OPERATING SYSTEMS: MEMORY

Memory Management

Goals

- □ To know the memory manager functions.
- □ To know the phases in generating an executable and the structure of a process memory map.
- □ To understand schemes for contiguous allocation memory.
- □ To be able to use memory management services for memory mapped files and dynamic libraries.

Contents

- **Functions of the memory manager.**
- Executable generation and dynamic libraries.
- Virtual memory.
- Memory manager services.

Memory manager goals

- OS multiplexes resources among processes.
  - Each process believes it has the whole machine for it.
    - **Process management**: Processor sharing.
    - **Memory management**: Memory sharing.

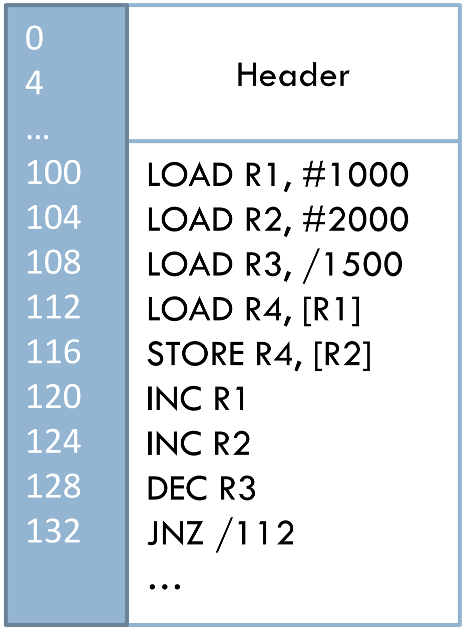
- Goals:
  - To offer each process its own separate logical space.
  - To provide protection among processes.
  - To allow processes to share memory.
  - To give support to process regions.
  - To Maximize multiprogramming degree.
  - To provide processes with very large memory maps.

Independent logical spaces

□ Program memory location is unknown.

□ Executable code generates references between 0 and N.

□ Example: Program copying a vector.



```
0      Header
4
...
100    LOAD R1, #1000
104    LOAD R2, #2000
108    LOAD R3, /1500
112    LOAD R4, [R1]
116    STORE R4, [R2]
120    INC R1
124    INC R2
128    DEC R3
132    JNZ /112
       ...
```

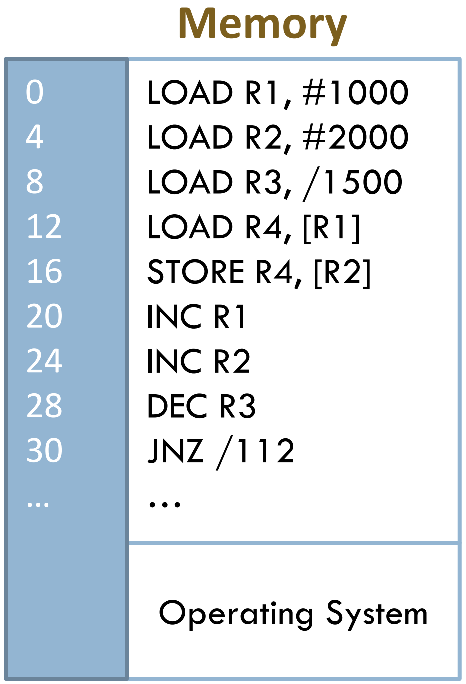
□ Destination vector starts at address 2000.

□ Vector size at address 1500.

□ Source vector starts at address 1000.

Monoprogrammed Operating Systems

- OS in upper addresses.
- Program loaded at address 0.
- Control is passed to program.



Memory

0	LOAD R1, #1000
4	LOAD R2, #2000
8	LOAD R3, /1500
12	LOAD R4, [R1]
16	STORE R4, [R2]
20	INC R1
24	INC R2
28	DEC R3
30	JNZ /112
…	…
	Operating System

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Relocation

- In a multiprogrammed operating system, all programs cannot be located starting at the same address (e.g. 0).
  - Need to be able to relocate a program starting from a given memory address.
  - **Relocation** -> Translate **logical addresses** into **physical addresses**.

- **Logical addresses**: Memory addresses generated by program.
- **Physical addresses**: Main memory addresses allocated to the process.

- **Translation function**:

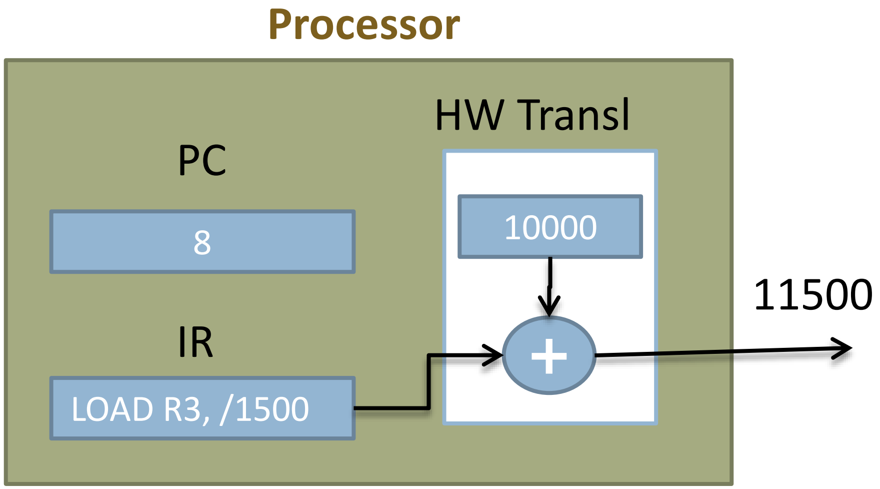
  Translation(ProcId, logical_addr) -> physical_addr

Relocation

- Relocation created independent logical space for process.
  - Operating System must be able to access to processes logical spaces.
- Example: Program has memory allocated starting at address 10000.
  - Add 10000 to all generated addresses.
- **Two alternatives**:
  - Hardware relocation.
  - Software relocation.

Hardware relocation

- Hardware Memory Management Unit (MMU) is in charge of translation.
- OS is in charge of:
  - Storing translation function for each process.
  - Specify which function must be applied by hardware.
- Program loaded into memory without modification



Processor

PC

8

IR

LOAD R3, /1500

HW Transl

10000

+

11500

Memory

…	…
10000	LOAD R1, #1000
10004	LOAD R2, #2000
10008	LOAD R3, /1500
10012	LOAD R4, [R1]
10016	STORE R4, [R2]
10020	INC R1
10024	INC R2
10028	DEC R3
10030	JNZ /112
…	…

Software relocation

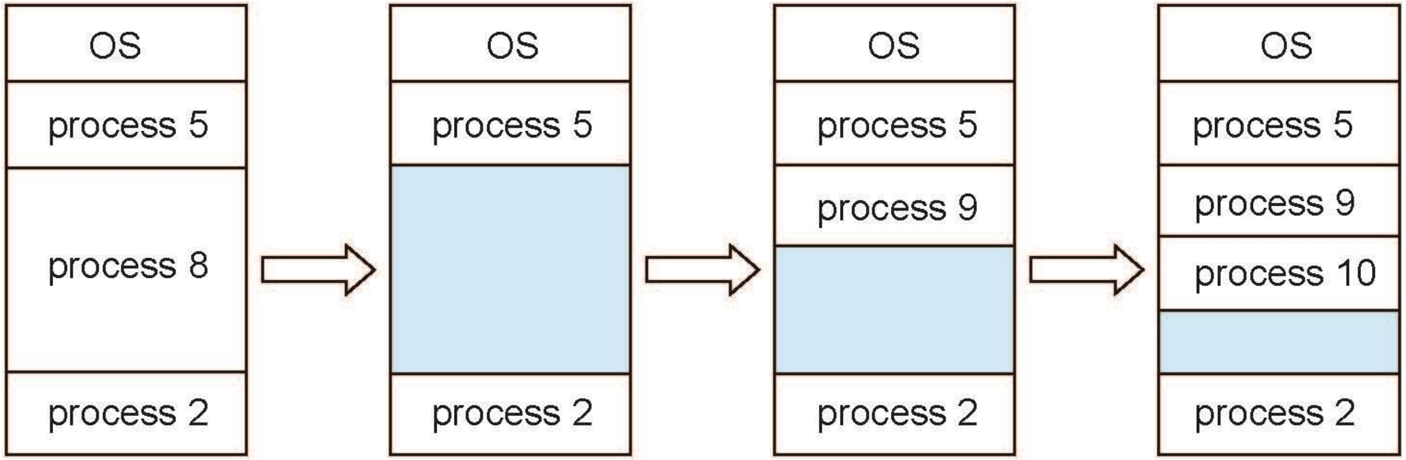
- Address translation during program load.
- Program in memory different from executable.
- Drawbacks:
  - Does not ensure protection.
  - Does not allow program movement at runtime.

Memory

…	…
10000	LOAD R1, #11000
10004	LOAD R2, #12000
10008	LOAD R3, /11500
10012	LOAD R4, [R1]
10016	STORE R4, [R2]
10020	INC R1
10024	INC R2
10028	DEC R3
10030	JNZ /112
…	…

Multiple-partition allocation

- Multiple-partition allocation
  - 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
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - **50-percent rule** (0.5N)/(N+0.5N) = 1/3 may be unusable

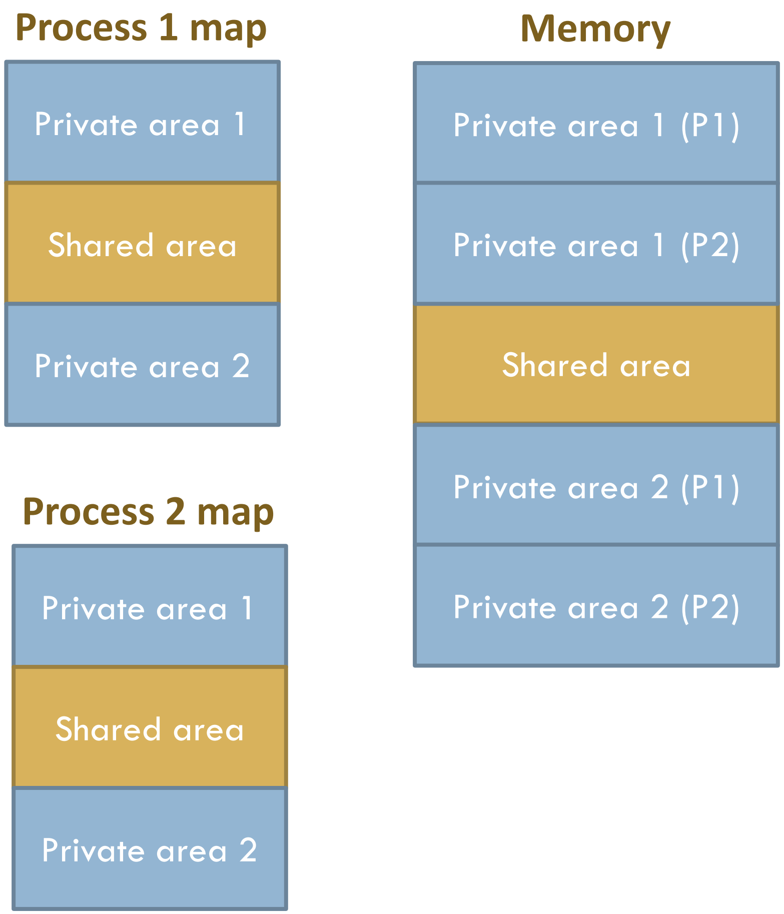
Protection

- **Monoprogramming**: OS protection.
- **Multiprogramming**: Additionally among processes.

