      (last location in module)
```

Absolute module

NOTE: Relocatable object files and absolute modules **are binary files** (not text) – Above is only a representation of the actual binary files

Program translation - Relocation issues

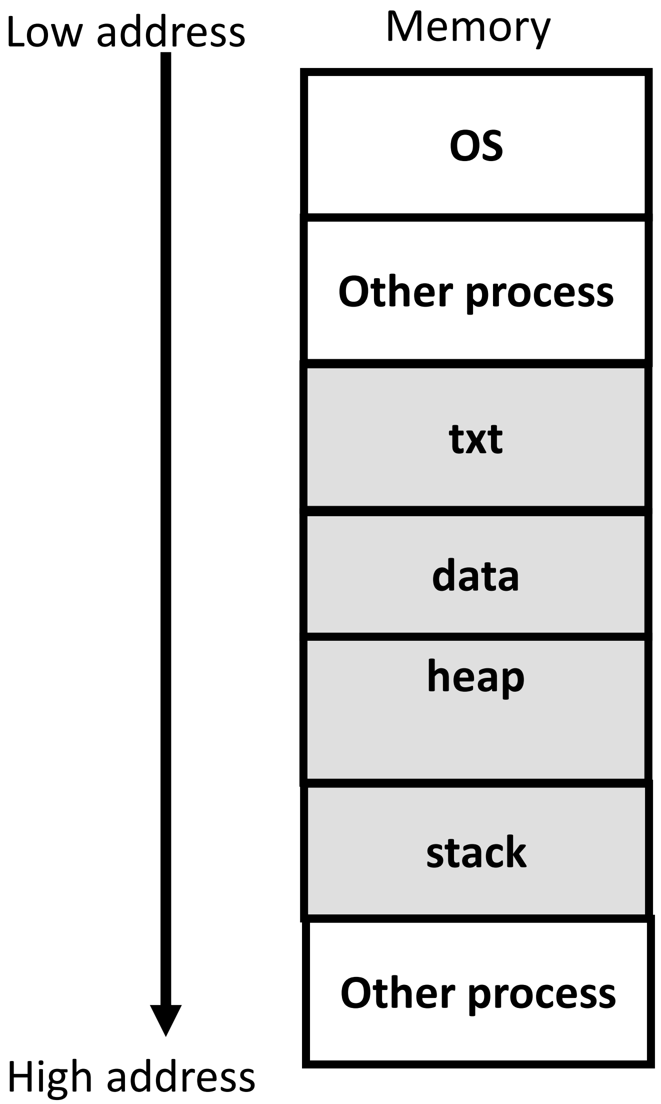
- Referencing static variables:
  - gVar was located at location 0008, but now it is located at 1008. What happens to instructions trying to access gVar ?
  - Clearly, the linker now needs to get to every instruction that accesses gVar and change its reference to location 1008. There are multiple ways of doing this:
    - Linker finds every load or store instruction trying to access the old memory location and changes the address to the new location.
    - Linker uses the optional symbol table, where each relocatable symbol is readily listed and it can easily pin point which instruction is accessing gVar
- Absolute branches
  - ***Relative branches*** are not a problem since they specify an offset from the current program counter location.
  - Linker needs to modify addresses referenced in ***absolute branches*** (or calls) just as it did for static variable, e.g. the call to “put_record” gets replaced with the absolute (aka logical) address of “put_record”.

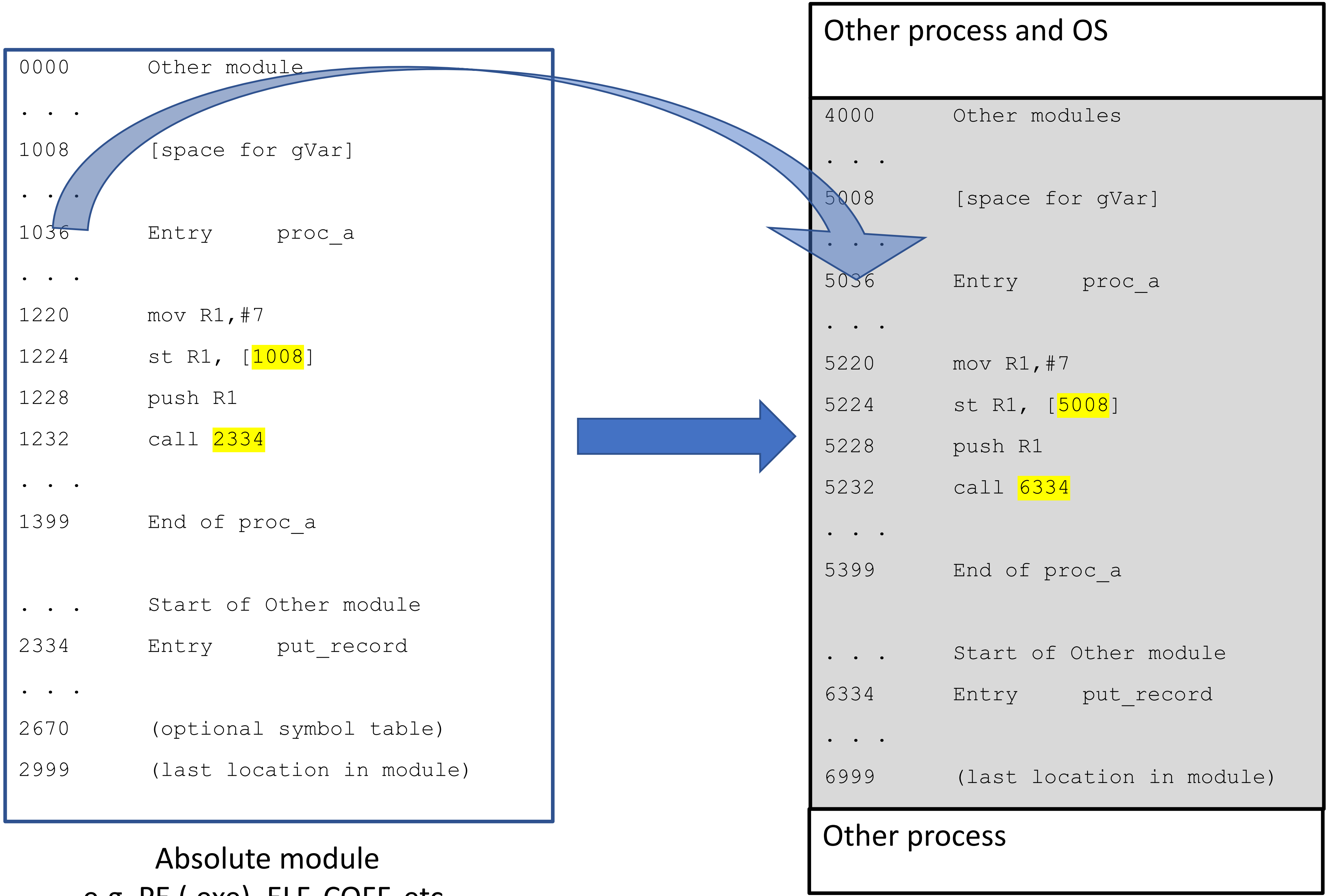
Program translation – file formats

- Source files:
  - Formatted as text files
  - e.g. main.c
- Relocatable object modules and absolute modules:
  - Contain **binary machine instructions** + **information used by linker or loader**
  - Formatted as binary files.
  - Common formats are:
    - COFF: common object file format – commonly used in embedded systems (e.g. microcontrollers: program is loaded into flash memory)
    - PE: portable executable format – used in Windows, extension is .exe
    - ELF: Executable and linkable format – used in Unix/Linux systems.

Program translation – loading

- Memory may be partitioned to accommodate multiple running processes.
- A program is loaded into an area of memory that does not necessarily start at address 0x00.
- Thus more relocation needs to take place at load time.





```
0000      Other module
. . .
1008      [space for gVar]
. . .
1036      Entry     proc_a
. . .
1220      mov R1,#7
1224      st R1, [1008]
1228      push R1
1232      call 2334
. . .
1399      End of proc_a

. . .     Start of Other module
2334      Entry     put_record
. . .
2670      (optional symbol table)
2999      (last location in module)
```

Absolute module
e.g. PE (.exe), ELF, COFF, etc.

Other process and OS

```
4000      Other modules
. . .
5008      [space for gVar]
. . .
5036      Entry     proc_a
. . .
5220      mov R1,#7
5224      st R1, [5008]
5228      push R1
5232      call 6334
. . .
5399      End of proc_a

. . .     Start of Other module
6334      Entry     put_record
. . .
6999      (last location in module)
```

Other process

MEMORY

Program translation – cont.

- The process of finding an address in the physical memory (physical address) and attaching it to the relocatable address in the absolute module (= logical address as in your text book) is referred to as **binding**. This binding takes place at **load time**.

- Similarly another binding takes place when attaching addresses in the relocatable object module into addresses in the absolute module. This binding takes place at **compile time**.

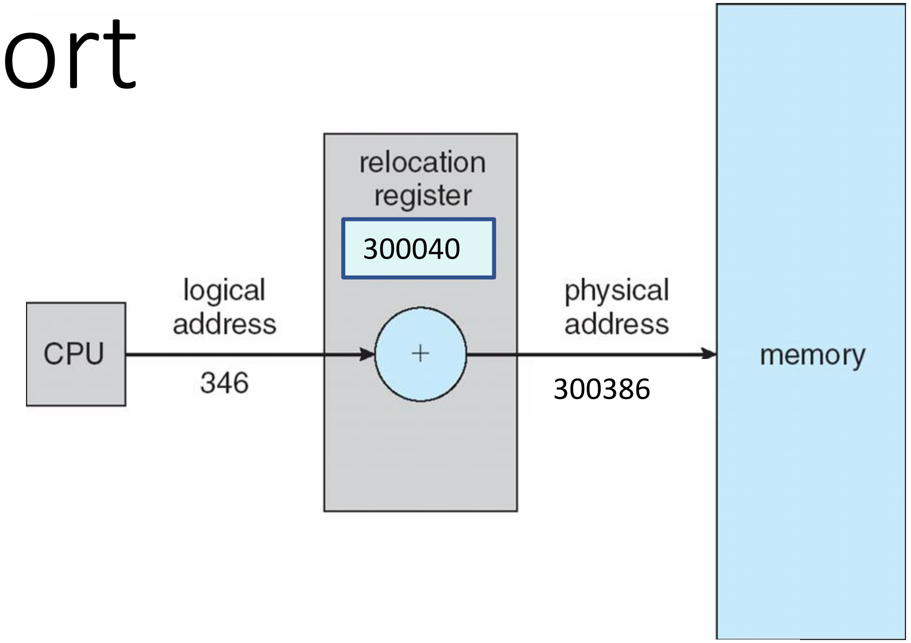
- Note that in many systems, it may be that the entire system only runs **one program** and the logical address may be the same as the physical address (e.g. a microcontroller), and thus there may be no need for load-time address binding.

