OPERATING SYSTEMS:

Lesson 9:
Introduction to Memory Management

Jesús Carretero Pérez
David Expósito Singh
José Daniel García Sánchez
Francisco Javier García Blas
Florin Isaila

Goals

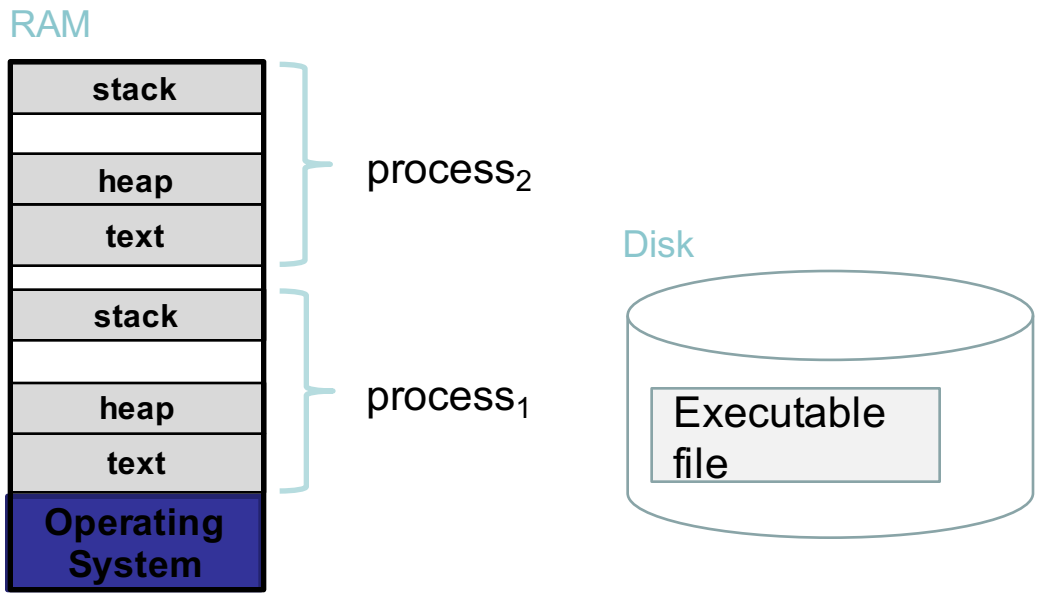
- To know the memory manager functions.
- To know the phases in generating an executable and the structure of a process memory map.
- To understand schemes for contiguous allocation memory.
- To be able to use memory management services for memory mapped files and dynamic libraries.

Contents

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

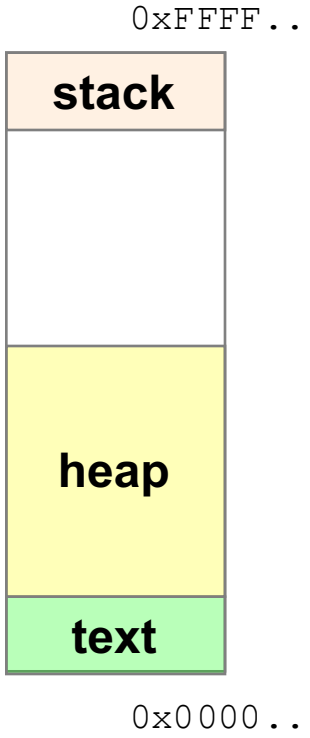
Process Memory Image

- Memory image: set of memory addresses allocated to a program in execution (a process).



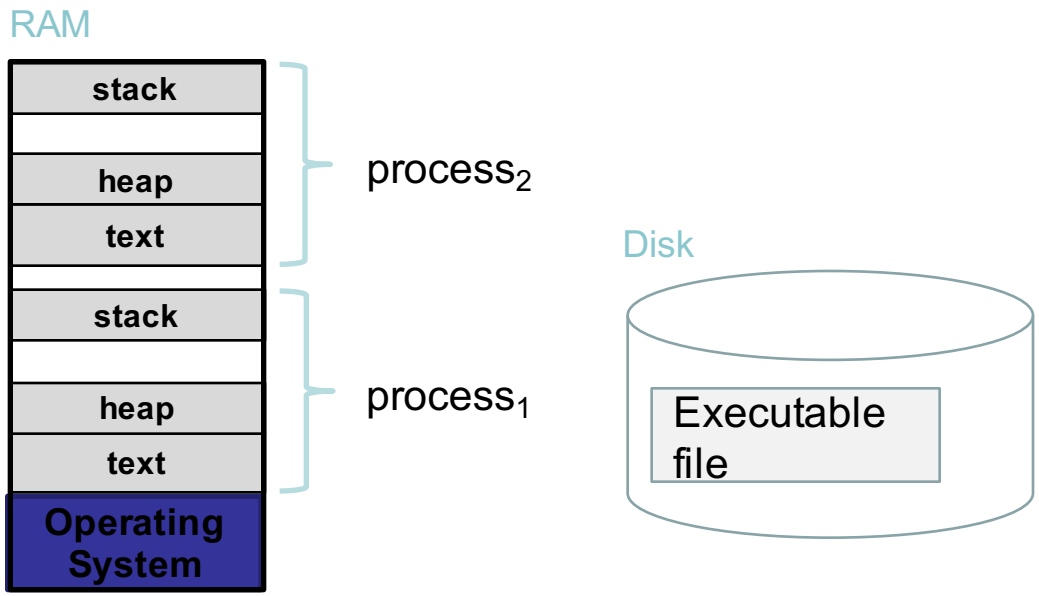
Process Memory Image

- A process is composed of a set of memory regions (*segments*).
- A region is a contiguous zone of memory with the same properties.
- Major properties:
  - Permisions: read, write, exec.
  - Sharing: *private* or *shared*
  - Size (Constant/variable)
  - Initial value (with/without support)
  - Static or dynamic
  - Growing direction (up/down)



Process Memory Image

- Memory image: set of memory addresses allocated to a program in execution (a process).