- Translation must create **disjoint spaces.**
- Need to validate all addresses generated by program.
  - Detection must be performed by processor hardware.
  - Handling performed by OS.
- In systems with I/O and memory single map:
  - Translation allows to avoid processes to directly access I/O devices.

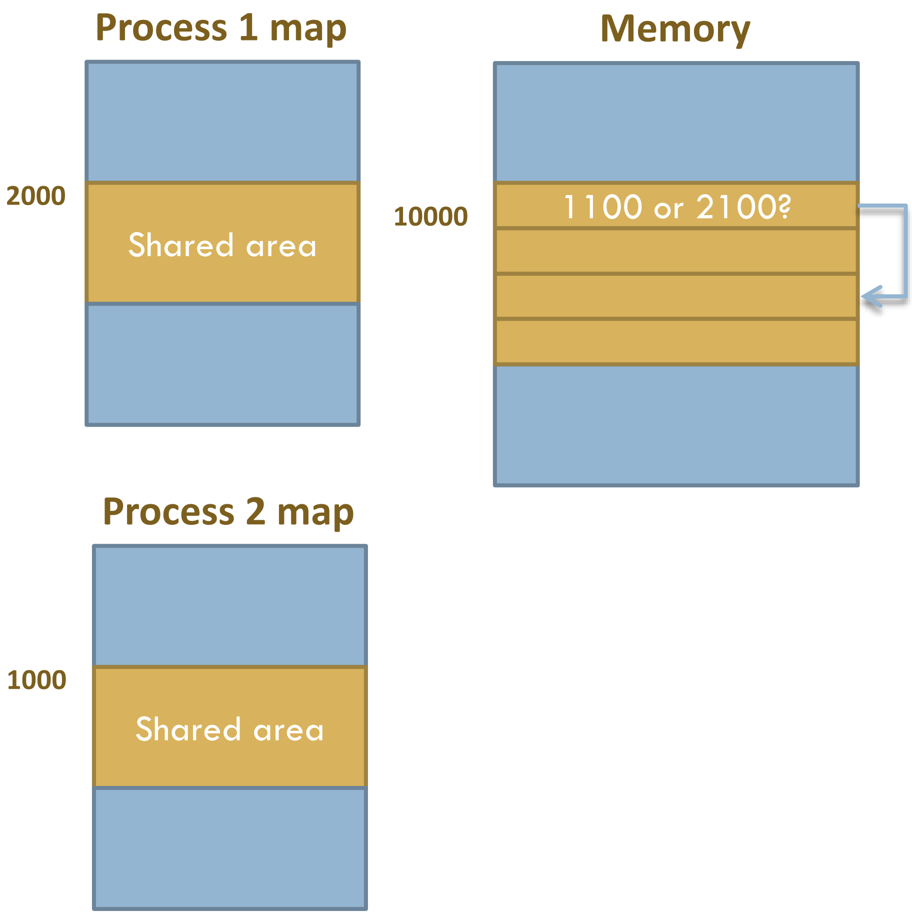
Memory sharing

- Logical addresses from 2 or more processes map to the same physical address.
- Under OS control.
- Benefits:
  - Processes running same program share code.
  - Very fast mechanism for inter process communication (IPC).
- Requires non contiguous allocation.



Problems with memory sharing

- If location at shared region contains reference to another location in the region.
- Example with code regions:
  - Shared region contains branch instruction to another location within region.
- Example with data regions:
  - Region contains a list with pointers.



Regions support

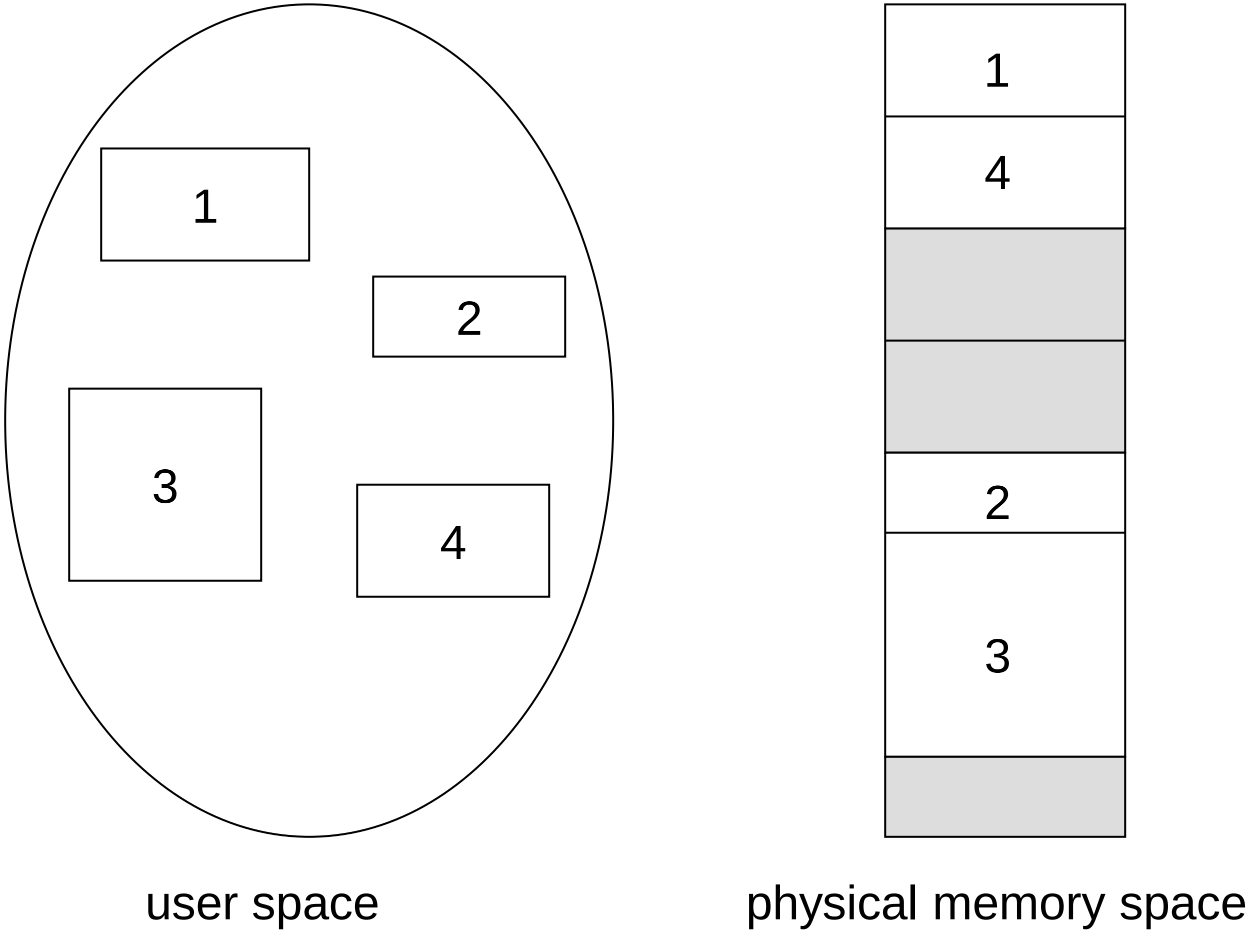
- Non homogeneous process map.
  - Set of regions with different characteristics.
  - Example: Non modifiable region with code.

- Dynamic process map.
  - Regions change its size (e.g. stack).
  - Regions are created and destroyed.
  - There are non allocated areas (holes).

- Memory manager must support those characteristics:
  - Detection of not allowed accesses to a region.
  - Detection of accesses to non allocated areas (holes).
  - Avoid allocating space for non allocated areas (holes).

- OS must store a region table for each process.

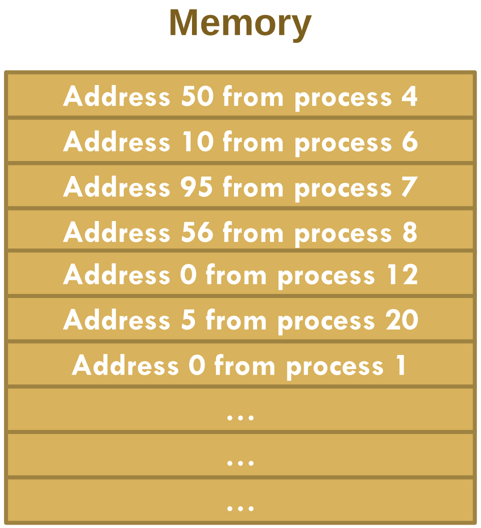
Logical View of Segmentation



Performance maximization

- □ Memory allocation maximizing multiprogramming degree.
- □ Memory *wasted* due to:
  - Non usable rests (fragmentation).
  - Tables required by memory manager.
- □ Lower fragmentation -> Larger tables.
- □ Compromise: Paging.
- □ Use of virtual memory to increase multiprogramming degree.