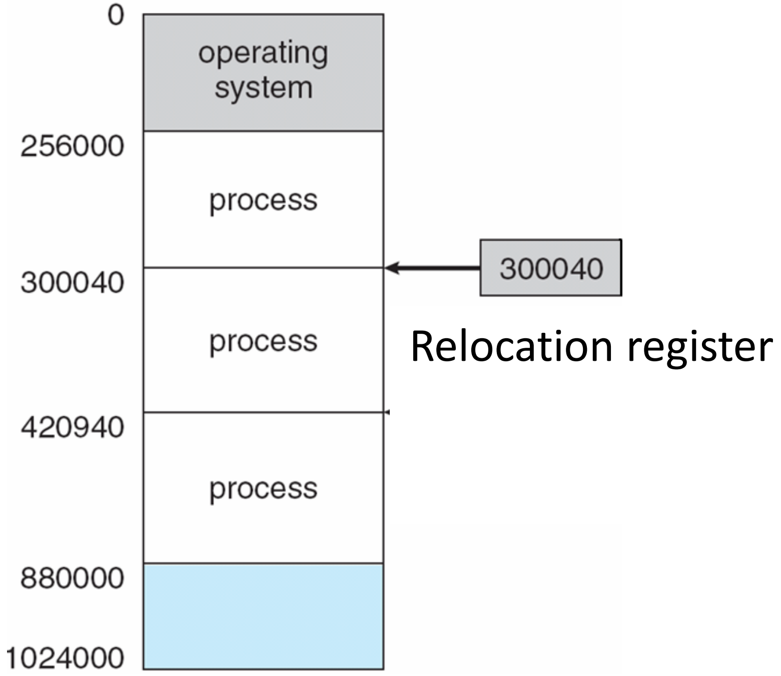
- User programs are designed using the logical address, i.e. the program thinks it is running alone in memory and thinks it starts at address 0.

Hardware relocation support

- A **relocation** register may be used to convert the logical address to a physical address. This register may be also referred to as a **base** register.

- Since a program's logical address starts at 0x0000 but needs to be loaded and executed at a different location in the physical memory, a base register may be summed to every logical address before that address is dispatched to the main memory -> thus it does the job of **binding** instead of the loader.
- Many of the older intel CPUs (e.g. 8088) supported 4 segment registers for: code, data, stack and extra segments.



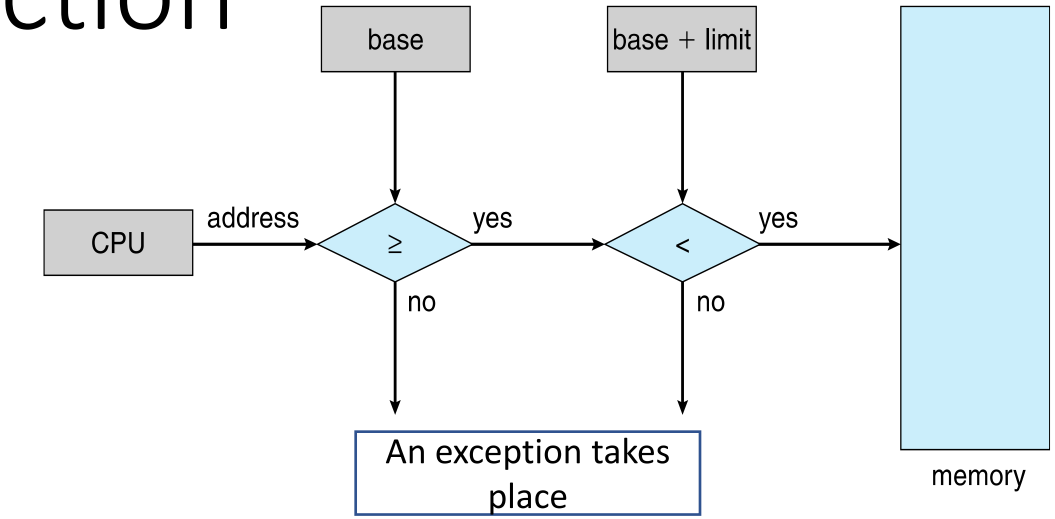


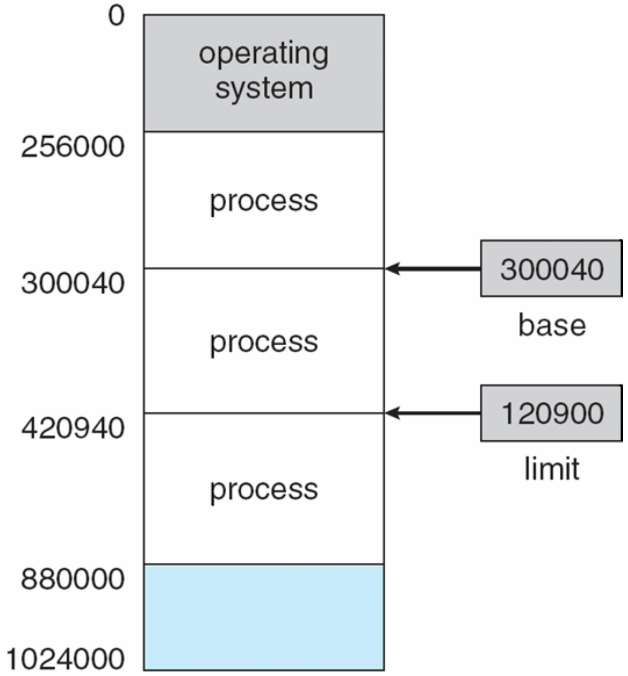
Hardware Memory protection

- CPU may check every physical memory access generated by a process to be sure it is between **base** and **limit** registers. This hardware thus performs **memory protection**.

- If a process attempts to access an address that is outside its limits, an **exception** takes place and the OS kernel is invoked.

- OS is responsible for setting the correct values in base and limit registers for each process prior to loading it. OS is also in charge of saving that information in the **PCB** during **context switching**.





Dynamic Linking

- **Static linking** – system libraries and program code are combined by the linker into the absolute module.
- **Dynamic linking** – Linking of some object modules (but possibly not all) is **deferred** until load time
  - Particularly useful in implementing **shared libraries** (uses .so extension for Linux libraries and .dll for Windows libraries).
  - If a shared library is used within your program, then the **external reference table** must be in the absolute module. That external reference table will help the loader find the symbols in the shared library and resolve their addresses, i.e. **late binding** (done at load time)

Memory allocation - partitioning

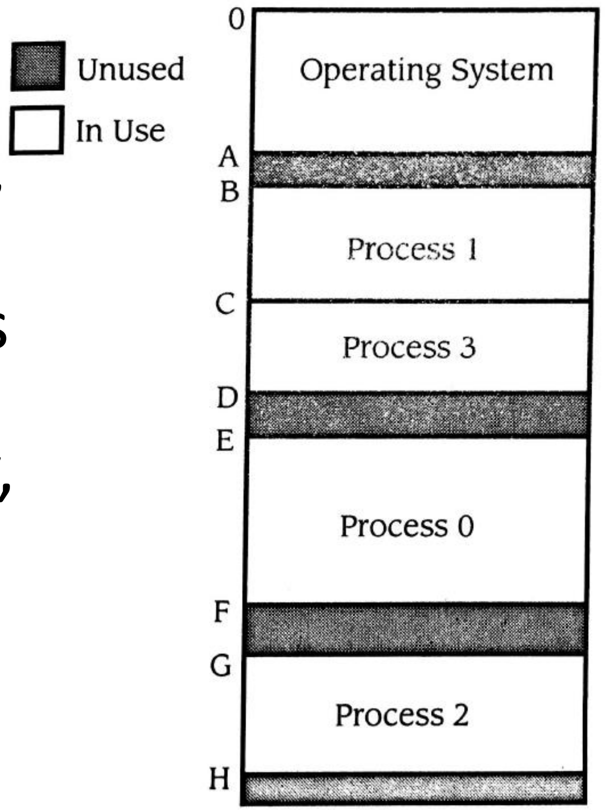
- Main memory must support both OS and user processes
- Limited resource: OS and multiple processes need to share the main memory. Need to allocate efficiently
- **Partitioning** is one of the earliest memory allocation strategies, and it uses **contiguous memory allocations**.
  - More advanced memory allocation strategies include:
    - Segmentation.
    - Paging.
- Main memory is divided into multiple **partitions**:
  - The first partition is reserved for resident operating system, usually held in low memory with the interrupt vectors
  - User processes are then held in high memory, where each process is contained within a single contiguous section of memory (partition)

Memory allocation - partitioning cont.

- Relocation registers are used to **load** each process’ absolute module into the allocated partition.
- Relocation registers are also used to **protect** user processes from each other, and from changing operating-system code and data.
  - Base register contains value of process’ smallest physical address
  - Limit register contains the address range – each logical address must be less than the limit register.
- As compared to load-time binding, relocation registers are advantageous because logical addresses may be mapped to physical addresses ***dynamically***, and thus a process may be **relocated** to a different partition by copying its memory content to the new location and modifying the base and limit registers.