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 manage regions.
  - To maximize multiprogramming degree.
  - To provide processes with very large memory maps.

Independent logical spaces

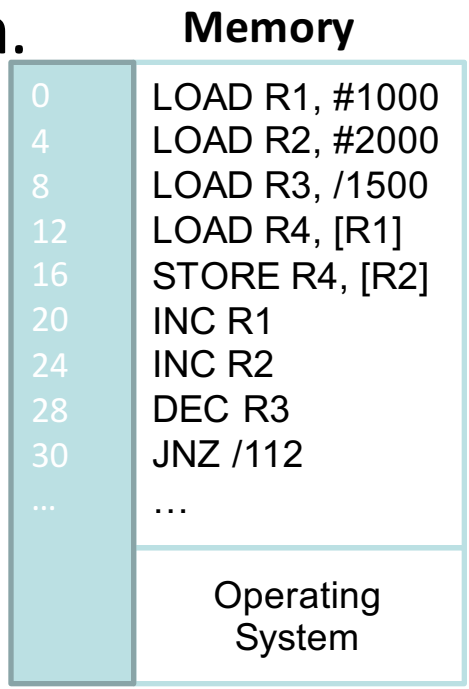
- Program memory location is unknown.
- Executable code generates references between 0 and N.
- Example: Program copying a vector.

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.

Single-programmed Operating Systems

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



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

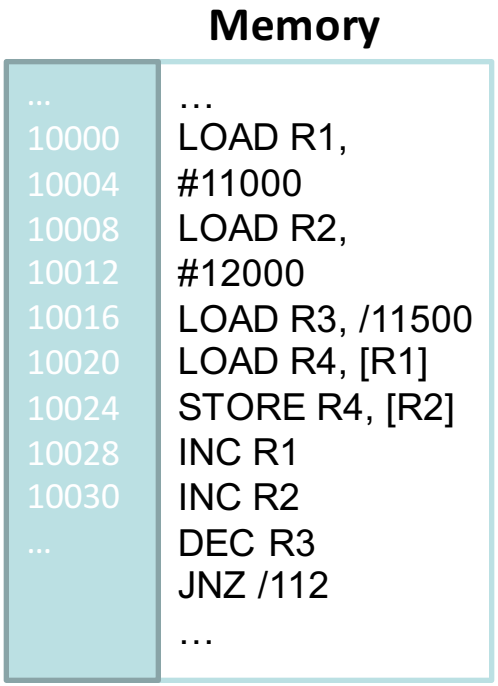
- Relocation creates independent logical space for processes.
  - Operating System must be able to access to processes logical spaces.

- Example: Program has memory allocated starting at address 10000.
  - Add 10000 to all generated addresses.

- **Two alternatives**:
  - Software relocation.
  - Hardware relocation.