One address pages
- No fragmentation
- Not realizable due to PT size



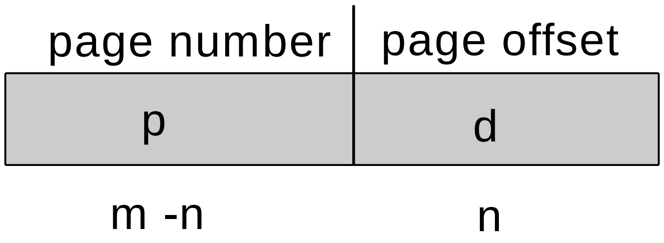
Memory
Address 50 from process 4
Address 10 from process 6
Address 95 from process 7
Address 56 from process 8
Address 0 from process 12
Address 5 from process 20
Address 0 from process 1
…
…
…

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids the problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of 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
- Backing store likewise split into pages
- Still have Internal fragmentation

Address Translation Scheme

- 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



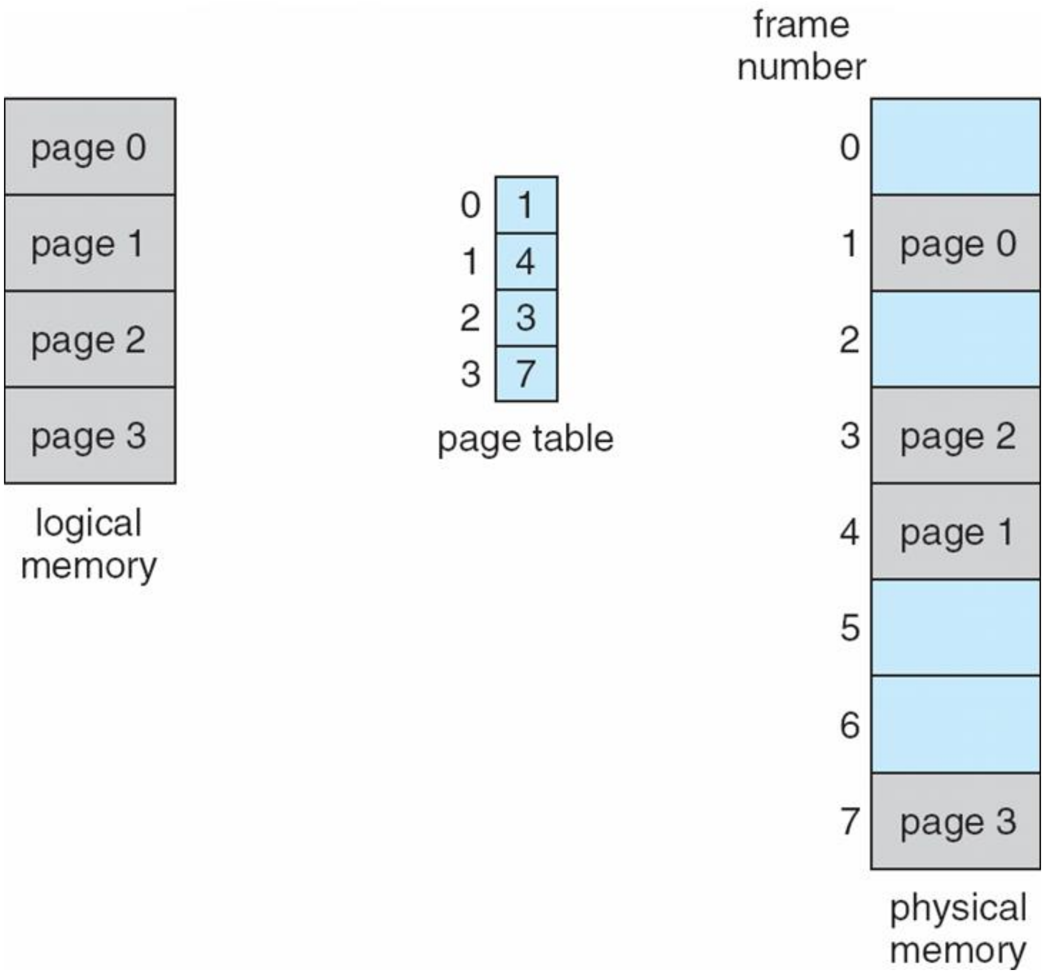
page number | page offset

p | d

m -n | n

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

Paging Model of Logical and Physical Memory



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**) a hardware register that stores the length of the page table
- Any violations result in a trap to the kernel

Memory Protection

Suppose, for example, that in a system with a 14- bit address space (0 to 16383) with pages of 2KiB, we have a program that should use only addresses 0 to 10468

00000	Page 0
	Page 1
	Page 2
	Page 3
	Page 4
10468	Page 5
12287	

	Frame number	Valid-invalid bit
0	2	V
1	3	V
2	4	V
3	7	V
4	8	V
5	9	V
6	0	i
7	0	i

0	
1	
2	Page 0
3	Page 1
4	Page 2
5	
6	
7	Page 3
8	Page 4
9	Page 5
	…
	Page n

Very large memory maps

- Processes need larger and larger maps.
    - More advanced applications with new features.
- Solved thanks to virtual memory.
- Historically ***overlays*** were used:
    - Program divided in phases executed in sequence.
    - In each point of time only a phase is resident in memory.
    - Each phase performs its task and loads next one.
    - Non transparent: All tasks performed by programmer.

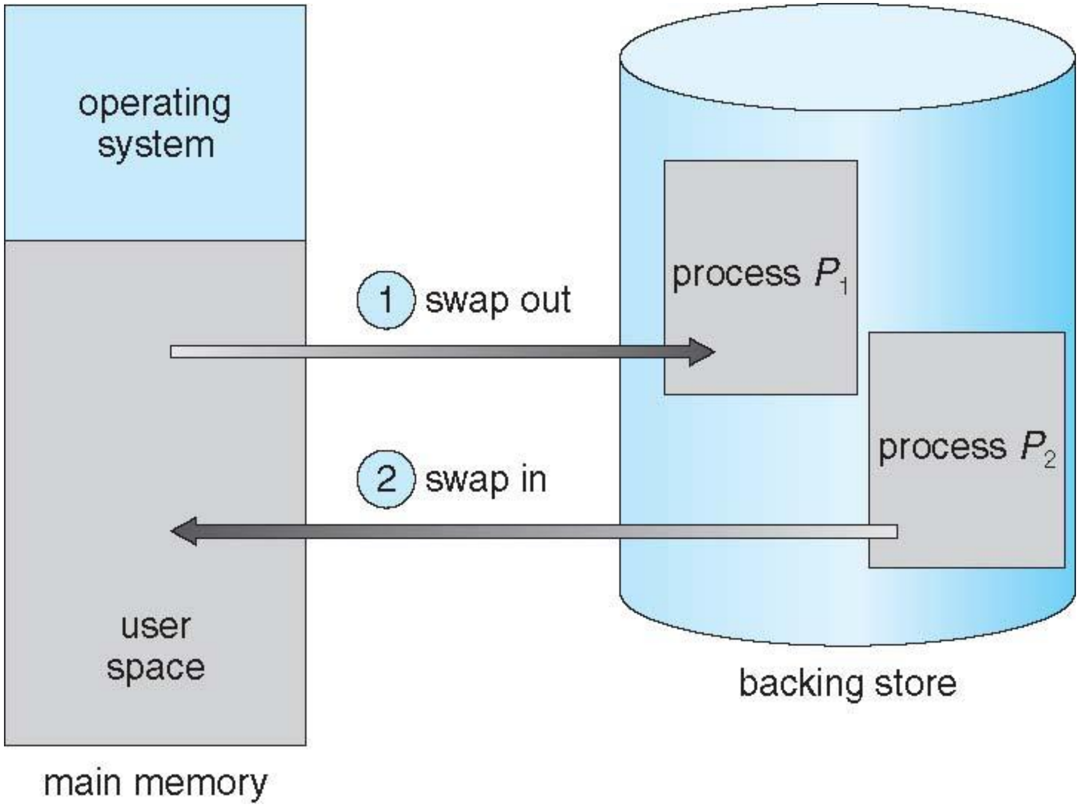
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- 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

Schematic View of Swapping



Context Switch Time including Swapping

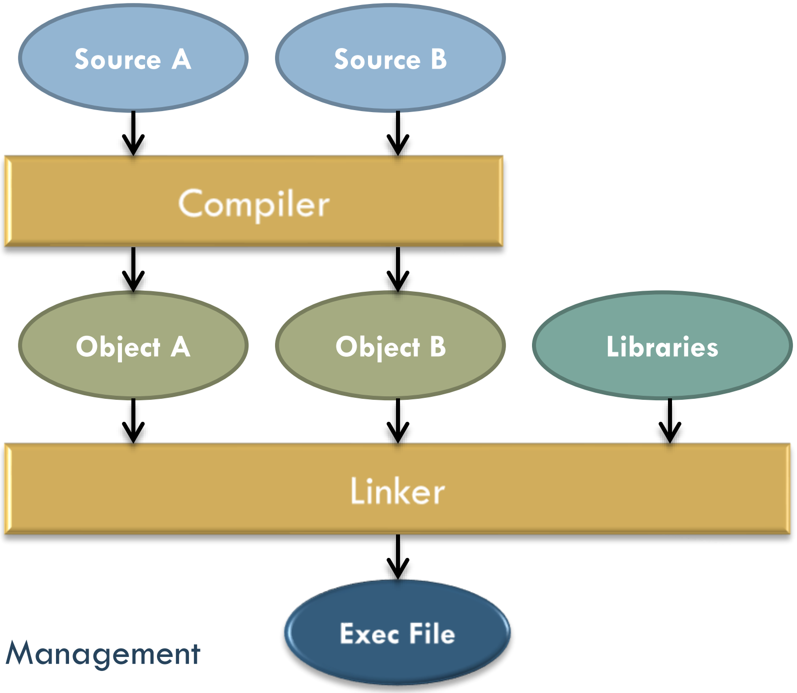
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Contents

- □ Functions of the memory manager.
- □ **Executable generation and dynamic libraries.**
- □ Virtual memory.
- □ Memory manager services.

Phases in executable generation

- Application: Set of modules in high level language.
- Processing in two phases: Compilation and linking.
- Compilation:
  - Solve references within each source module.
  - Generate a single object.
- Linking:
  - Solve references to other object modules.
  - Solve references to symbols in libraries.
  - Generate executable including libraries.