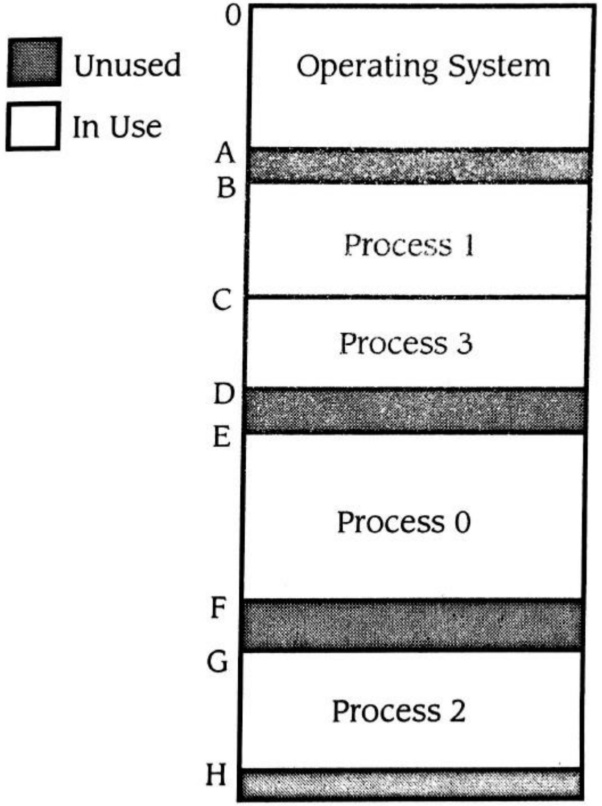
Fixed partition allocation

- Memory is divided into fixed sized partitions
  - Figure to the right shows 5 partitions: 0-B, B-C, C-E, E-G, G-end)
- Partitions are expected be of different sizes to accommodate processes of different sizes, but does not change size dynamically, i.e. sizes are fixed.
- If the process size is smaller than the partition it is residing in -> **internal fragmentation**.
- **Degree of multiprogramming** limited by number of partitions



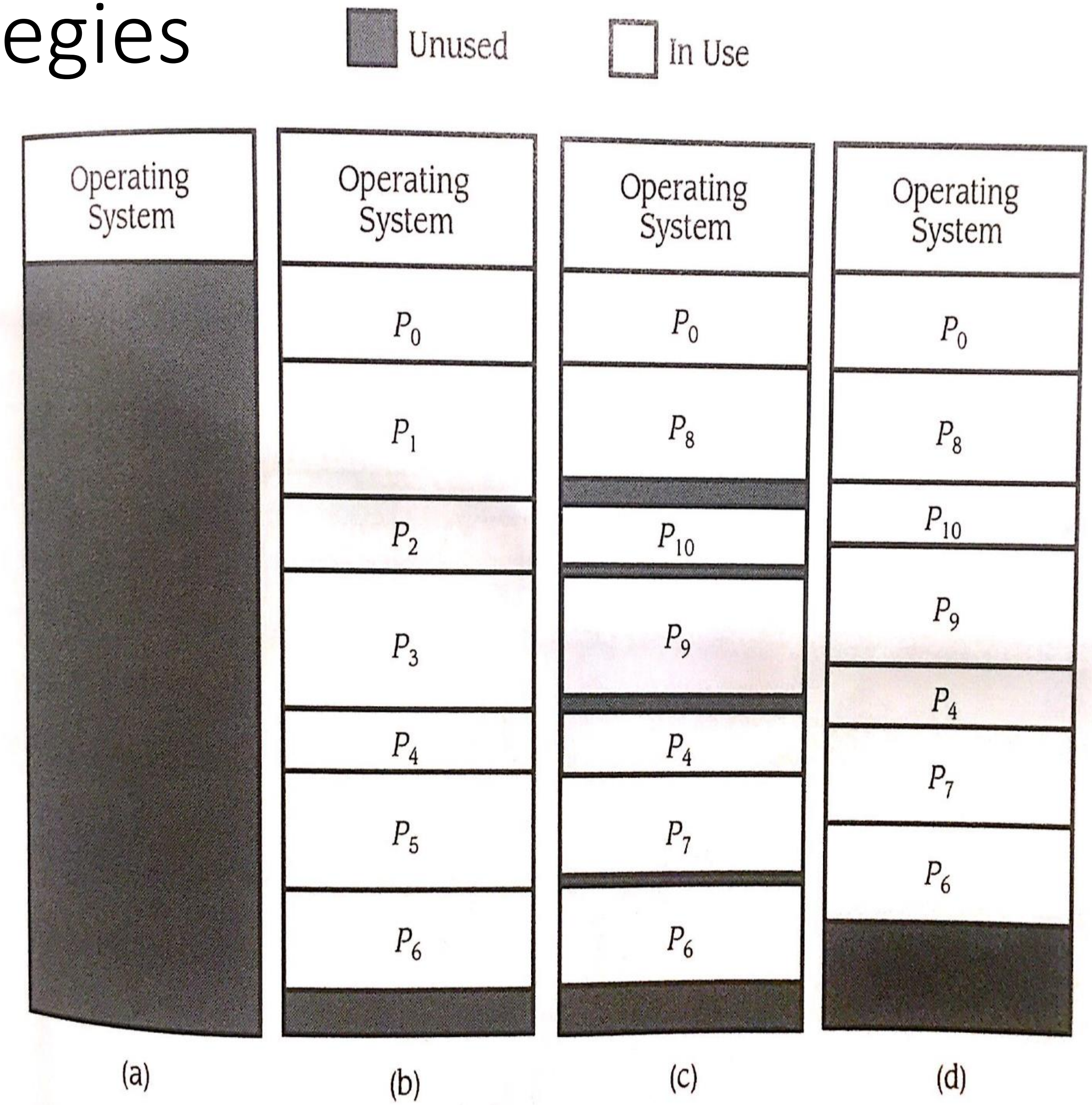
Fixed partition allocation – cont.

- If more processes than partitions, then **each partition will have a queue of waiting processes**. A process is allocated to a partition's Queue (i.e. FIFO) based on a strategy such as:
  - Best fit.
  - Queue balancing
- Alternatively, a **single queue** may be used,
- Fixed partition strategies suffer significantly from fragmentation, particularly since most systems start and end processes dynamically and it is thus hard to predict a reasonable set of fixed size partitions in advance.



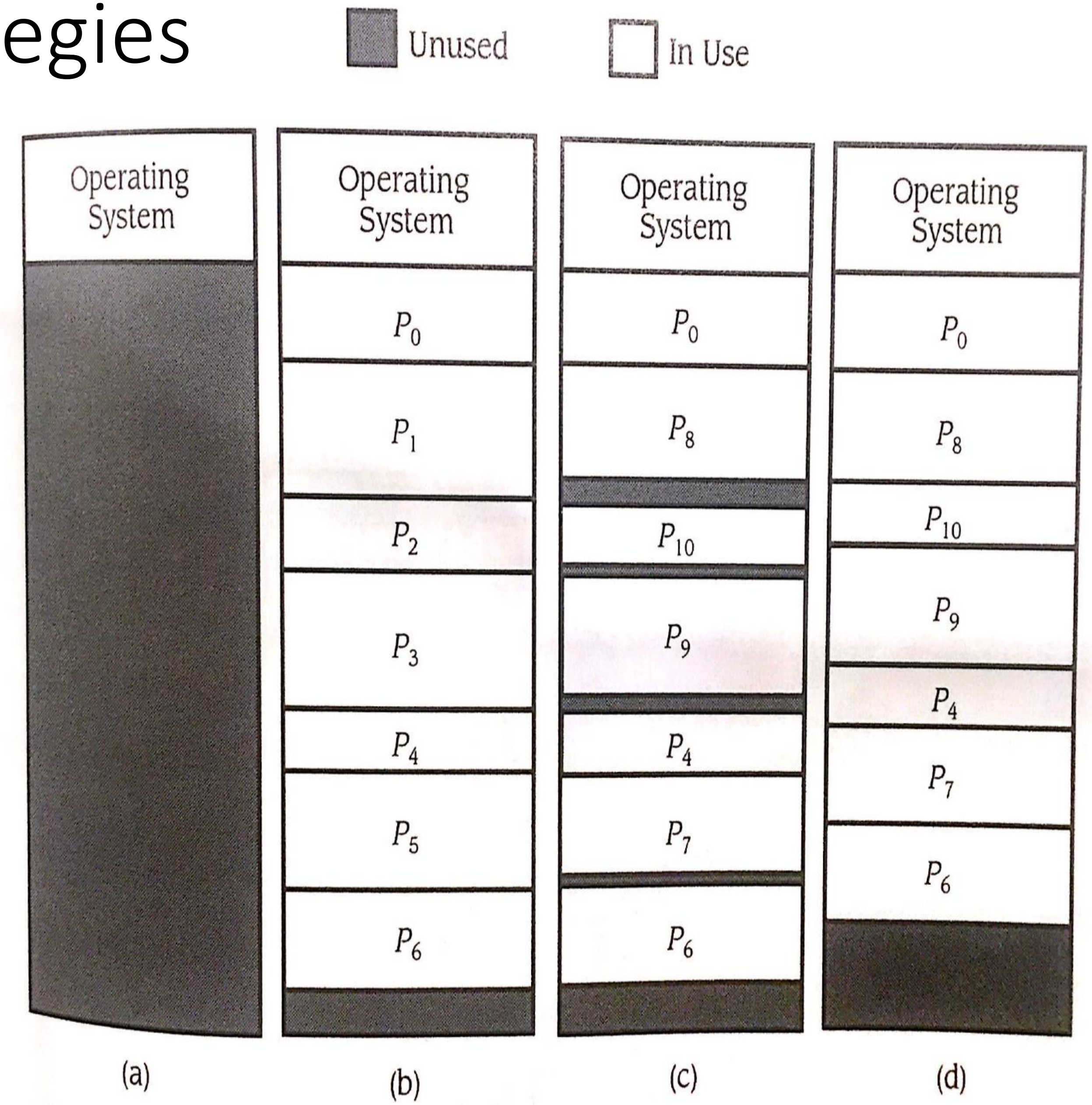
Variable partition strategies

- A partition's size may change dynamically according to what a loaded process needs
- Initially, there is no internal fragmentation loss, and only a small external frag. Loss (Fig. b).