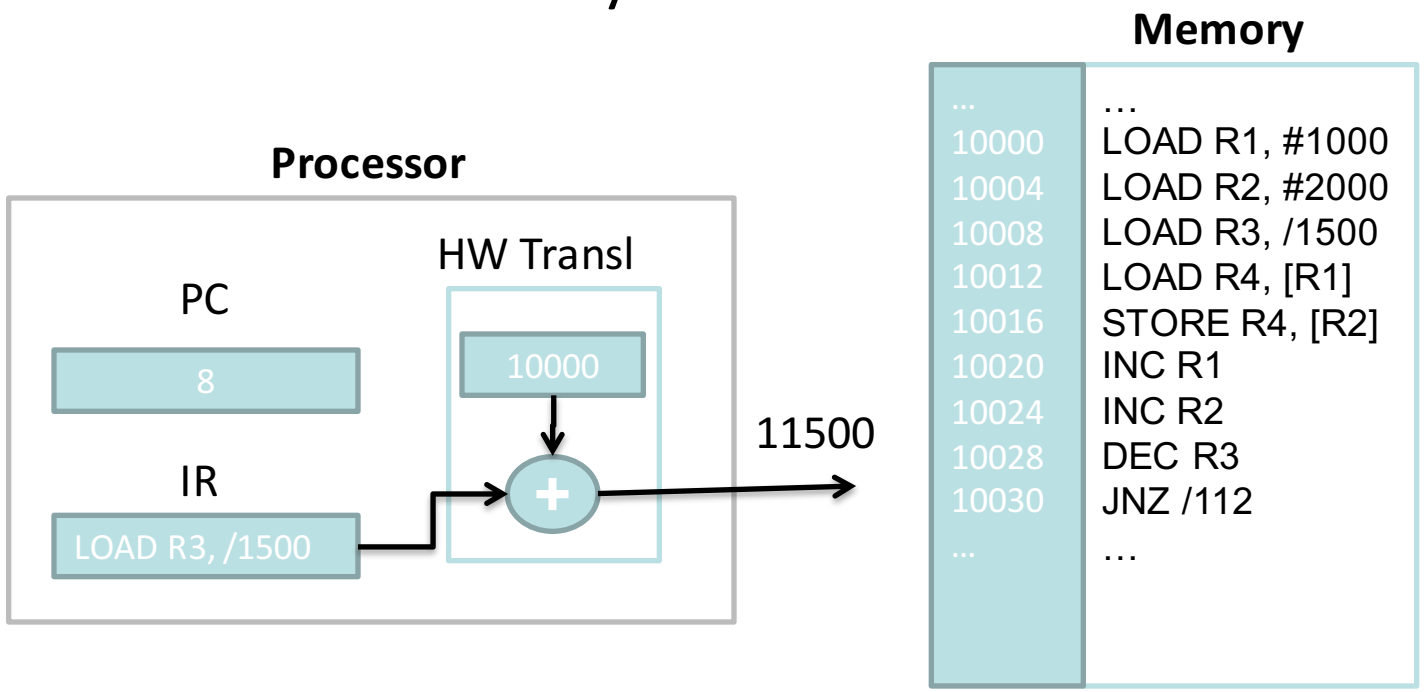
Software relocation

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



Hardware relocation

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



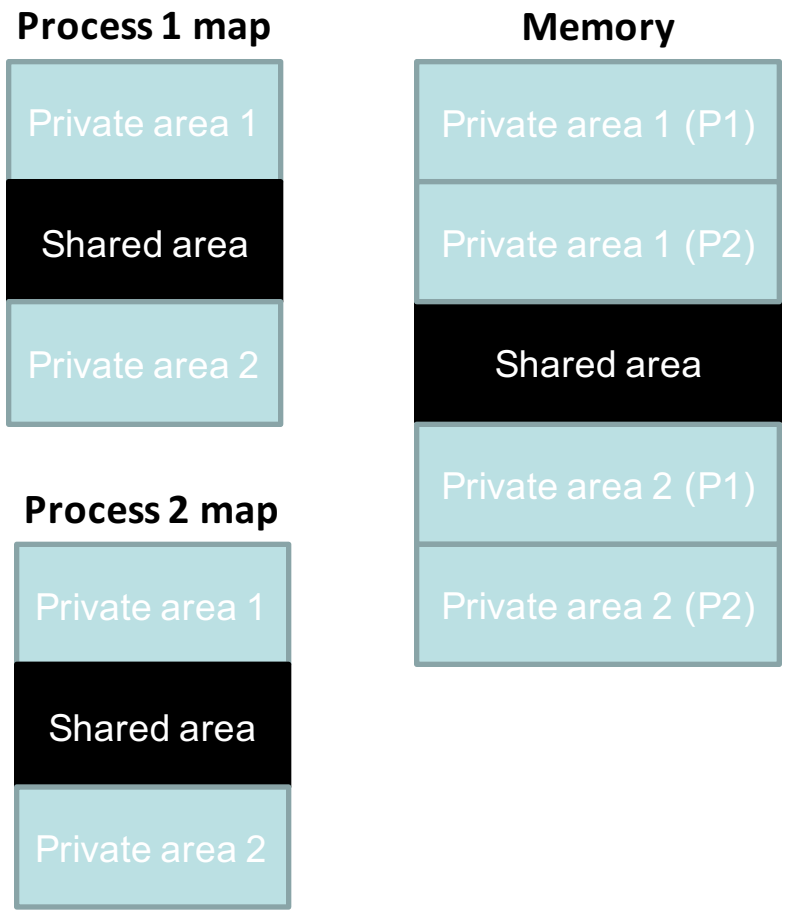
Protection

- **Monoprogramming**: protection for the OS.
- **Multiprogramming**: additional protection among processes.

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

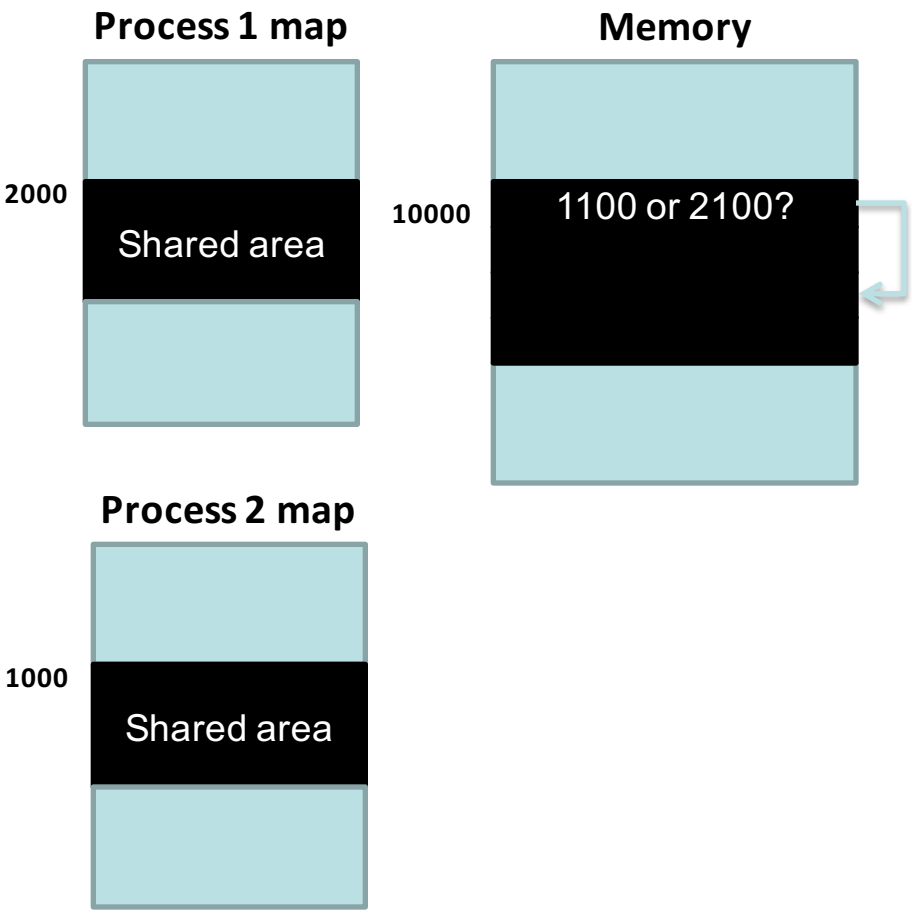
Memory sharing

- Logical addresses from 2 or more processes map to the same physical address.
- Under OS control.
- Benefits:
  - Processes running same program share code.
  - 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.