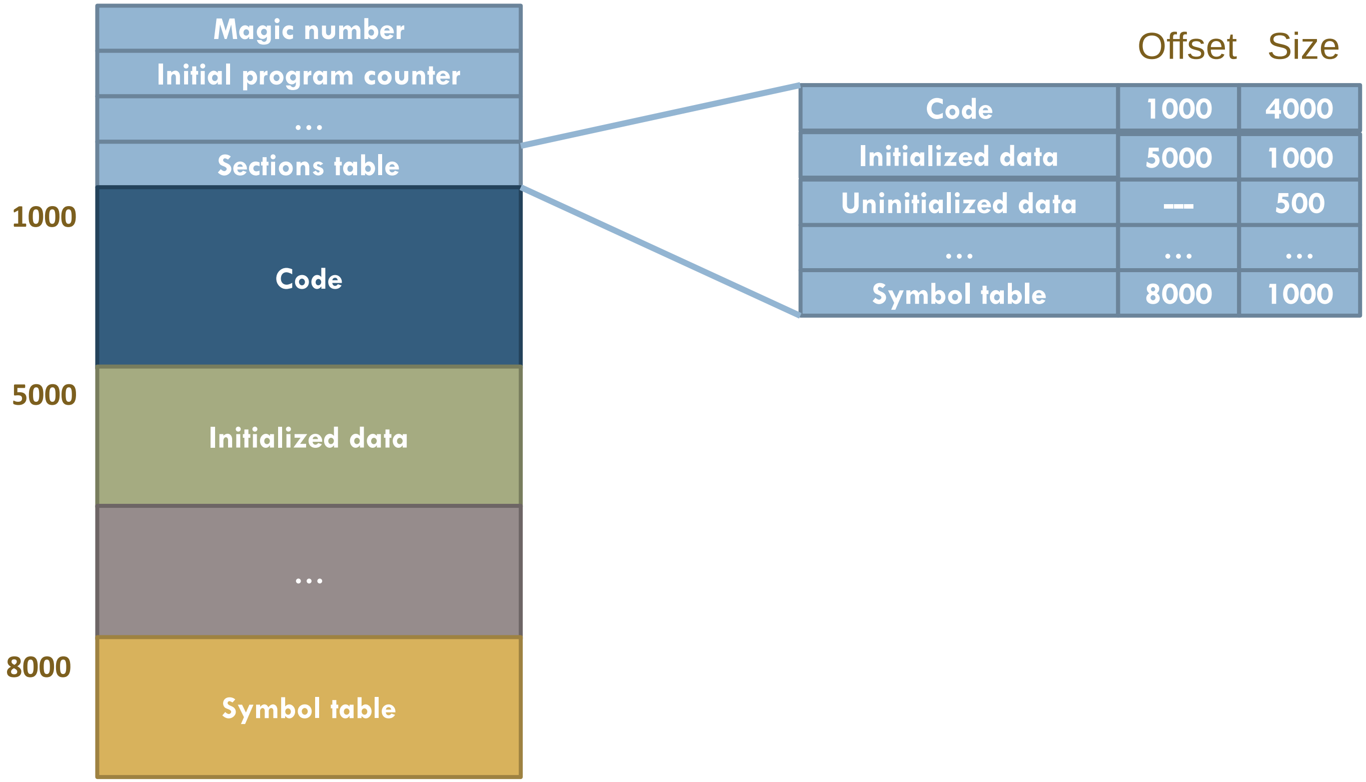
Object libraries

- Library: Collection of related object modules .
- System library or created by users.
- Static libraries:
  - Linking: Links program object modules and libraries.
  - Self-contained executable.
- Drawbacks of static linking:
  - Large executables.
  - Library function code replicated in may executables.
  - Multiples copies in memory for library function code.
  - Updating a library implies to link again.

Dynamic libraries

- Load and link library at runtime.
- Executable contains:
  - Library name.
  - Run-time load and linking routine.
- At 1st library symbol reference at run time:
  - Routine load and links corresponding library.
  - Sets instruction performing reference so that next references access library symbol.
    - Problem: It would imply modifying program code.
    - Typical solution: Indirect reference through table.

Executable format

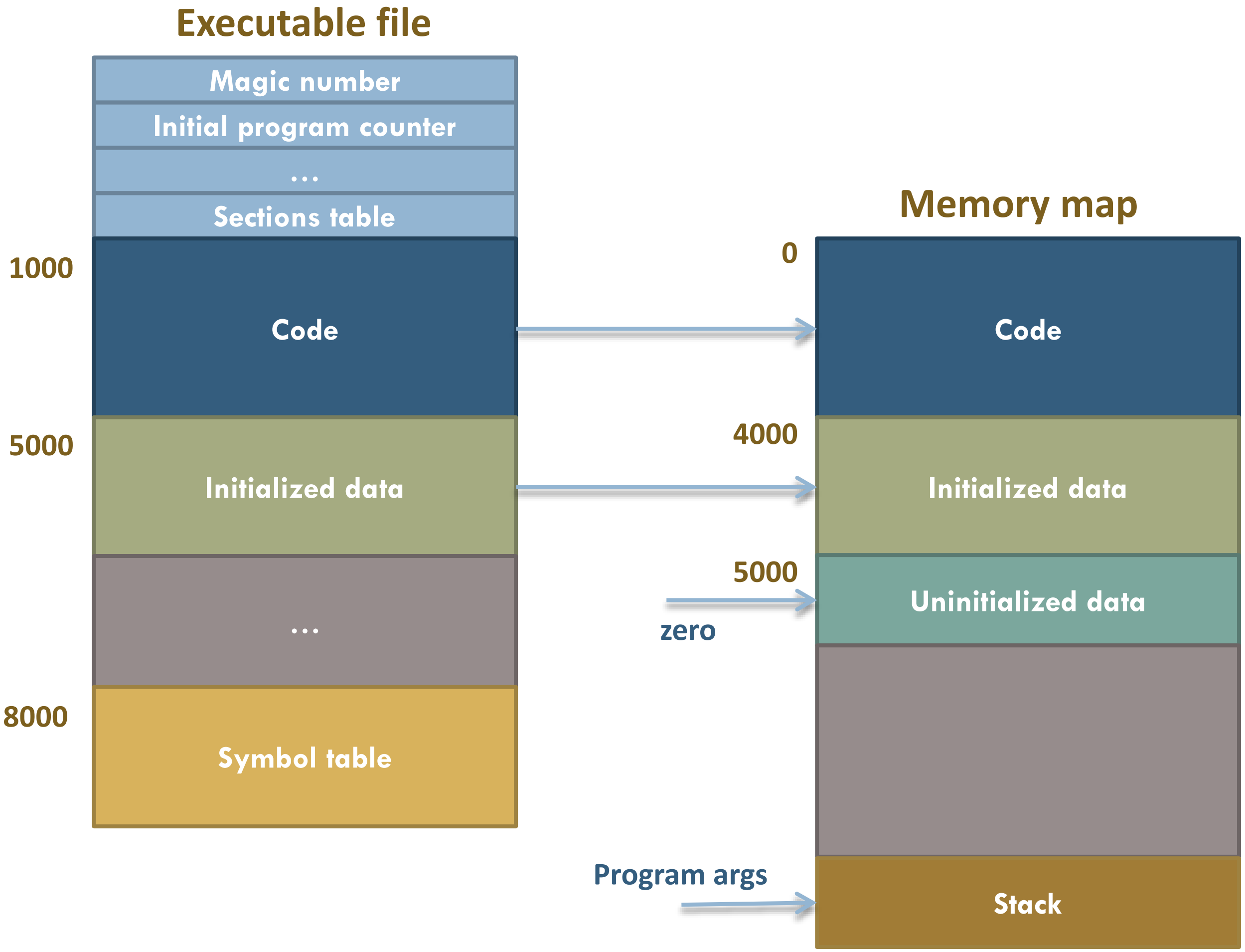


	Offset	Size
Code	1000	4000
Initialized data	5000	1000
Uninitialized data	---	500
…	…	…
Symbol table	8000	1000

Creating an initial memory map from executable

- Program execution: Create map from executable.
  - Initial map regions -> Executable sections.
  - **Code:**
    - Shared, RX, Fixed size, support in executable.
  - **Initialized data**:
    - Private, RW, Fixed size, support in executable.
  - **Non initialized data**:
    - Private, RW, Fixed size, without support (fill with zero).
  - **Stack**:
    - Private, RW, Variable size, without support (fill with zero).
    - Grows towards decreasing addresses.
    - Initial stack: Program arguments.

Creating initial memory map from executable



Other regions

- During process execution new regions are created.
  - Memory map is of dynamic nature.

- **Heap region.**
  - Support for dynamic memory (malloc/calloc/free in C).
  - Private, RW, Variable size, Without support (fill with zero).
  - Grows towards increasing addresses.
- **Thread stacks.**
  - Each thread corresponds to a region.
  - Same characteristics as process stack.
- **Dynamic library loading.**
  - Regions created associated to library code and data.

Other regions

- **Shared memory.**
  - Region associated to shared memory area.
  - Shared, Variable size, without support (fill with zero).
  - Protection specified in mapping.
- **Memory mapped file.**
  - Region associated to mapped file.
  - Variable size, Support in file.
  - Protection and use (shared/private) specified in mapping.