- Over time processes exit and others are created and thus fragmentation holes appear (Fig. c). At this stage, the system favors smaller processes (in order to fit into available holes).
- Variable partitioning thus suffers from **external fragmentation**.
- **Periodically**, the system thus relocates the processes in order to **compact** multiple holes and form a larger fragment (Fig. d).



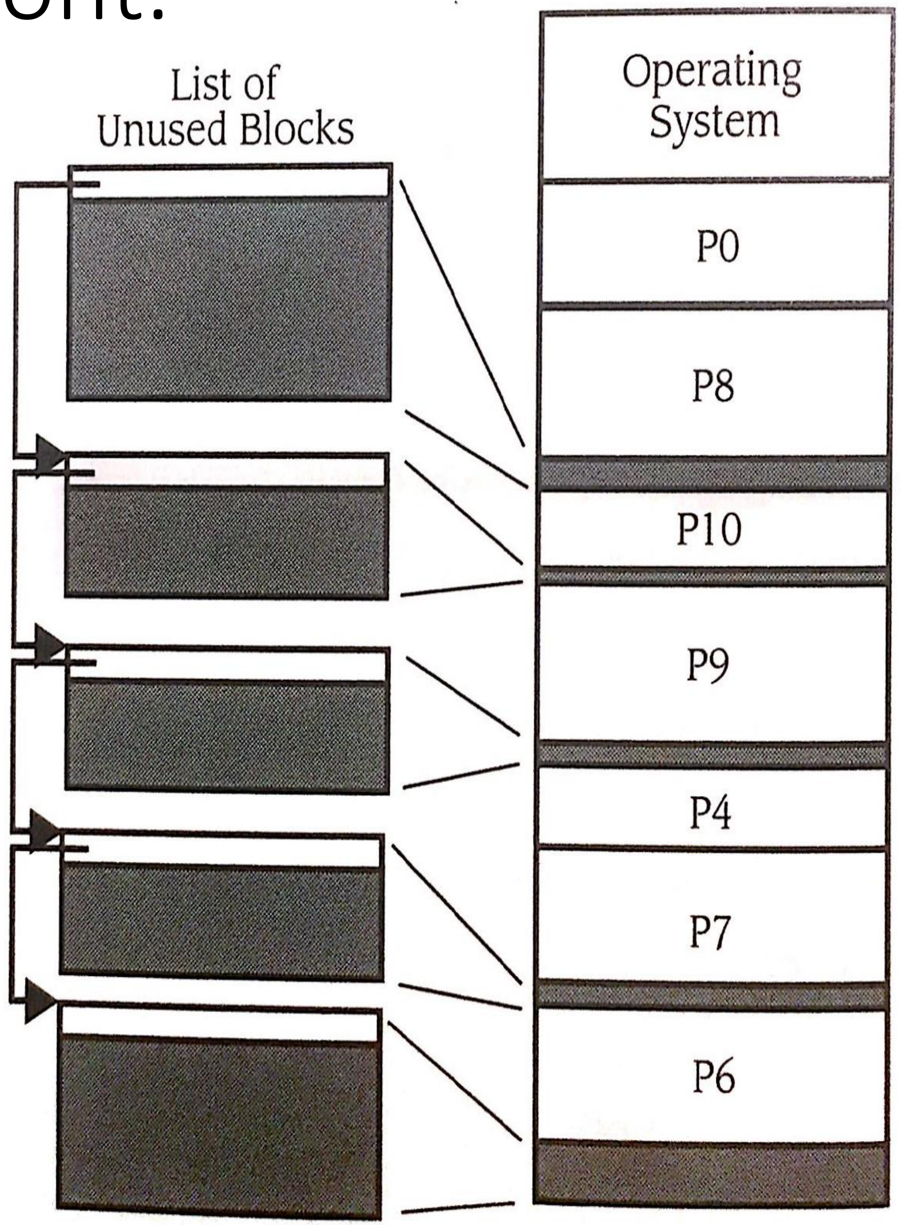
Variable partition strategies

- Does the hardware implement one set of relocation registers per process, or just a single set?
- How does the kernel keep track of the holes or fragments?



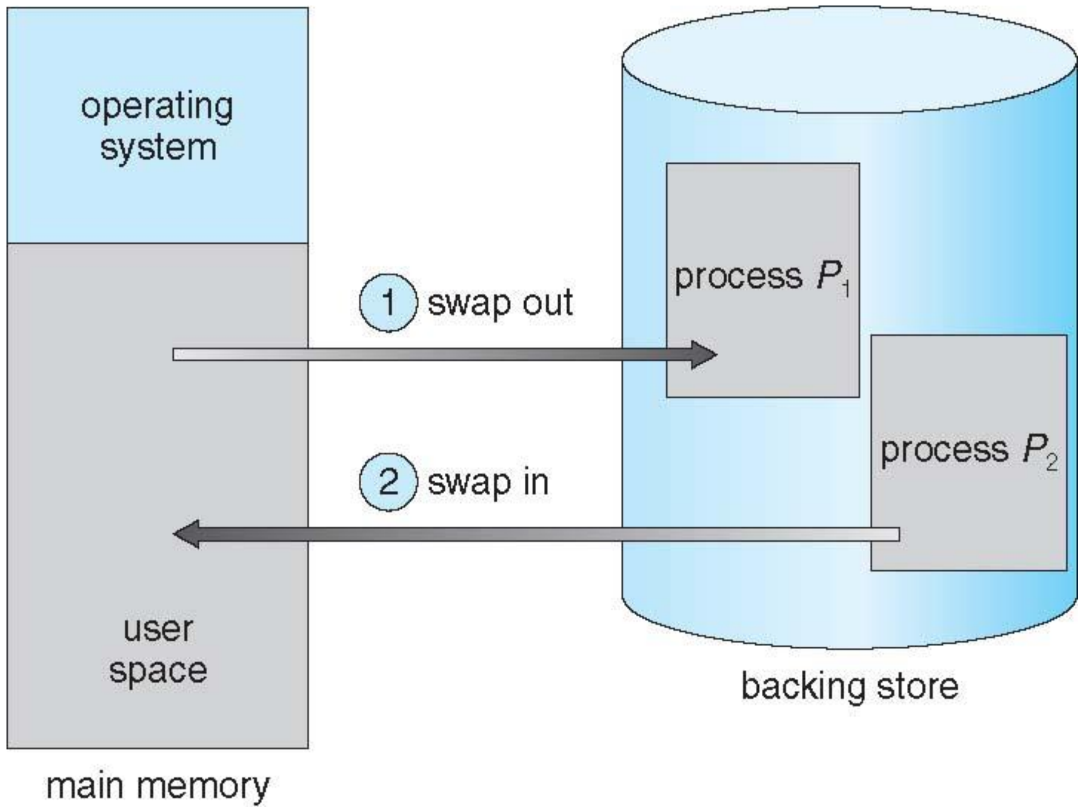
Variable partition strategies – cont.

- Prior to **compaction**, the system may keep track of holes using linked lists.
- If a process requests additional memory, the system may need to relocate it to accommodate the request.
- A number of allocation strategies may be used:
  - **Best fit:** allocates smallest hole that is larger than the process' required space.
  - **Worst fit:** allocates the largest hole of available memory. The theory is that this allows other processes to be allocated the remainder of the hole.
  - **First fit:** Allocates first hole in the linked list (to reduce processing time)
  - **Next fit:** Allocates next hole in the list (even if another was freed behind it). Thus, needs to have the list converted into a circular list. This ensures we try all the holes instead of just using holes at the beginning of the list.



Swapping

- With partitioning, the total memory space of all processes must be smaller than the available physical memory space.
- Swapping allows the system to circumvent that and thus increase the degree of multiprogramming
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk, large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images



Swapping – cont.

- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **queue** of ready-to-run processes which have memory images on disk

Swapping – cont.

- Does the swapped out process need to swap back into the same physical addresses?
  - Depends on address binding method – certainly not needed when relocation hardware is available or when relocation overhead is low.

- Modified versions of swapping are found on many systems (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

Context Switch Time including Swapping

- If next process to run (on CPU) is not in memory, then we need to swap out a process and swap in the target process
- Context switch time can then be very high
- Example: A 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)

Context Switch Time including Swapping – cont.