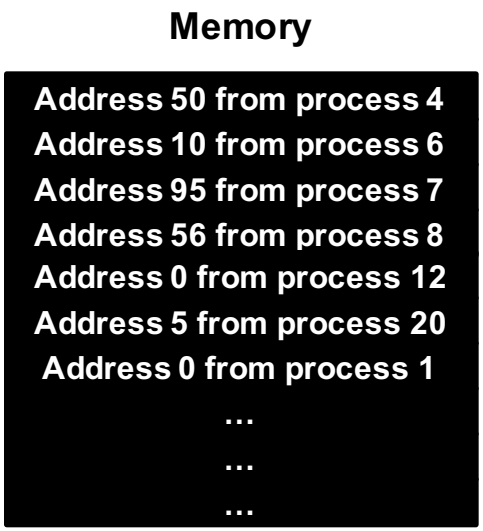
Regions support

- Non homogeneous process map.
  - Set of regions with different characteristics.
  - Example: Non-modifiable region with code and R/W data
- 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:
  - Detect not-allowed accesses to a region.
  - Detect accesses to non-allocated areas.
  - Avoid allocating space for empty areas (holes).
- OS must store a region table for each process.

Performance maximization

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

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



Very large memory maps

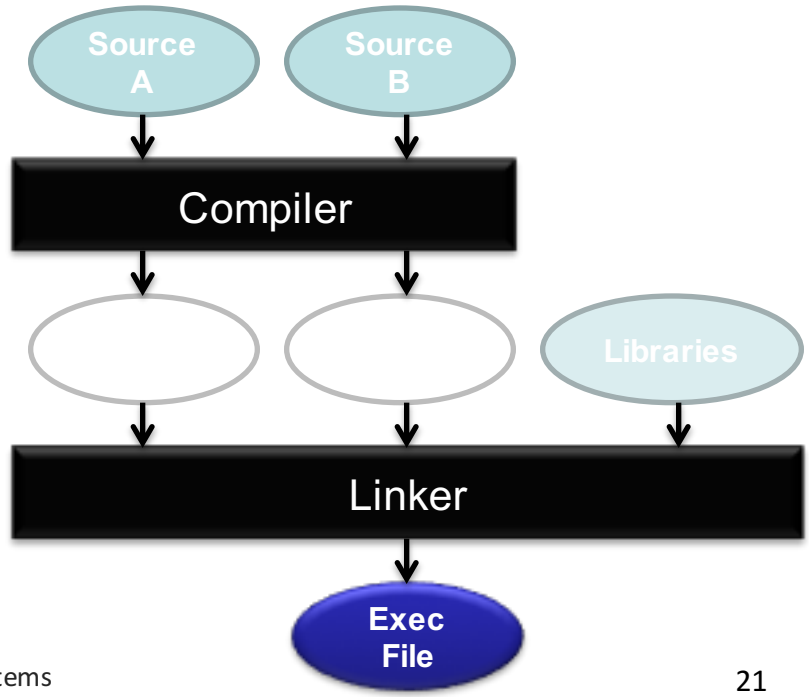
- Processes need larger and larger maps.
  - More advanced applications with new features.
- Solved thanks to virtual memory.
- Historically *overlays* were used:
  - Program divided in phases executed in sequence.
  - In each point of time only a phase is resident in memory.
  - Each phase performs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

Contents

- Functions of the memory manager.
- **Executable generation and dynamic libraries.**
- Executable file format.
- 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 many 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 of instructions performing reference so that next reference access to library symbol.
    - Problem: It would imply modifying program code.
    - Typical solution: Indirect reference through table.

Dynamic libraries: Advantages and drawbacks

- Advantages:
  - Smaller executables.
  - Library routines code only in library file.
  - Processes may share library code.
  - Automatic library update: Versioning.
- Drawbacks:
  - Higher execution time due to load and linking.
    - Tolerable: Pay-off by other advantages.
  - Executable is not self-contained.
- Dynamic libraries use is transparent.
  - Compilation and linking commands similar that for static.
  - Same source code.

Using dynamic libraries

- Usually implicit:
  - Specify at link time libraries to be used but defer loading and linking to run-time.

- Explicit use:
  - Required by applications determining at run-time which library to use.
  - Library not specified in link command.
  - Program requests library through system service.
    - Dlopen in UNIX and LoadLibrary in Win32.
  - Non-transparent access to library symbols.
    - Program needs to use system services.
    - Dlsym in UNIX and GetProcAddress in Win32.

Dynamic libraries sharing

- Dynamic library contains internal references.
  - Problem in shared area with auto-references.
- Three possible solutions:

1. Each library is assigned to a fixed address range.
   - **Drawback**: Low flexibility.

2. Link phase: at run-time the auto-references are adjusted.
   - **Drawback**: Does not allow for sharing library code.

3. Create a library with Position Independent Code (PIC).
   - Code generated with register relative addressing.

   Drawback (although tolerable): Relative addressing is less efficient.