Region characteristics

Region	Support	Protection	Shared/ Private	Size
Code	File	RX	Shared	Fixed
Initialized data	File	RW	Private	Fixed
Uninitialized data	No support	RW	Private	Fixed
Stacks	No support	RW	Private	Variable
Heap	No support	RW	Private	Variable
Mapped file	File	User def.	Shared/ Private	Variable
Shared memory	No support	User def.	Shared	Variable

Memory mapped files

- Generalization of virtual memory.
  - Entry in PT reference a user file.
- Program requests mapping a file (or portion) in its own map.
  - May specify protection and use (shared/private).
- OS fills corresponding entries with:
  - Non resident, load from file.
  - Private/Shared and protection as specified by call.
- When program accesses memory location associated to mapped file, it is really accessing file

Contents

- □ Functions of the memory manager.
- □ Executable generation and dynamic libraries.
- □ Virtual memory.
- □ **Memory manager services.**

Concept

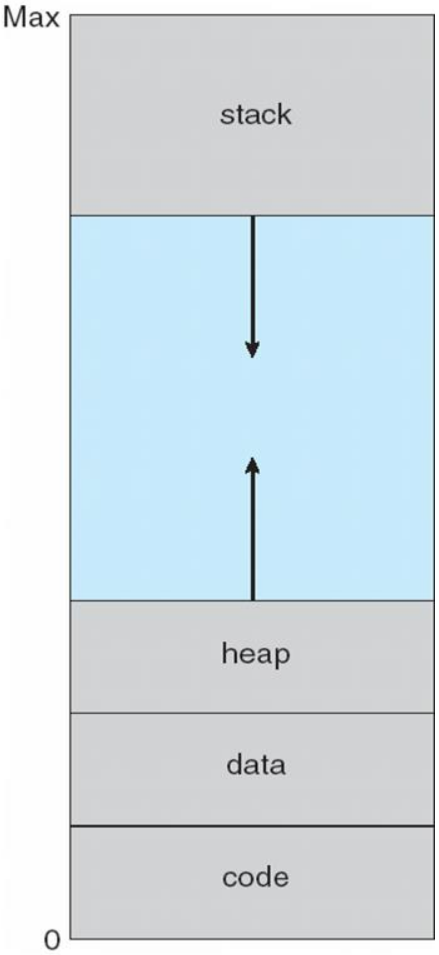
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Concept

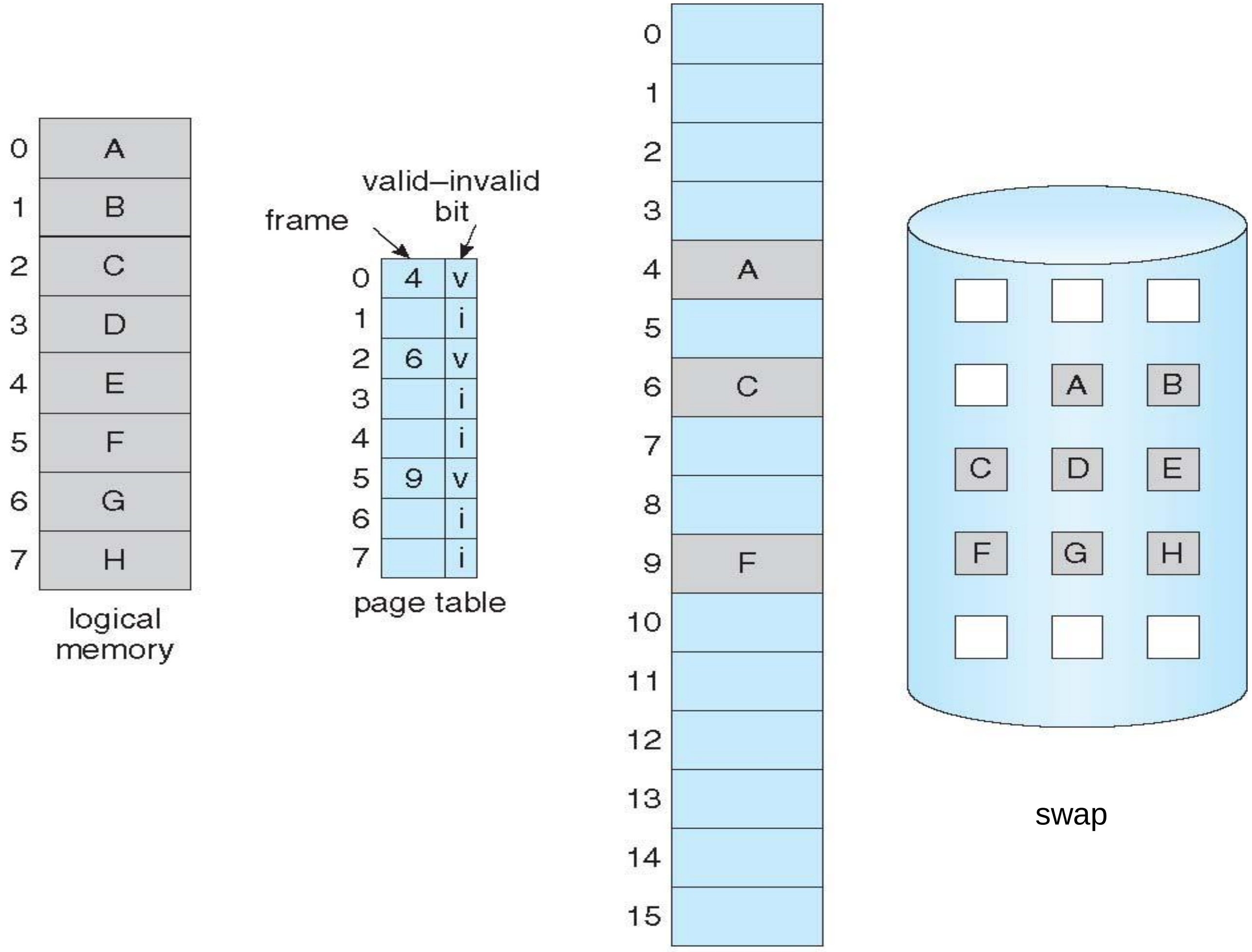
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is 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
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Page Table When Some Pages Are Not in Main Memory

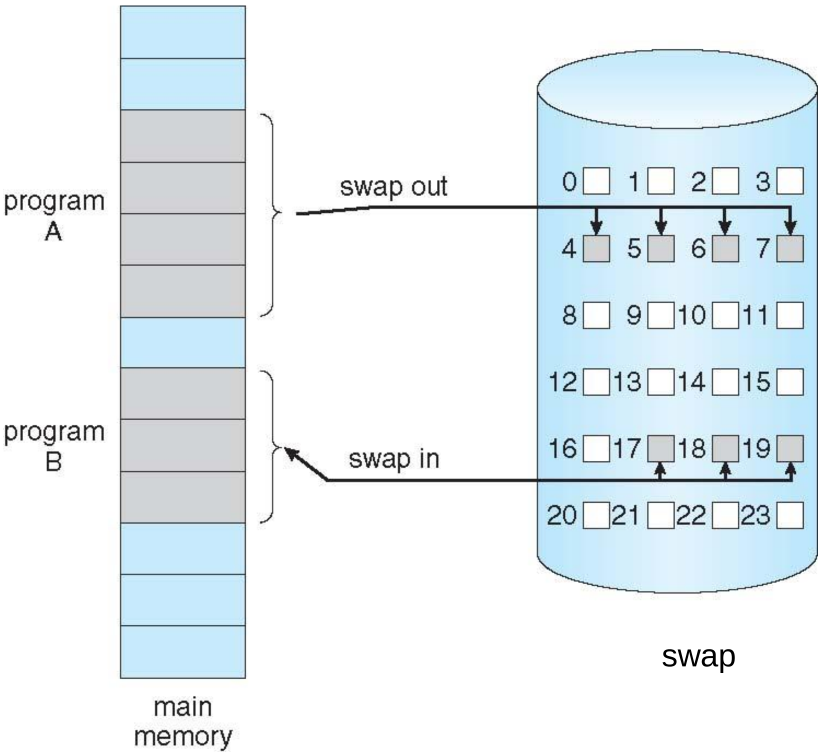


	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

Demand Paging

- Bring a page into memory when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



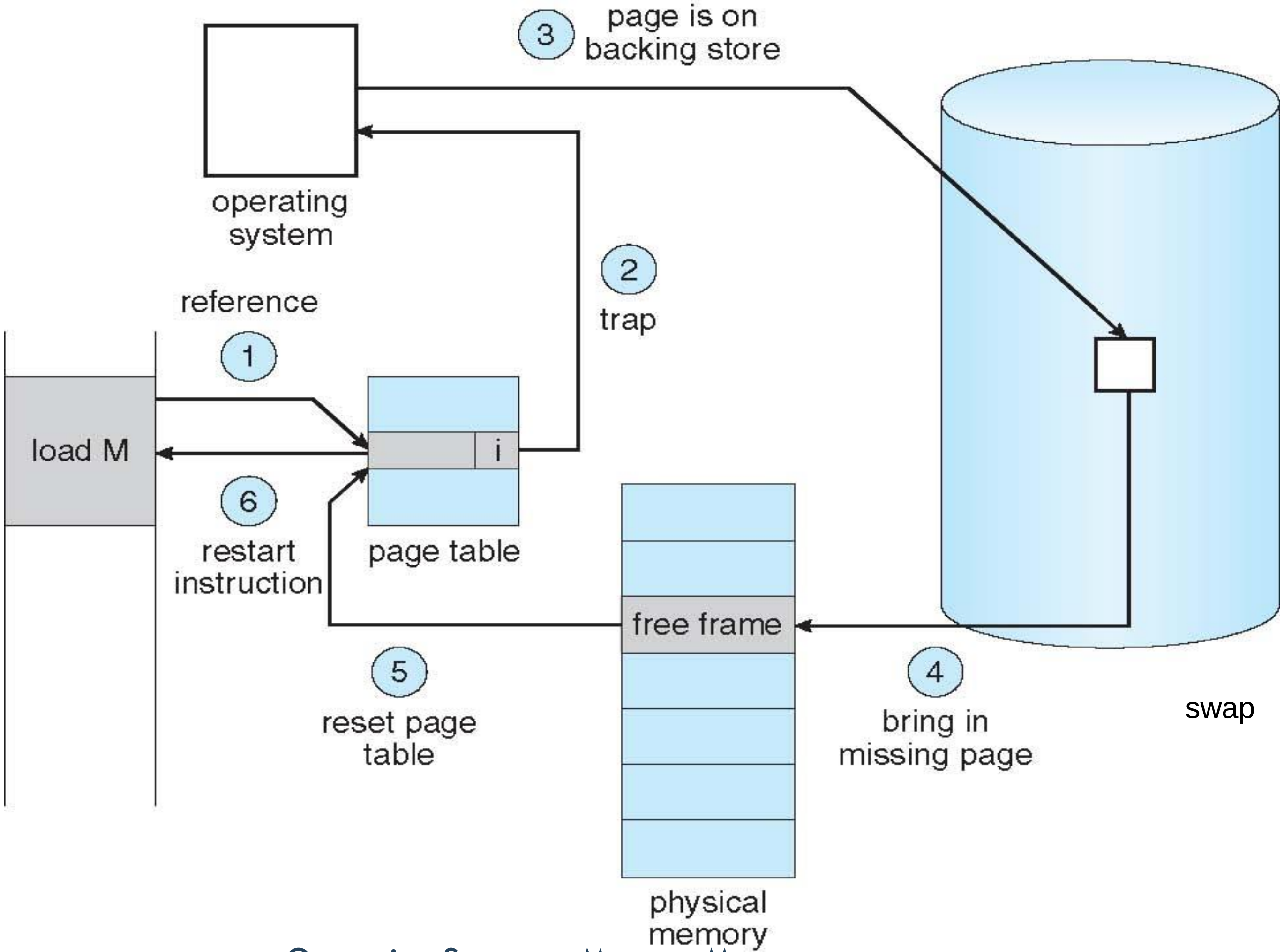
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



What Happens if There is no Free Frame?