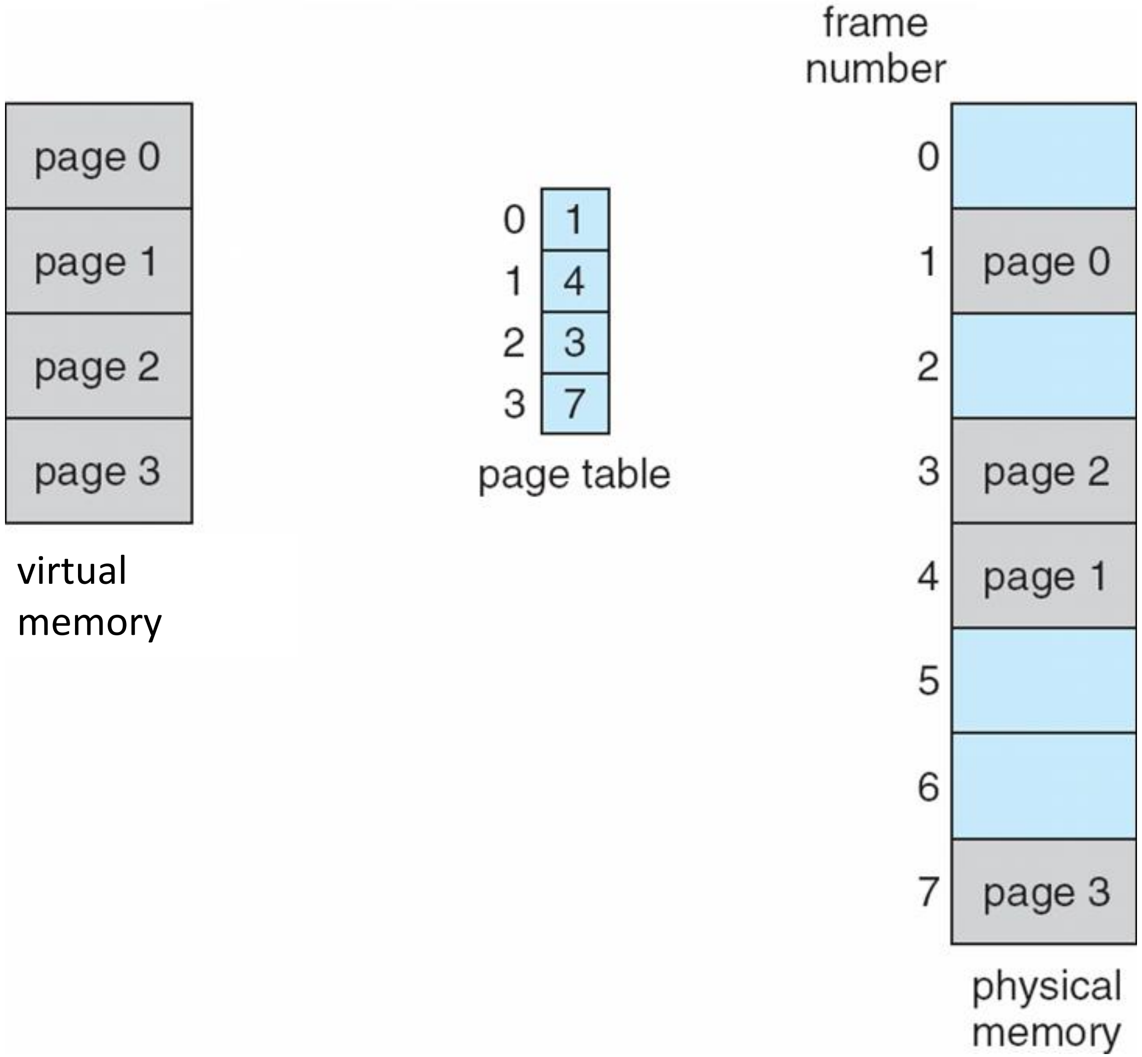
- Other constraints on swapping:
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common: Swap only when free memory extremely low

Introduction to Virtual memory

- Motivation:
  - Code needs to be in memory to execute, but **entire program is rarely needed** at the same time:
    - Error code, unusual routines, large data structures, etc.
    - Programs usually have a **locality of reference**, in which only a portion of the program is actually being invoked within each time window, e.g. loops.
- Consider ability to **partially load and execute** programs
  - Program no longer constrained by limits of physical memory and may thus have an address space that is much larger than the actual physical memory, this address space is referred to as the **virtual address space**.
  - Logical address can now be referred to as virtual addresses.
  - Each program takes less memory while running → a higher degree of multiprogramming may be achieved → increased CPU utilization.
  - Compared to swapping, less I/O time is then needed to swap processes in/out of memory → each user process runs faster
- Supported by segmentation or paging.

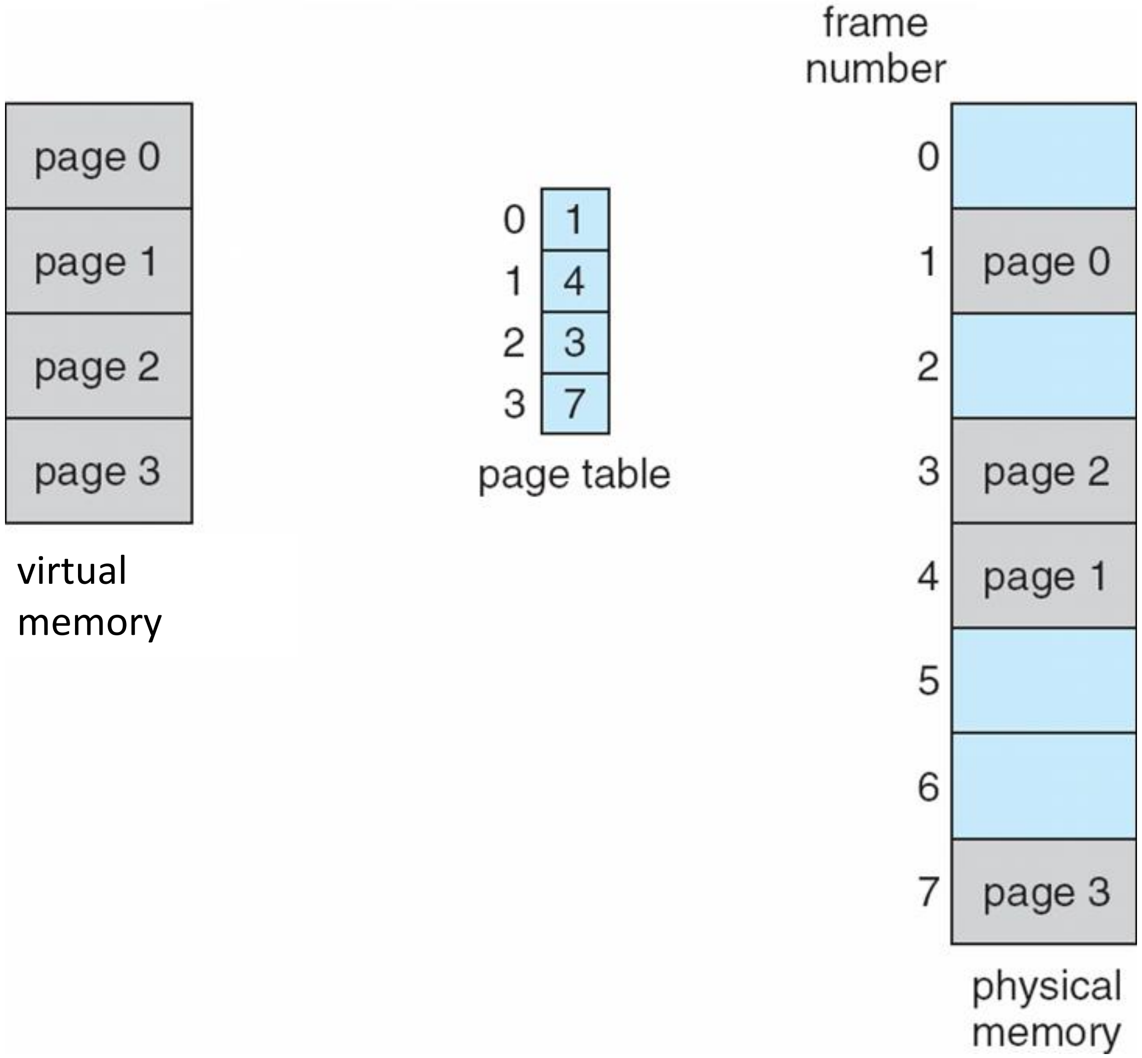
Memory paging

- A memory management scheme that supports virtual memory;
  - The physical address space a process occupies may not be contiguous
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks (as in segmentation)
- Divide physical memory into fixed-sized blocks called **frames**
  - **Size is power of 2**, e.g. between 512 bytes and 16 Mbytes
- Divide a program's virtual memory space into blocks of same size called **pages**
- Backing store likewise split into pages.



Memory paging – cont.

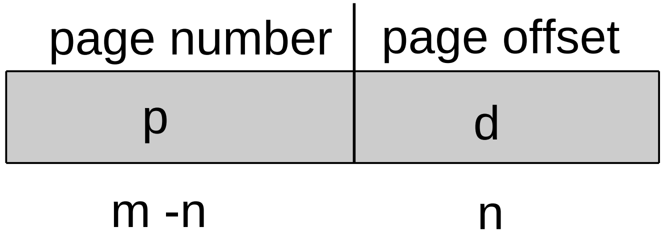
- The OS kernel keeps track of all free frames in main memory.
- To run a program of ***N*** pages, the kernel finds ***N*** free frames, then it loads the entire program using the N frames.
- The kernel also sets up a **page table** to translate logical to physical addresses
- Paging systems may suffer from **internal fragmentation**



Address Translation Scheme

- Page size is in powers of 2 → offsets within the page can be fully spanned using an offset address of n bits, where $\mathrm{n} = \log_2(page\ size)$.

- Thus, the virtual address of m-bits generated by CPU is divided into:
  - **Page offset** (*d*) – lower n bits of the virtual address → *2^{m-n}* pages may exist.