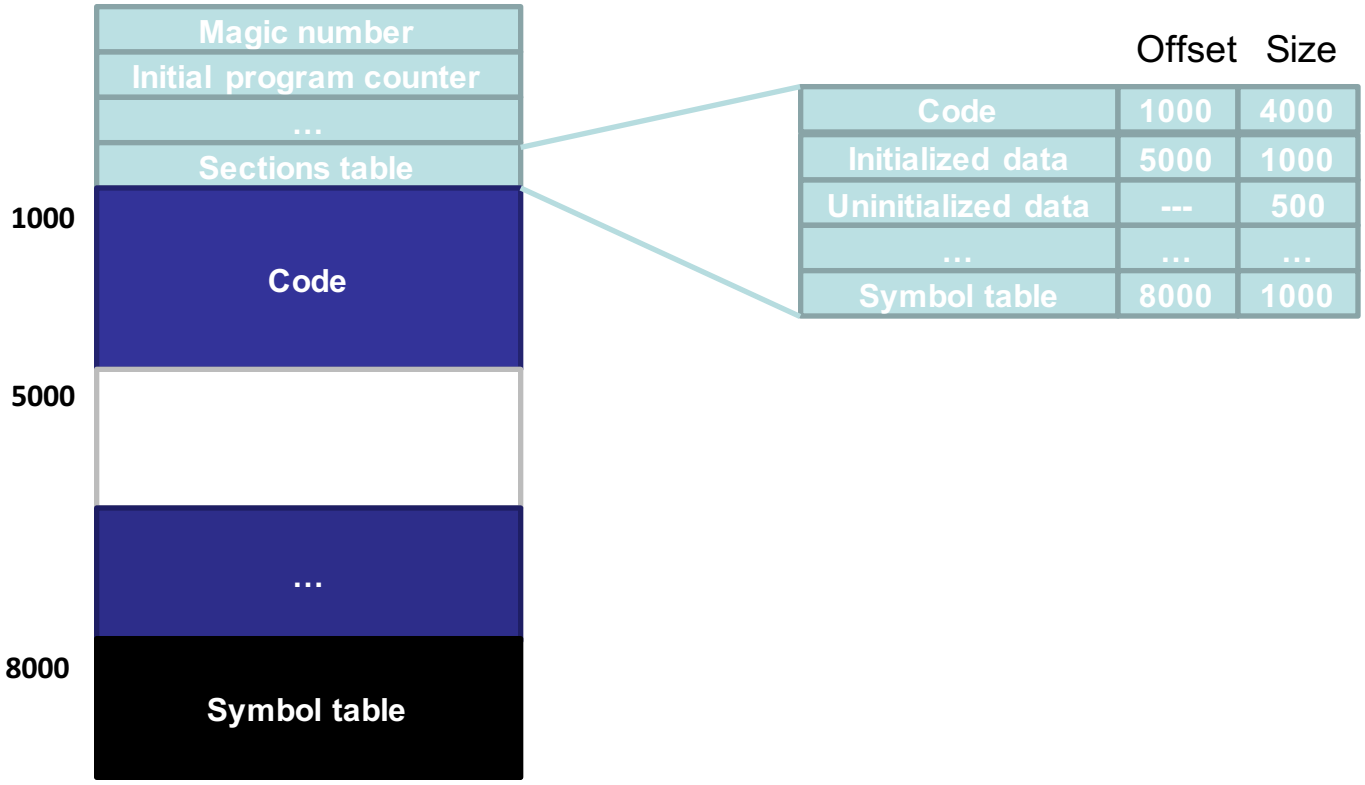
Contents

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

Executable format

- Different vendors use different formats.
  - Example: Linux -> Executable and Linkable Format (ELF).
- Structure: Header and sequence of sections
- Header:
  - Magic number identifying executable (0x7f454c46).
  - Program entry point.
  - Sections table.

Executable format



Executable section

- Variety in sections types.
  - Examples:
    - Symbol table for debugging.
    - List of dynamic libraries being used.
- Most relevant sections in process memory map:
  - Code (text).
    - Contains program code.
  - Initialized data.
    - Global variables with initial value.
  - Non-initialized data.
    - Global variables without an initial value.
    - In sections table but it is not stored in executable
  - What about a section for local variables?
    - Not needed. Stack and/or records.

Global and local variables

- **Global variables**:
  - Static.
  - Created on program initiation.
  - Exist during full lifetime of program.
  - Fixed address in memory and executable.

- **Local variables and parameters**:
  - Dynamic.
  - Created on function invocation.
  - Destroyed on return.
  - Address computed at run-time.
  - Recursive functions: multiple instances from a local variable.

Process memory map

- Memory map or process image:
  - Set of regions.

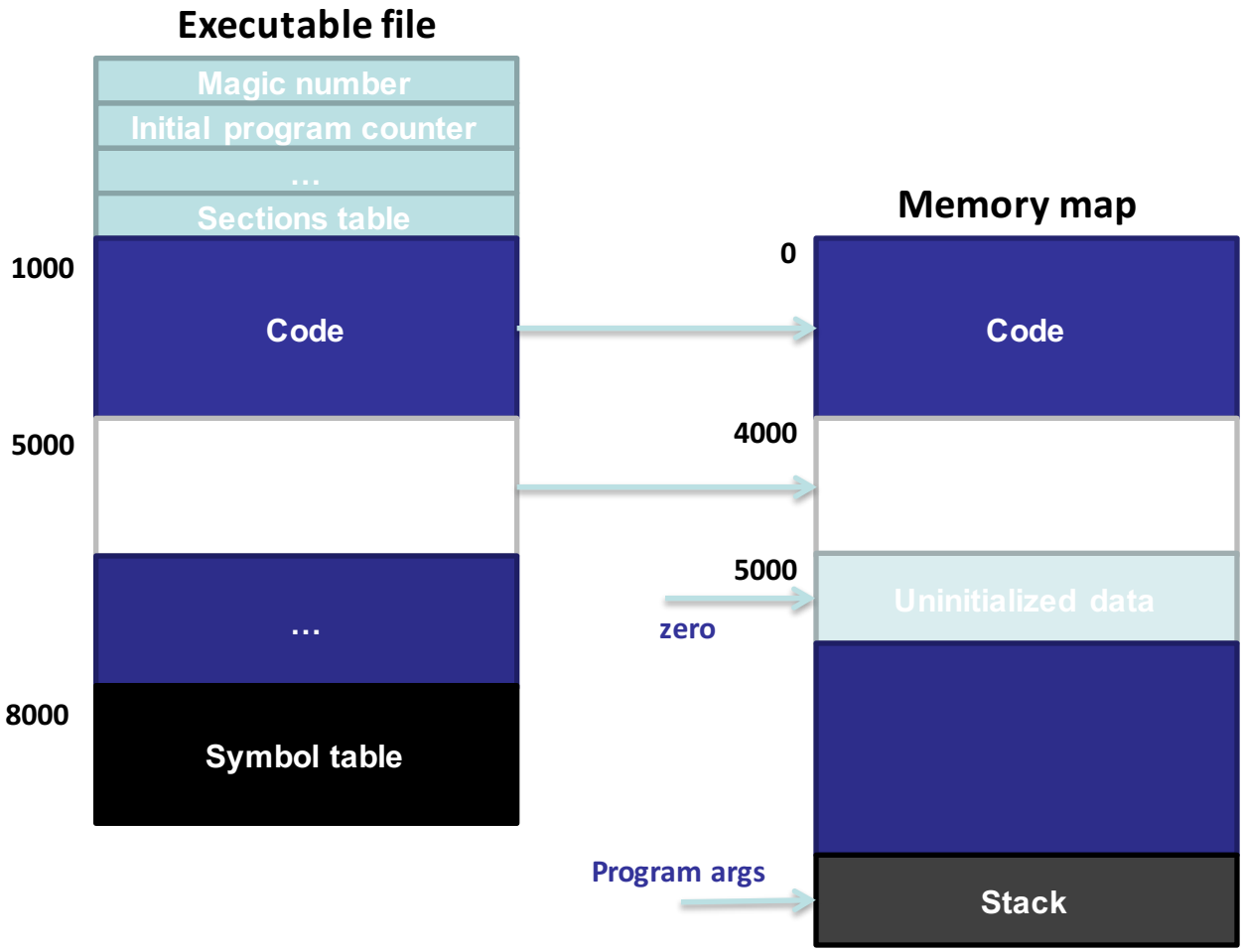
- **Region**:
  - Has an associated information (a *memory object*).
  - Contiguous area treated as unit to be protected or shared.