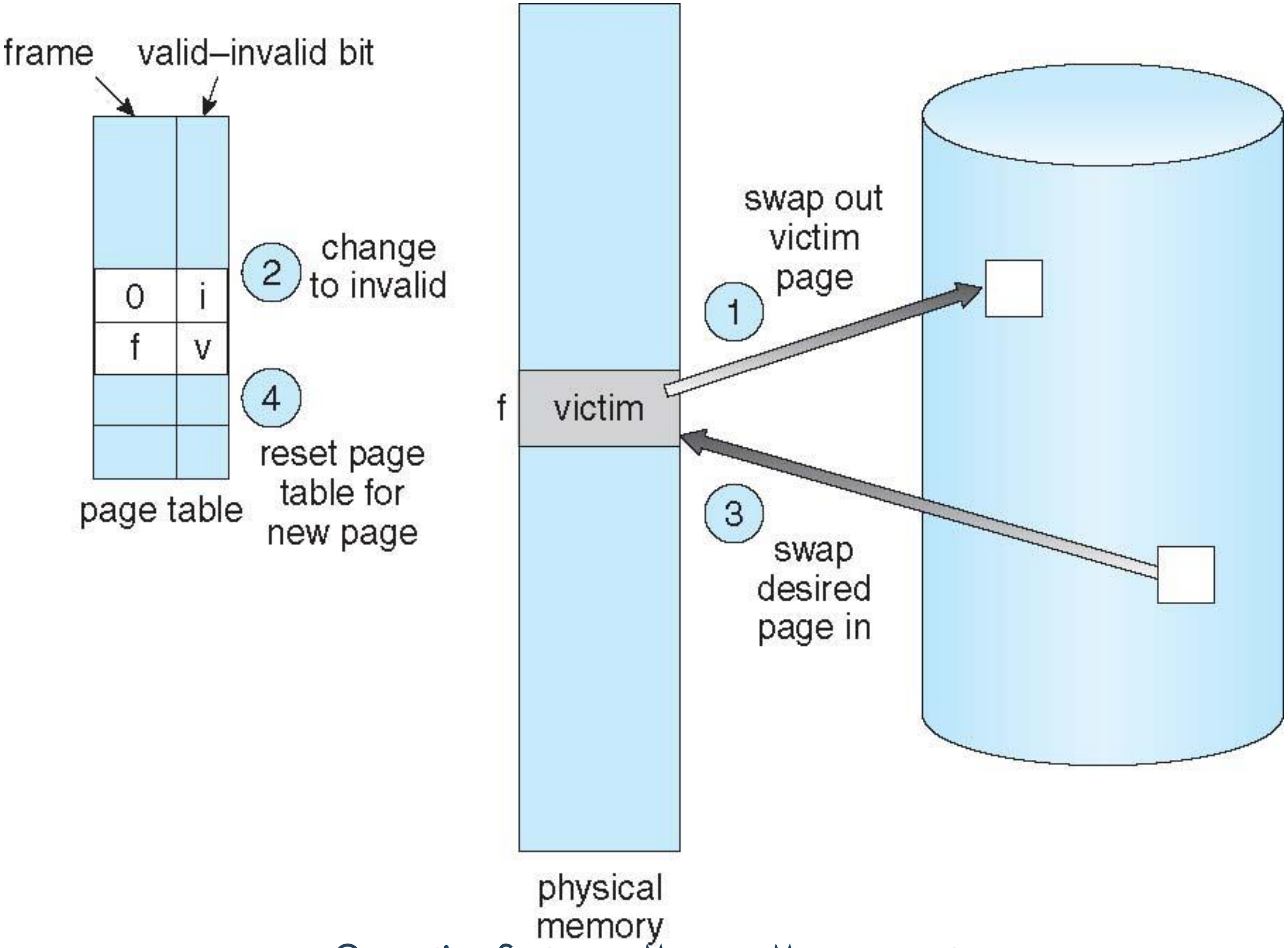
- How much to allocate to each?
  - **Working set:** set of pages or memory locations that a program is actively using at a given time
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty (the page's contents have been "dirtied" or changed since they were last loaded from disk)
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing Estimated Available Time, time to wait for the disk to become available before it can read the page into memory

Page Replacement

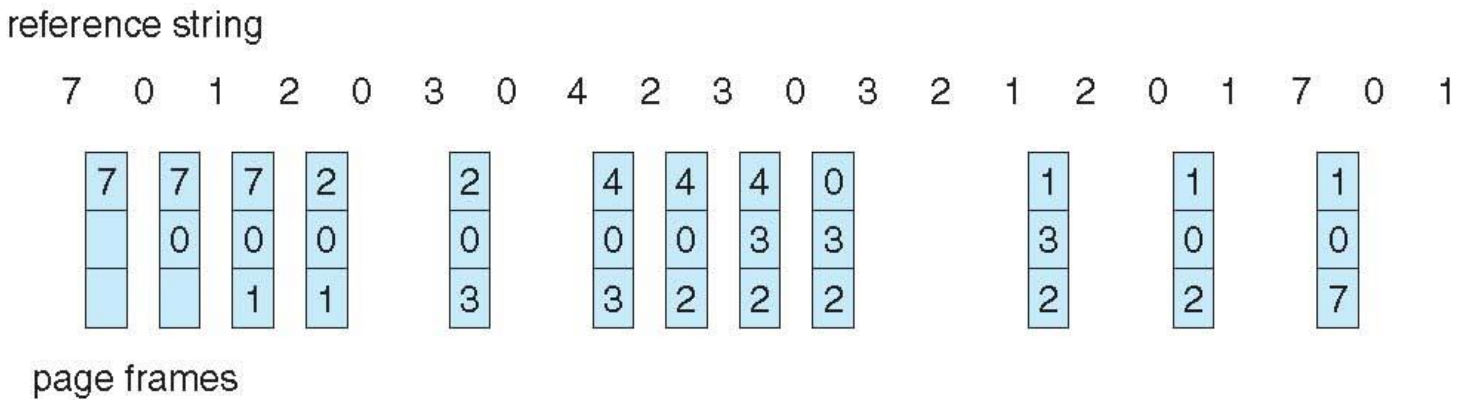


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
  - FIFO, Optimal, Least Recently Used (LRU), Second Chance, …
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

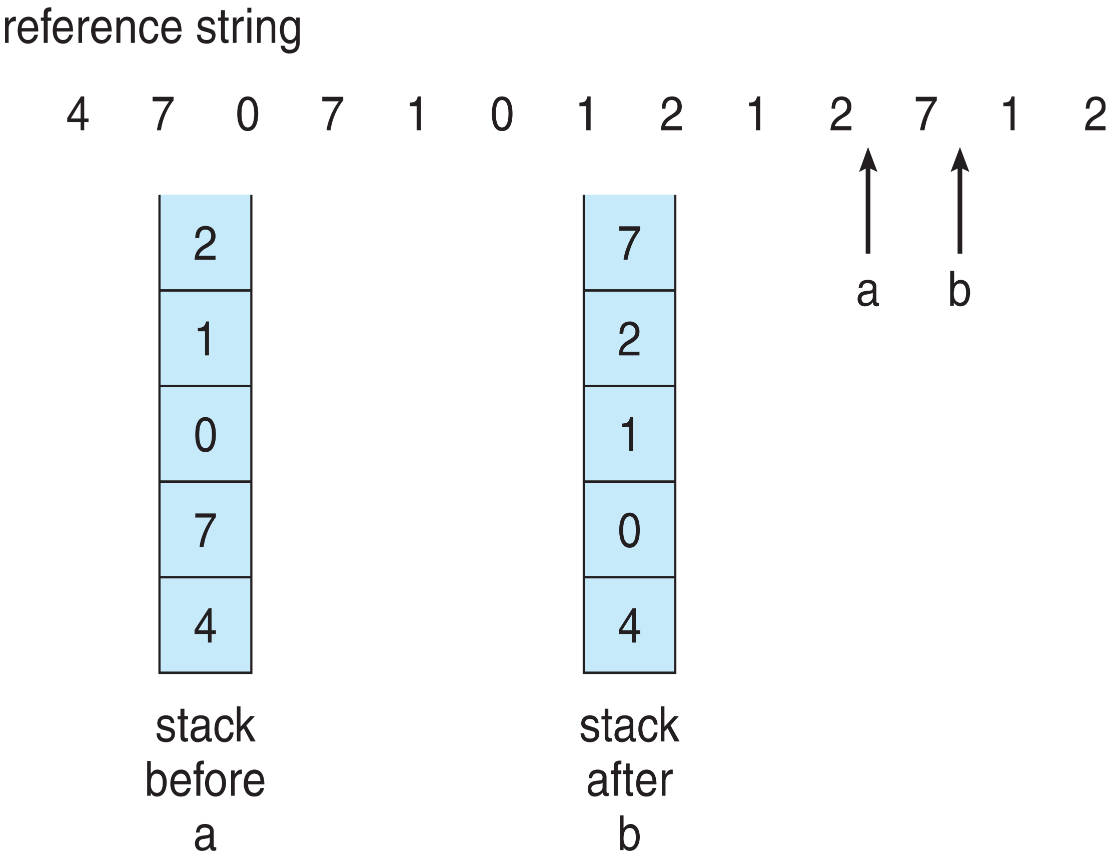


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

Use Of A Stack to Record Most Recent Page References



Working set

- Each process needs a ***minimum*** number of frames to execute at every moment:
  - Working set
- Two major allocation schemes
  - fixed allocation.
    - Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Priority allocation.
    - Use a proportional allocation scheme using priorities rather than size
  - Dynamic allocation
    - Increase/decrease working set following process behavior