  - **Page number** (*p*) - upper (m-n) bits of the virtual address



page number | page offset

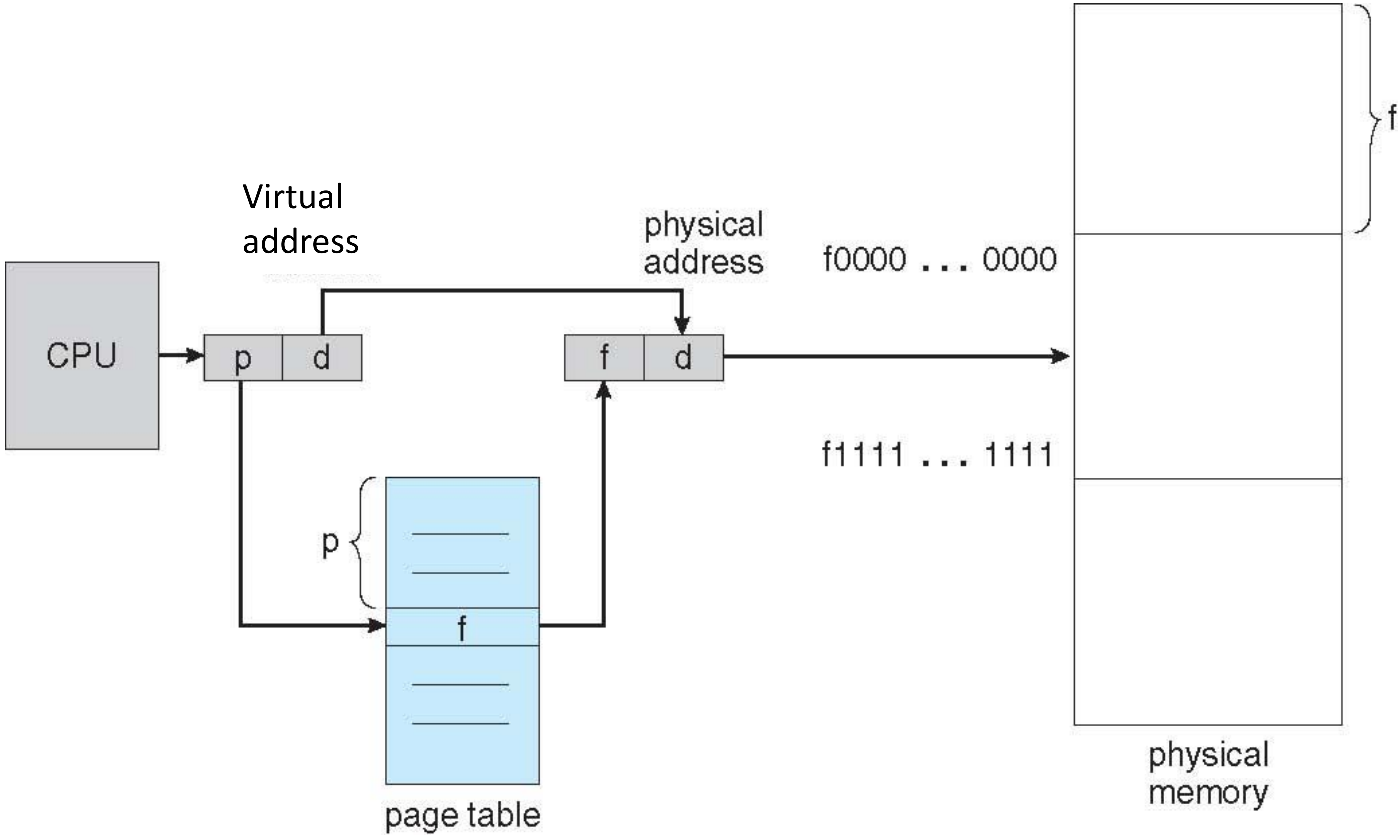
p | d

m -n | n

- After reaching the end of the page, incrementing the address by one results in:
  - Page number (*p*) incrementing by 1
  - Offset within the page (*d*) goes back to 0

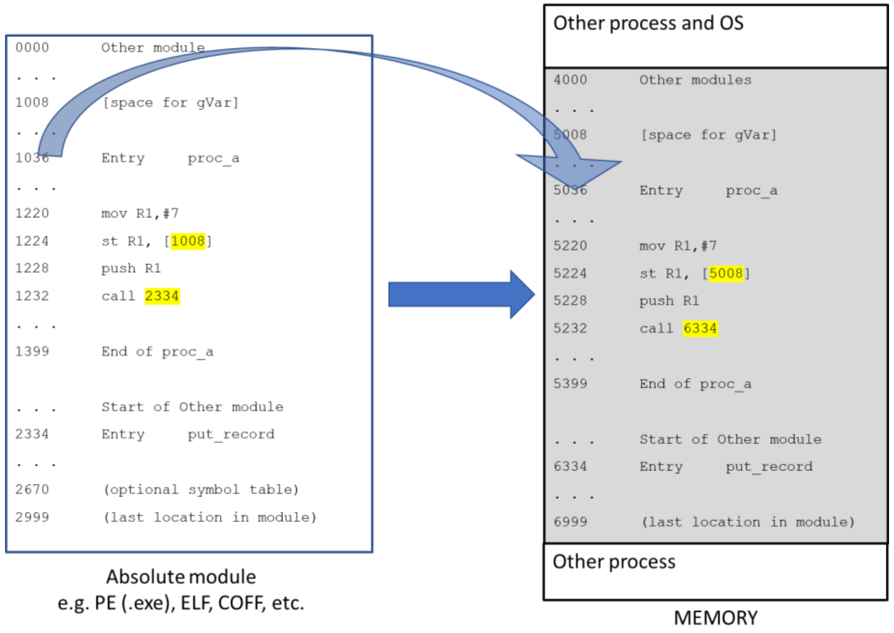
Paging hardware

- The page table holds an entry for each page in the process.
- The page table is used to map a virtual address into a physical address.
- The page number is used as an offset to that table.



Paging Example 1

- Process’ pages are allocated to frames in the physical memory.



```
0000      Other module
. . .
1008      [space for gVar]
. . .
1036      Entry     proc_a
. . .
1220      mov R1,#7
1224      st R1, [1008]
1228      push R1
1232      call 2334
. . .
1399      End of proc_a

. . .     Start of Other module
2334      Entry     put_record
. . .
2670      (optional symbol table)
2999      (last location in module)
```

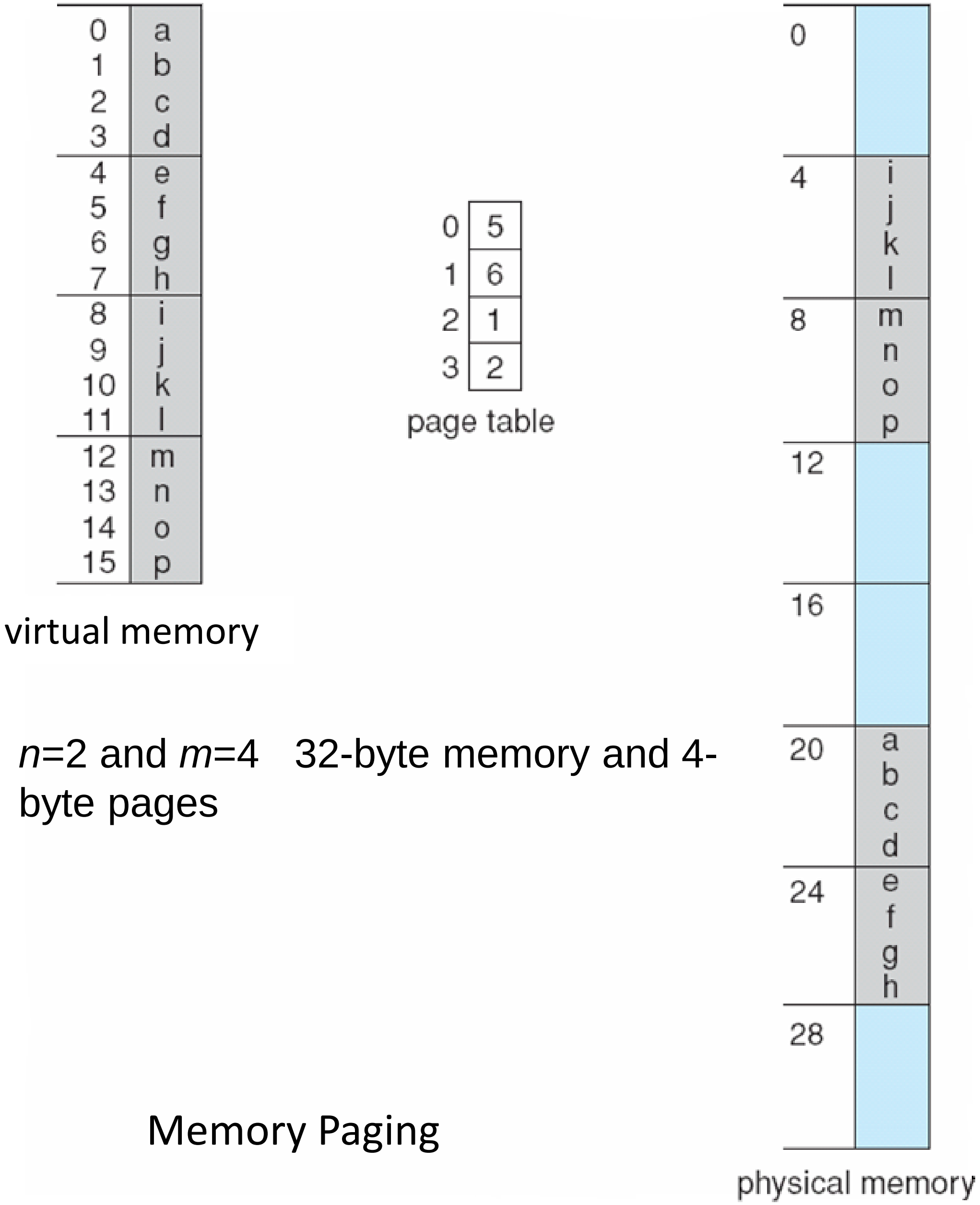
Absolute module
e.g. PE (.exe), ELF, COFF, etc.

```
Other process and OS
4000      Other modules
. . .
5008      [space for gVar]
. . .
5036      Entry     proc_a
. . .
5220      mov R1,#7
5224      st R1, [5008]
5228      push R1
5232      call 6334
. . .
5399      End of proc_a

. . .     Start of Other module
6334      Entry     put_record
. . .