- Each region characterized by:
  - Start address and initial size.
  - Support: where its initial content is stored.
  - Protection: RWX.
  - Use: Shared or Private.
  - Size: Fixed or variable.

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.

- **Memory mapped file**.
  - Region associated to mapped file.
  - Variable size, Support in file.
  - Protection and use (shared/private) specified in mapping.

Other regions

- **Shared memory**.
  - Region associated to shared memory area.
  - Shared, Variable size, without support (fill with zero).
  - Protection specified in mapping.

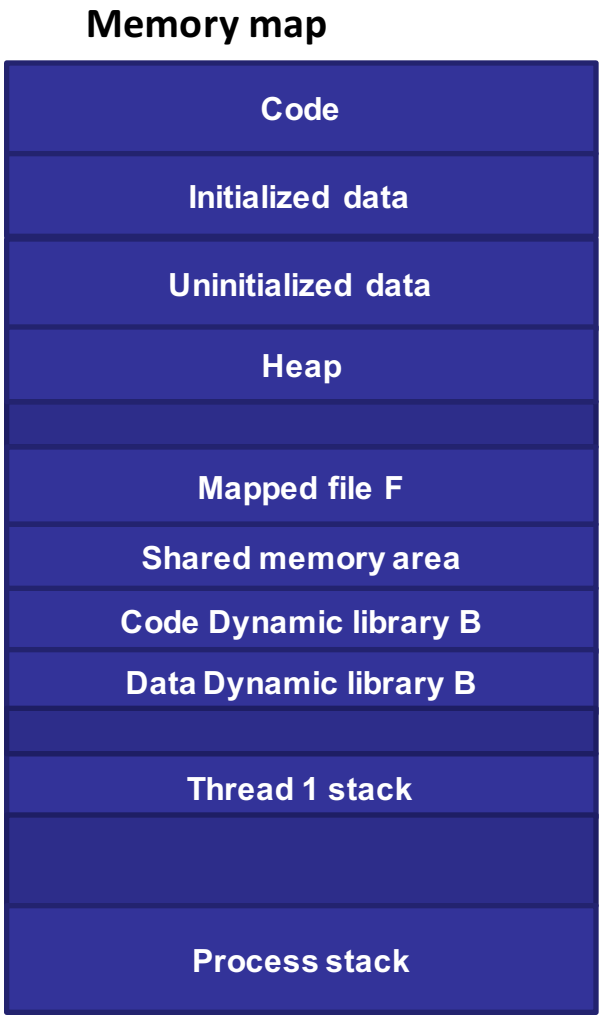
- **Thread stacks**.
  - Each thread corresponds to a region.
  - Same characteristics as process stack.

- **Dynamic library loading**.
  - Regions created associated to library code and data.

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

A possible memory map



Operations on regions

- To study evolution of memory map three operations may be identified:
  - **Create region**.
    - Implicitly on initial map creation.
    - By program request at runtime (e.g. map file).
  - **Eliminate region**.
    - Implicitly on process termination.
    - By program request at runtime (e.g. unmap file).
  - **Change region size**.
    - Implicitly for stack.
    - By program request for heap.
  - **Duplicate region**.
    - Required by FORK service.

Contents

- Functions of the memory manager.
- Executable generation and dynamic libraries.
- Executable file format.
- **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