Global vs. Local page replacement

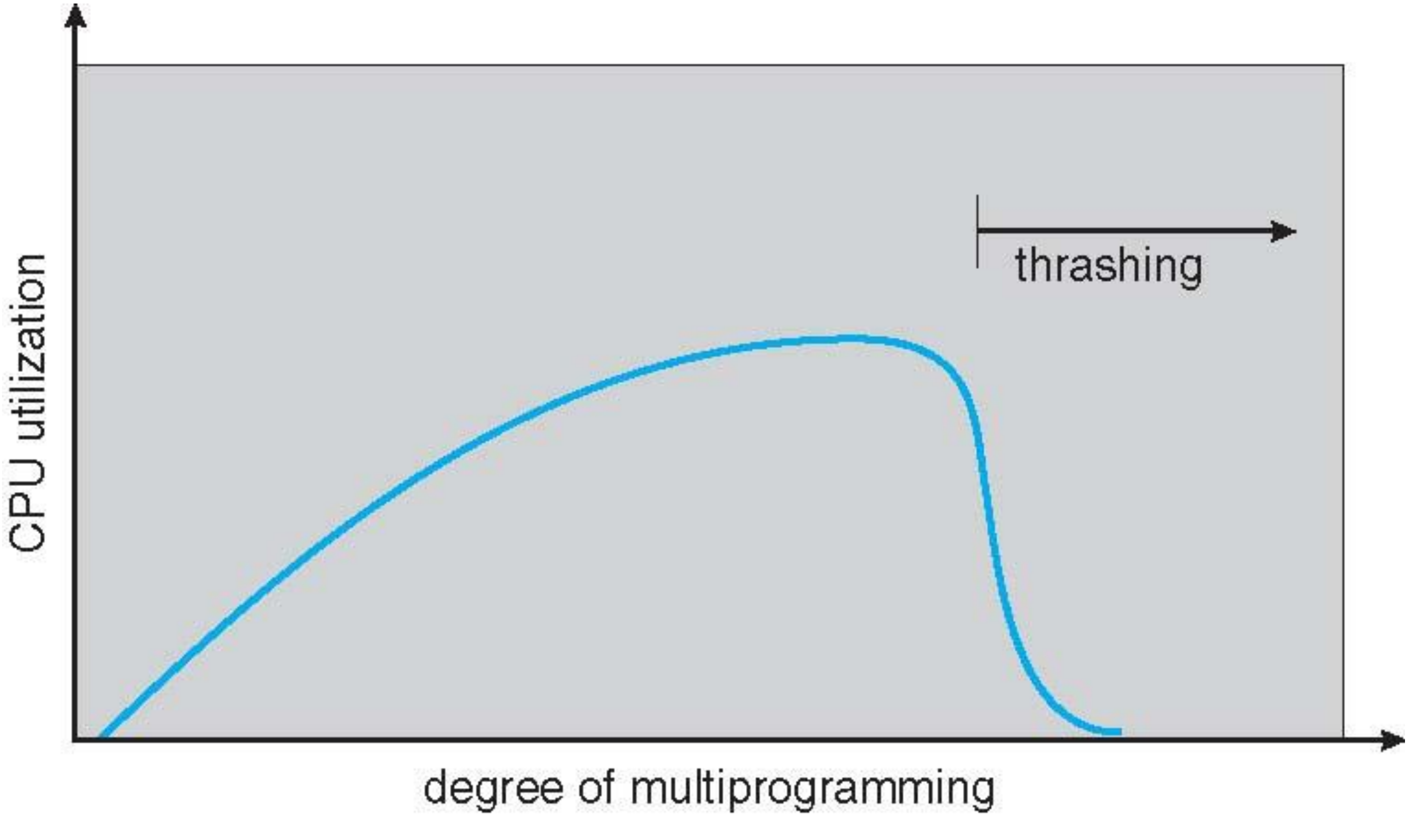
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Contents

- Functions of the memory manager.
- Executable generation and dynamic libraries.
- Virtual memory.
- **Memory manager services.**

Memory management services

- Memory manager performs internal functions.
- Few direct services to applications.

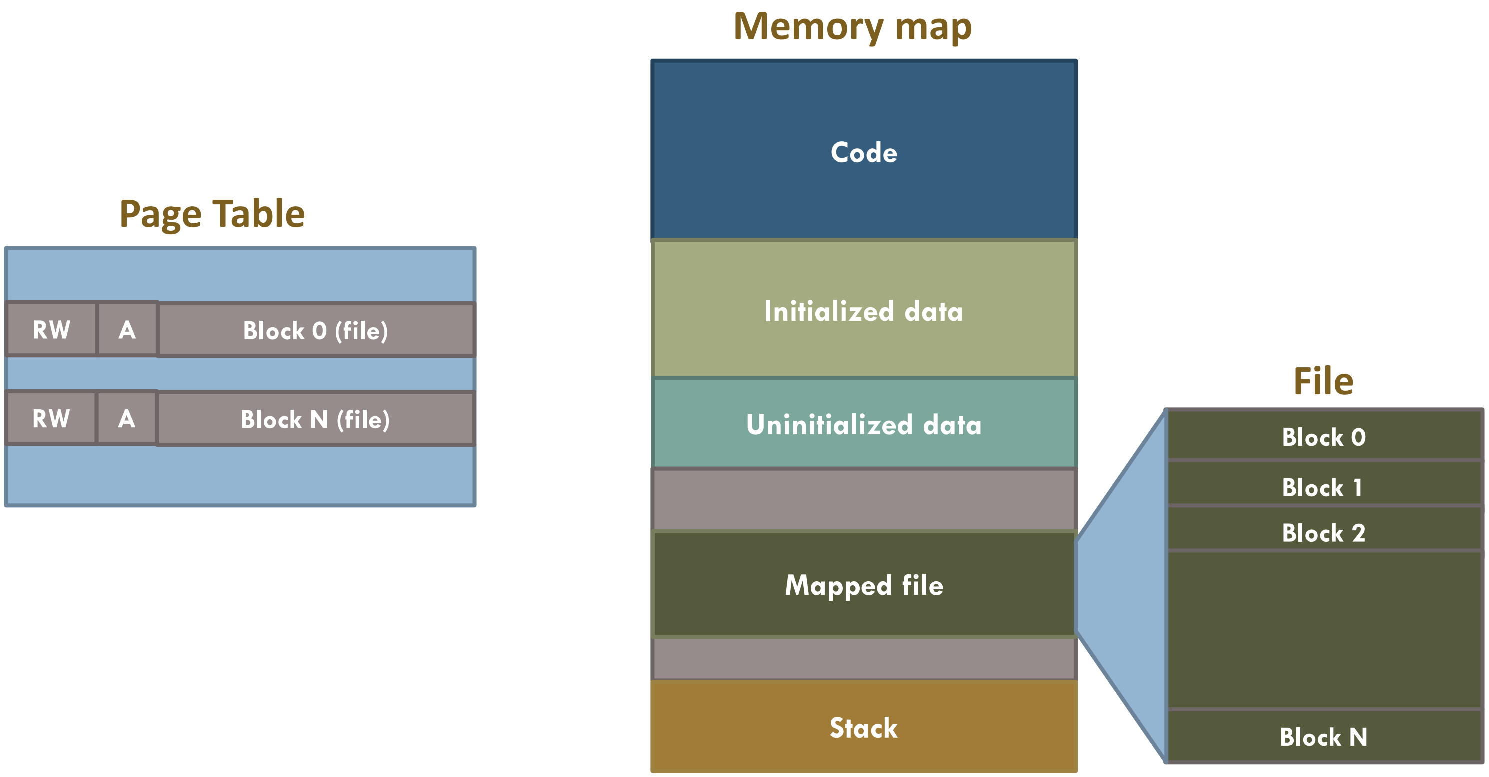
- Services:
  - **POSIX.**
    - File mapping management: mmap, munmap.
    - Dynamic libraries management: dlopen, dlsym, dlclose.
  - **Win32**
    - File mapping management: CreateFileMapping, MapViewOfFile, UnmapViewOfFile.
    - Dynamic libraries management: LoadLibrary, GetProcAddress, FreeLibrary.

Memory mapped files

- Alternate way to access files instead of read/write.
  - Less system calls.
  - Avoid intermediate copies in file system cache.
  - Ease programming as after mapping file is accessed as memory data structures.

- Used for loading dynamic libraries.
  - Code mapped as shared.
  - Initialized data mapped as private.