6999      (last location in module)
Other process
```

MEMORY

Memory Partitioning



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

virtual memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

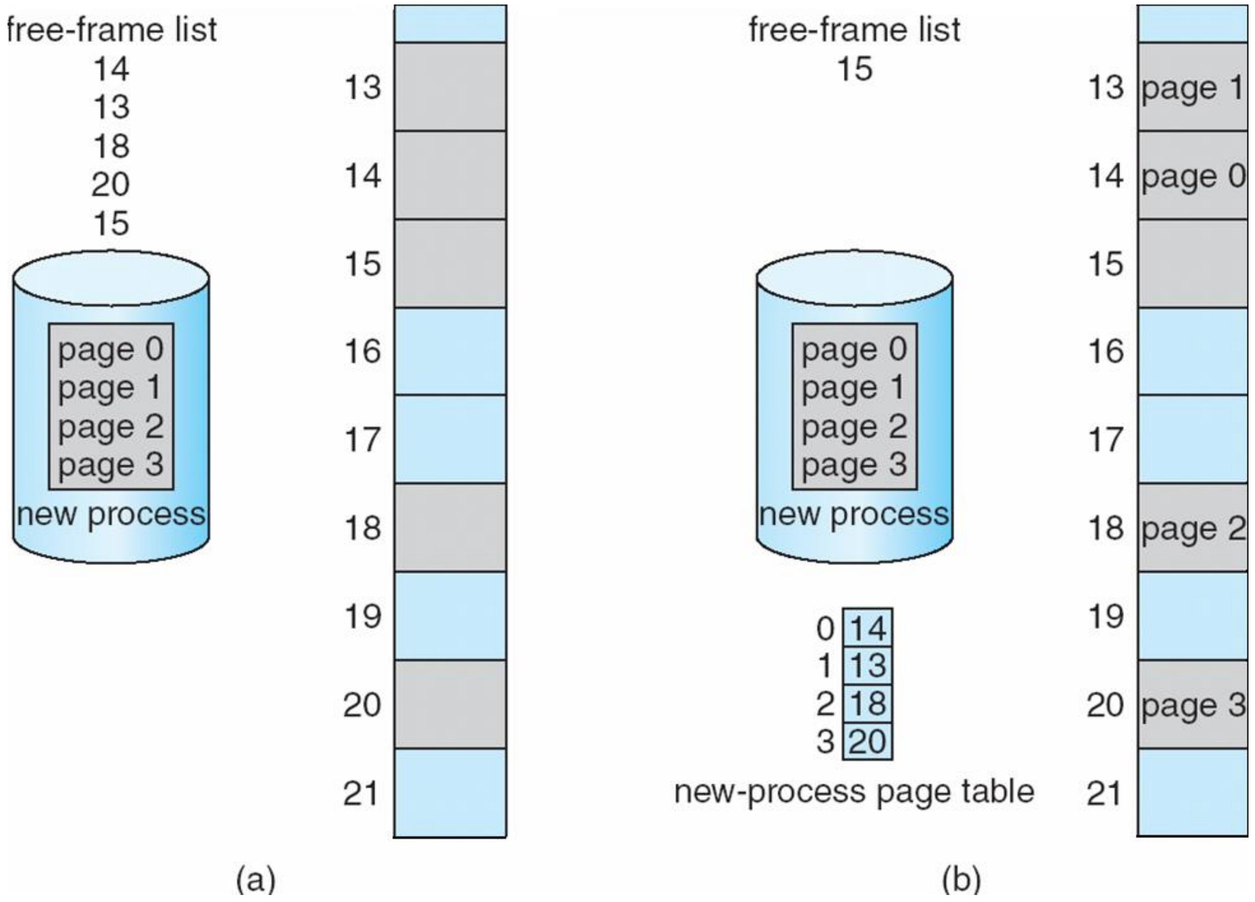
n=2 and *m*=4   32-byte memory and 4-byte pages

Memory Paging

Paging example 2

- Example - Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
- Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory are now very different.
- By implementation, a process can only access its own memory space.

Paging example 3- allocation of free frames



Before allocation

After allocation

Implementation of Page Table

- Page Tables are:
  - **Kept in** main memory (kernel's memory)
  - **Written by** the OS kernel (software)
  - **Read by** the Memory Management Unit (hardware)
- Two registers are used inside the MMU hardware to identify the page table:
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Issues in this scheme:
  - Every data/instruction access requires two memory accesses; one for the page table and one for the data / instruction
  - The dual-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 #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

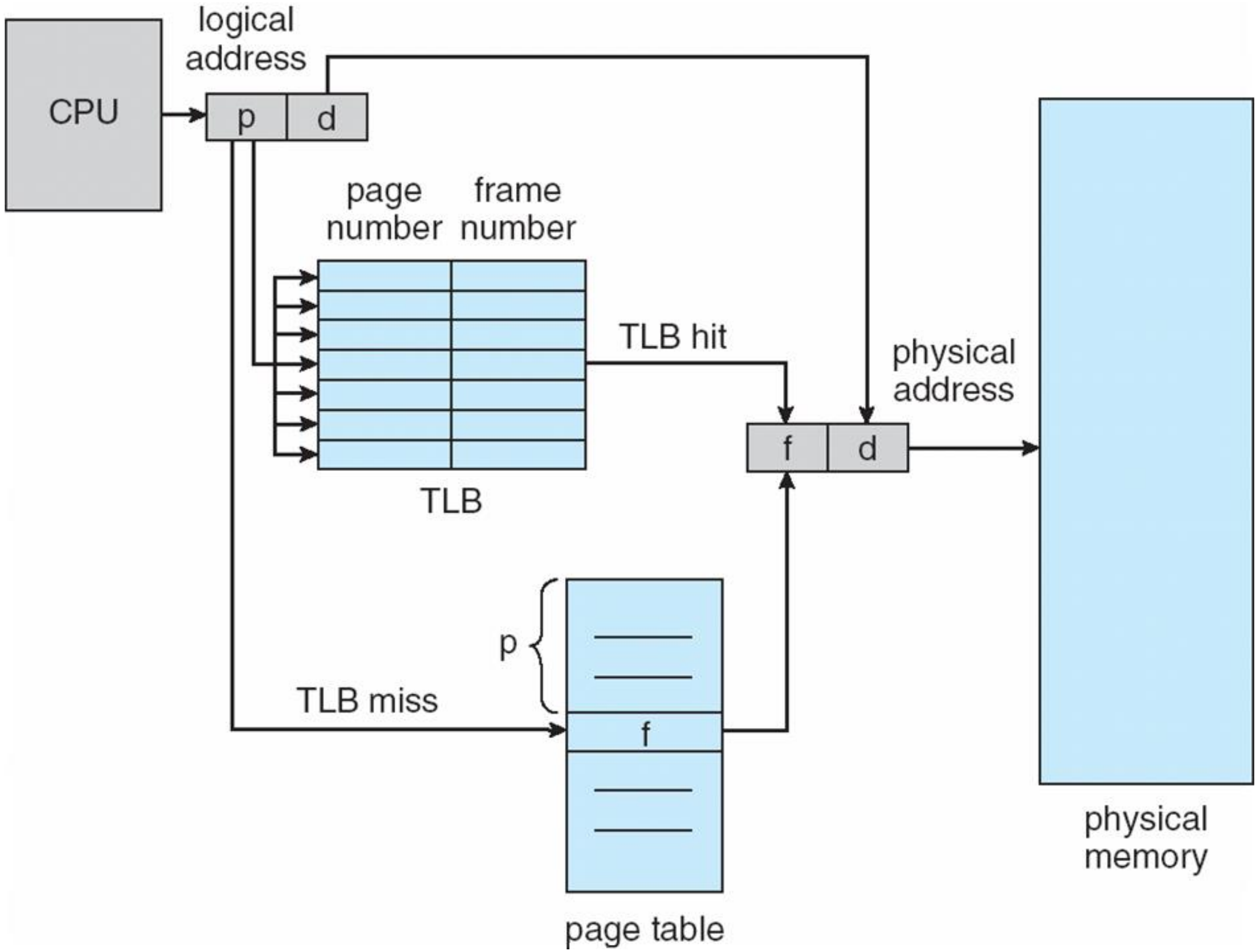
```
0000       Other module
. . .
1008       [space for gVar]
. . .
1036       Entry     proc_a
. . .
1220       mov R1,#7
1224       st R1, [1008]
1228       push R1
1232       call 2334
. . .
1399       End of proc_a


. . .      Start of Other module
2334       Entry     put_record
. . .
2670       (optional symbol table)
2999       (last location in module)
```

Absolute module
e.g. PE (.exe), ELF, COFF, etc.

Paging Hardware With TLB

- Note the difference in structure between the TLB and the page table stored in main memory.
  - In the page table stored in main memory, each entry only has the frame number. The page number is used as an offset into a page table entry.
  - In the TLB, each entry has a page number and a frame number. **Why?**



Implementation of Page Table – TLB's

- Many TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs are typically small (64 to 1,024 entries)
- On a **TLB page miss**, value is loaded into the TLB from the page table (in memory) for faster access next time
  - If TLB is full → replacement policies must be considered.
  - Some entries can be **wired down** for permanent fast access
  - The new value is loaded by the MMU hardware without OS intervention.
- Thus, TLBs are:
  - Located inside the MMU (Hardware)
  - Read and written by the MMU (Hardware)

Effective Access Time for a page in main memory

- Let δ be the associative memory (TLB) lookup time and ε be the main memory access time.
- Define the percentage of time a page number is found in the associative memory as the hit ratio, α
- **Effective Access Time** (**EAT**)

$$\text{EAT} = (\delta + \epsilon)\alpha + (\delta + 2\epsilon)(1 - \alpha)$$

For $\epsilon \gg \delta$,

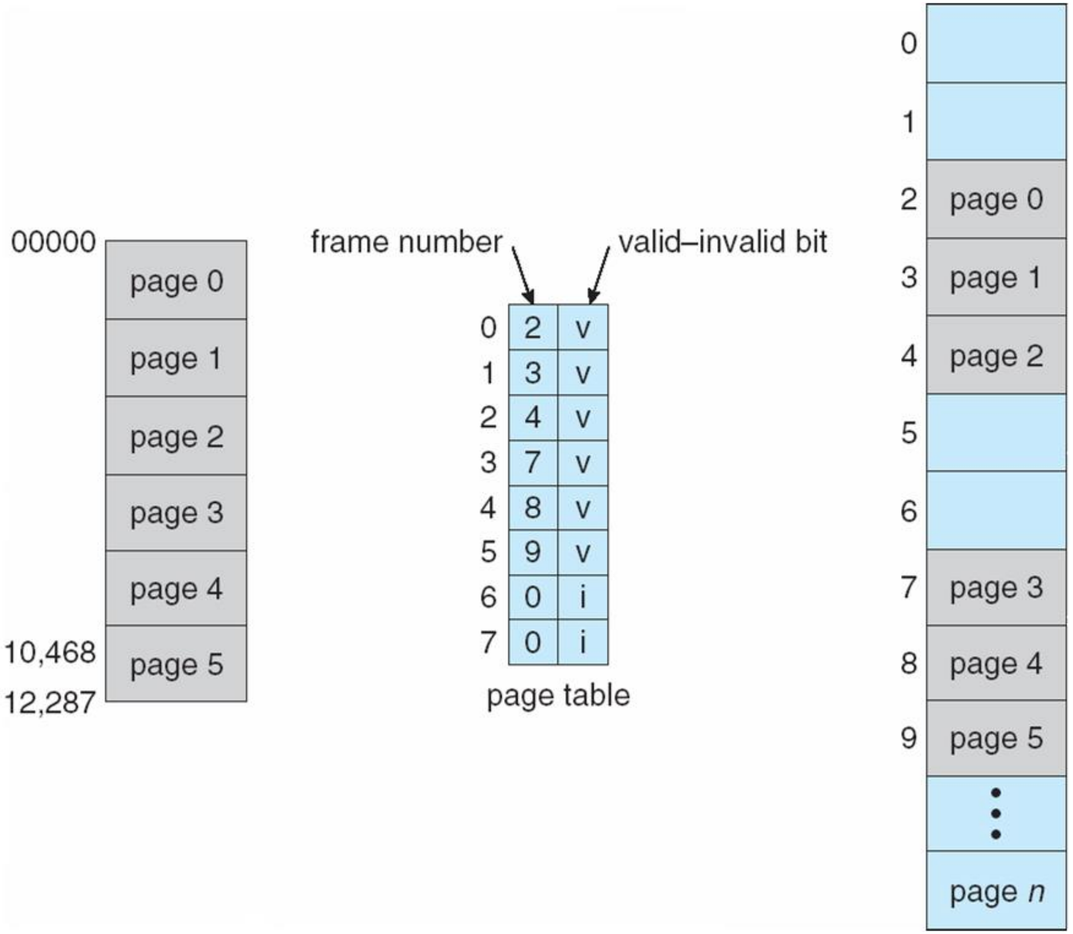
$$\text{EAT} \approx \epsilon\alpha + 2\epsilon(1 - \alpha)$$

- Consider α = 80% and ε = 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε =100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
    - Q: With TLB usage, how is the page validity determined?