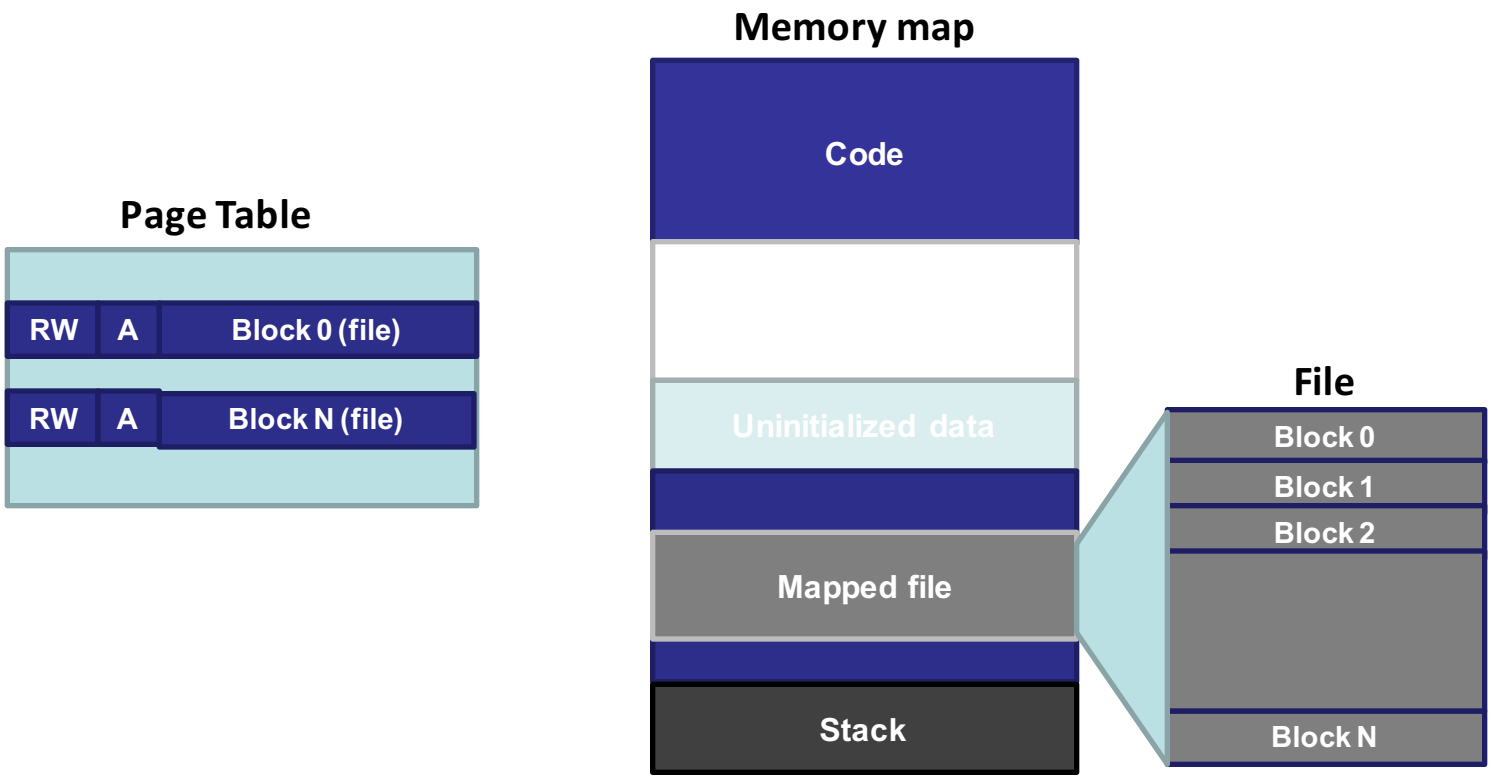
- Generalization of virtual memory.
  - Entry in PT reference a user file.
- Program requests mapping a file (or portion) in its own map.
  - May specify protection and use (shared/private).
- OS completes corresponding entries with:
  - Non resident, load from file.
  - Private/Shared and protection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

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);

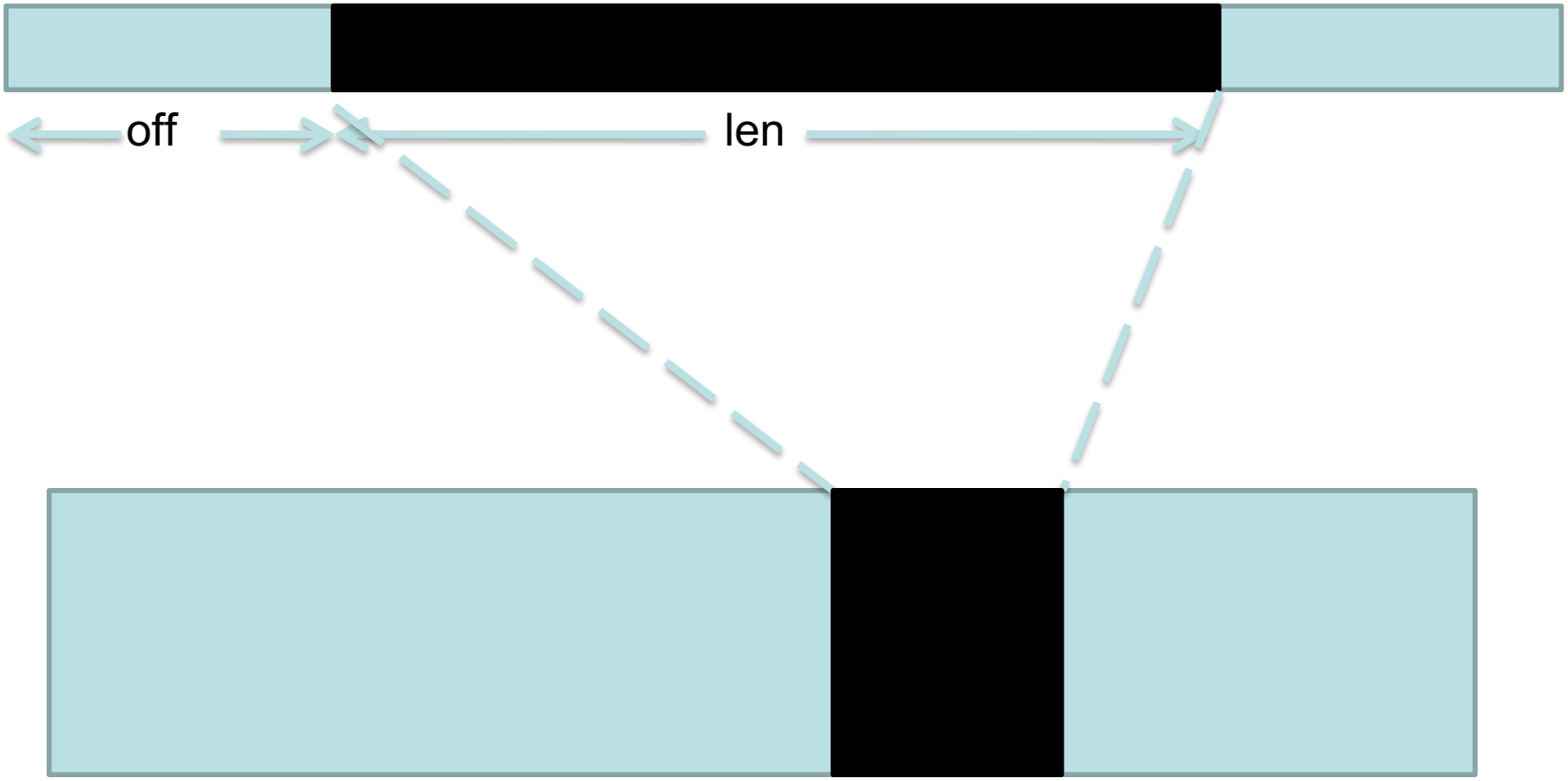
- Established mapping between process address space and file.
  - Returns memory address where file is mapped.
  - **addr**: Address for projection. If NULL, the OS makes a choice.
  - **len**: Specify number of bytes to map.
  - **prot**: Protection for area (may be combined with |).
  - **flags**: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