Memory mapping



Mapping in posix POSIX

void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off);

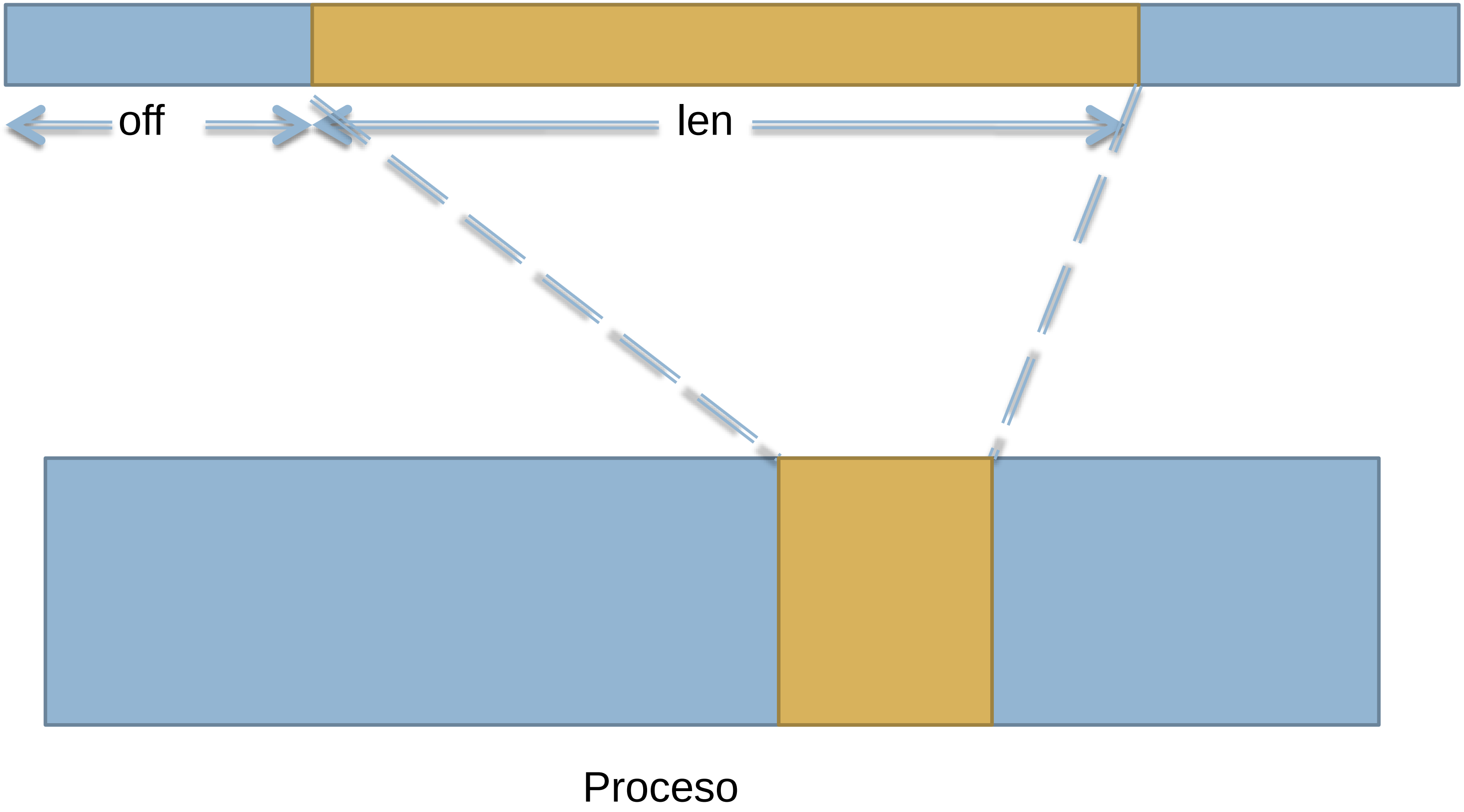
- Established mapping between process address space and file.
  - Returns memory address where file is mapped.
  - **addr**: Address for projection. If NULL, the OS makes a choice.
  - **len**: Specify number of bytes to map.
  - **prot**: Protection for area (may be combined with |).
  - **flags**: Area properties.
  - **fd**: File descriptor to be mapped in memory.
  - **off**: Initial offset on file.

POSIX: mmap

- Protection:
  - **PROT_READ**: Reading allowed.
  - **PROT_WRITE**: Writing allowed.
  - **PROT_EXEC**: Execution allowed.
  - **PROT_NONE**: Data access not allowed.

- Memory region properties:
  - **MAP_SHARED**: Region is shared.
    - Modifications affect to file.
    - Child processes share region.
  - **MAP_PRIVATE**: Region is private.
    - File is not modified.
    - Child processes get non shared duplicates.
  - **MAP_FIXED**:
    - File must be mapped at address specified in call.

POSIX mapping



Unmappin in POSIX

void munmap(void *addr, size_t len);

- Unmanps portion of address space from process.
- From addr to addr+len-1.
- Can be less than initially mapped.

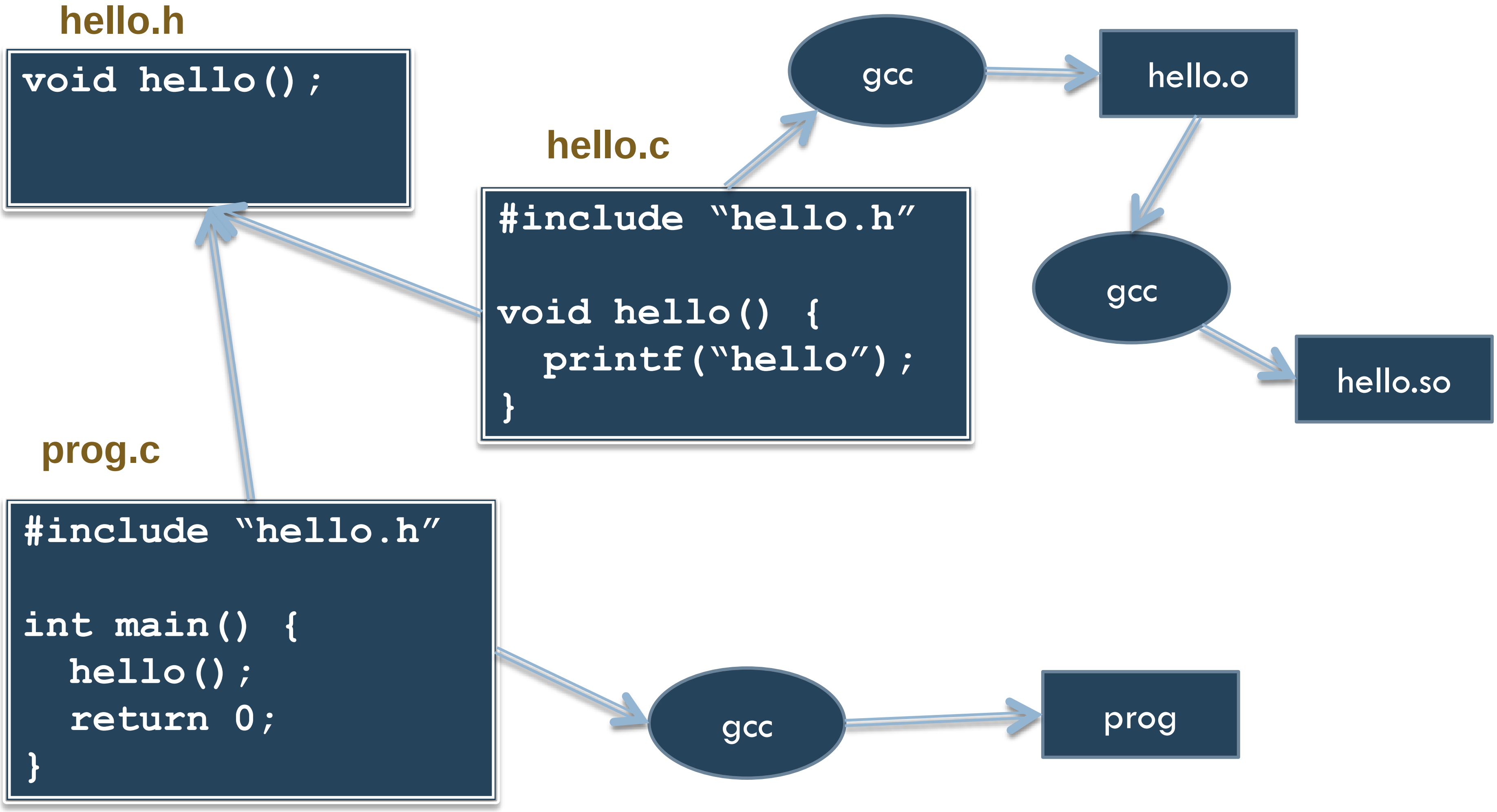
Example: Count number of blanks

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int fd;
  struct stat dstat;
  int i, n=0;
  char c,
  char * vec;

  fd = open(“data.txt”,O_RDONLY);
  fstat(fd, &dstat);
  vec = mmap(NULL, dstat.st_size, PROT_READ, MAP_SHARED, fd, 0);
  close(fd);
  c =vec;
  for (i=0;i<dstat.st_size;i++) {
    if (*c==‘ ‘) {
      n++;
    }
    c++;
  }
  munmap(vec, dstat.st_size);
  printf(“n=%d,\n”,n);
  return 0;
}
```

Example: Copy file

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int i, fd1, fd2;
  struct stat dstat;
  char * vec1, *vec2, *p, *q;

  fd1 = open(“f1”, O_RDONLY);
  fd2 = open(“f2”,
        O_CREAT|O_TRUNC|O_RDWR,0640);
  fstat(fd1,&dstat);
  ftruncate(fd2, dstat.st_size);

  vec1=mmap(0, dstat.st_size,
    PROT_READ, MAP_SHARED, fd1,0);
  vec2=mmap(0, dstat.st_size,
    PROT_WRITE, MAP_SHARED, fd2,0);

  close(fd1); close(fd2);

  p=vec1; q=vec2;
  for (i=0;i<dstat.st_size;i++) {
    *q++ = *p++;
  }

  munmap(vec1, dstat.st_size);
  munmap(vec2, dstat.st_size);

  return 0;
}
```

Dynamic library: generation



hello.h

```
void hello();
```

hello.c

```
#include "hello.h"

void hello() {
  printf("hello");
}
```

prog.c

```
#include "hello.h"

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

Dynamic libraries

- Usually implicit linking with dynamic libraries is enough.

- Explicit linking:
  - Need to write code to load and link symbols from dynamic library.
  - Examples of usefulness:
    - Decide at runtime between two libraries implementing the same API.

Dynamic libraries: POSIX services

void * dlopen(const char * lib, int flags);

- Load a dynamic library and link with current process.
- Return descriptor to be used with **dlsym** and **dlclose**.
- **lib**: Library name.
- **flags**: Options.
  - **RTLD_LAZY**: Deferred resolution of references.
  - **RTLD_NOW**: Immediate resolution of references.

Dynamic libraries: POSIX services

void * dlsym(void * ptrlib, char * symb);

- Returns pointer to a symbol from dynamic library.
  - **ptrlib**: Library descriptor obtained through dlopen.
  - **symb**: String with name of symbol to be loaded.

void dlclose(void * ptrbib);

- Unload dynamic library from process.

Explicit dynamic library loading

```
#include <stdio.h>
typedef void (*pfn)(void);

int main() {
  void * lib;
  pfn func;
  lib = dlopen(“libhello.so”, RTLD_LAZY);
  func=dlsym(bib,”hello”);
  (*func)();
  dlclose(lib);
  return 0;
}
```

SISTEMAS OPERATIVOS: GESTIÓN DE MEMORIA

Gestión de memoria