- Any violations result in an exception, thus invoking the kernel's handler

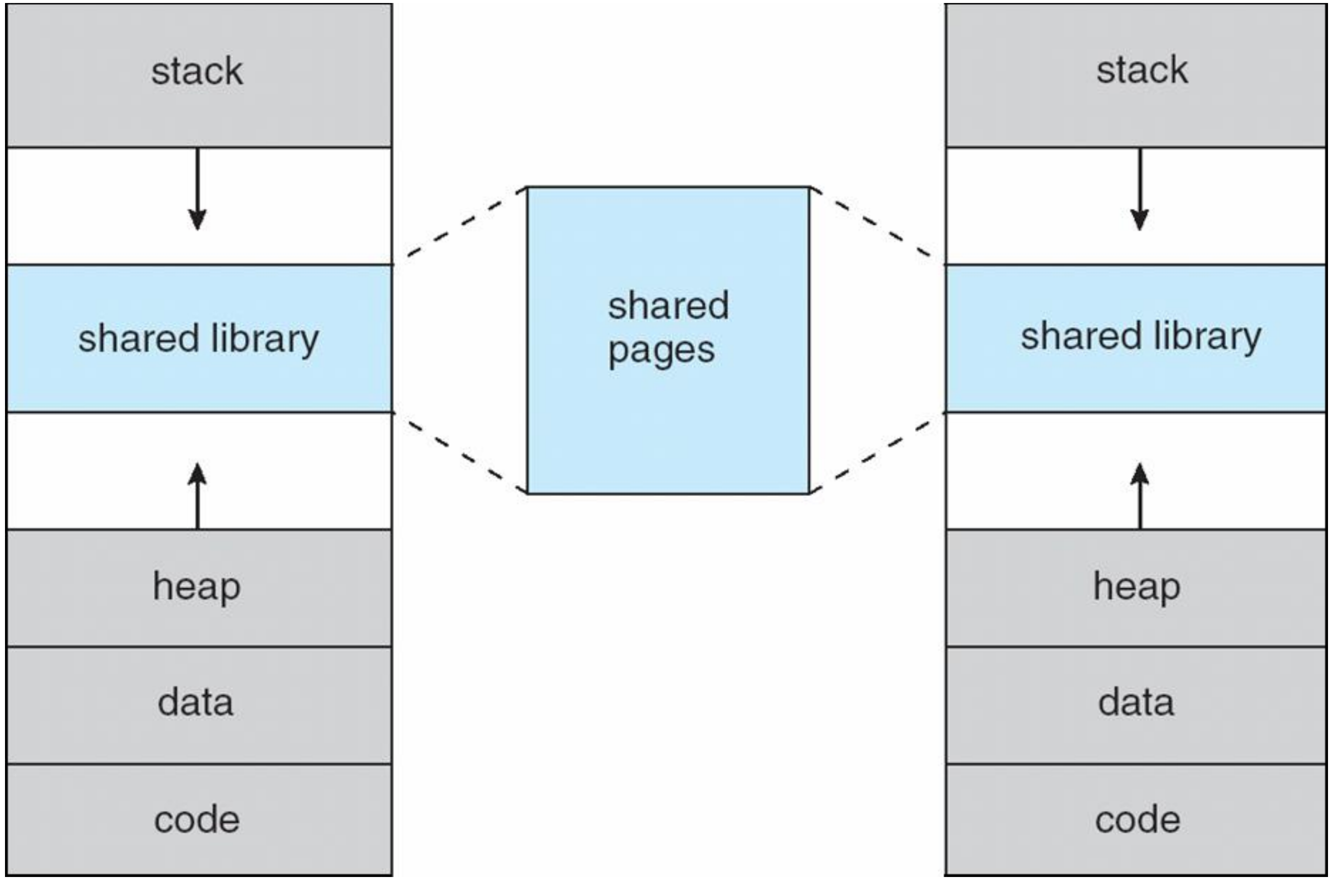
Valid (v) or Invalid (i) Bit In A Page Table



0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Shared pages

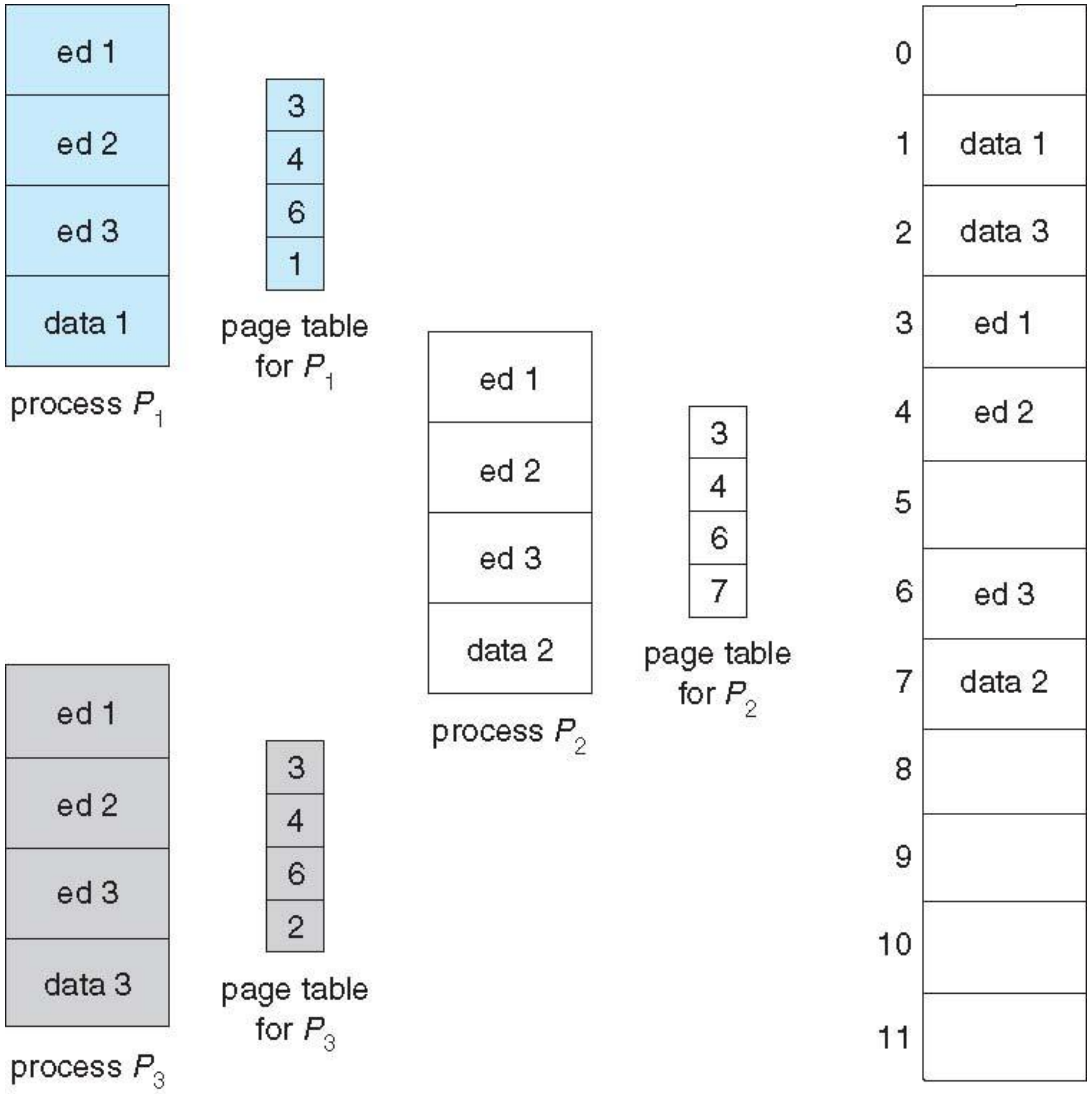
- System libraries may be shared via mapping into virtual address space
- Shared memory may be implemented by mapping pages (read, write and/or execute) into virtual address space



Shared Pages – cont.

- **Shared code and data**
  - One copy of read-only **reentrant code (text)** can be shared among processes (i.e., text editors, compilers, window systems)
  - **Shared data** is useful for inter-process communication if sharing of read-write pages is allowed
  - Applicable to whole programs (e.g. text editors) as well as shared libraries.
  - Such shard pages are similar to multiple threads sharing the same process space
- **Private (not shared) code and data**
  - Each process keeps a separate copy of its code and data
  - The pages for the private code and data can appear anywhere in the virtual address space

Shared Pages - example


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11