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

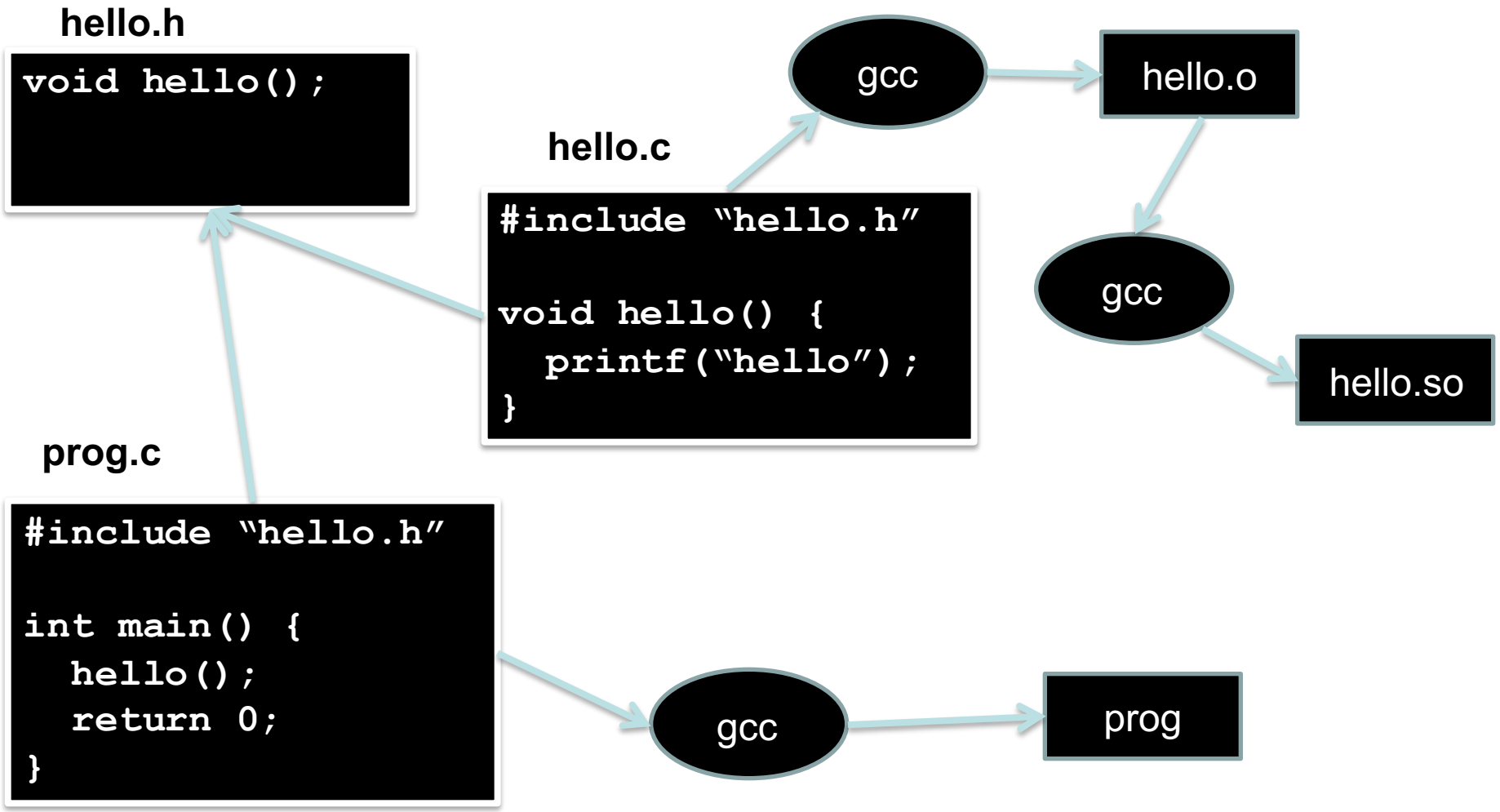
Example: Count number of blanks


```
#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,064
     0);
  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



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.

Implicit load of a dynamic library

```
#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;
}
```

OPERATING SYSTEMS:

Lesson 9:
Introduction to Memory Management

Jesús Carretero Pérez
David Expósito Singh
José Daniel García Sánchez
Francisco Javier García Blas
Florin Isaila