Operating Systems

Complutense University of Madrid

2020-2021

Unit 5: Memory Management

Juan Carlos Sáez

Contents

Contents

Memory management requirements

The OS shares resources among processes

- Each process "believes" it is running alone on the system
- Process management: share the CPU among processes
- Memory management: share memory resources among processes

Typical requirements:

- Offer a private logical address space to each process
  - Guarantee protection among processes
  - Support for regions in the process memory image
- Allow processes to share memory (if they wish to)
- Maximize memory utilization
- Maximize degree of multiprogramming
- Support for huge process memory images

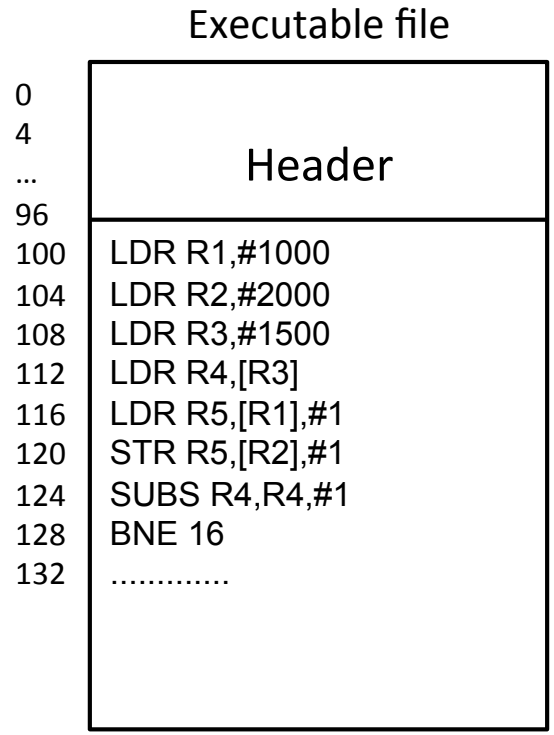
Private logical address spaces

- The memory address where a program will be loaded is often unkown at compile time
- Code in executable file generates *logical addresses* ranging from 0 to N
- Example: Executable file of a program that copies an array into another

Example

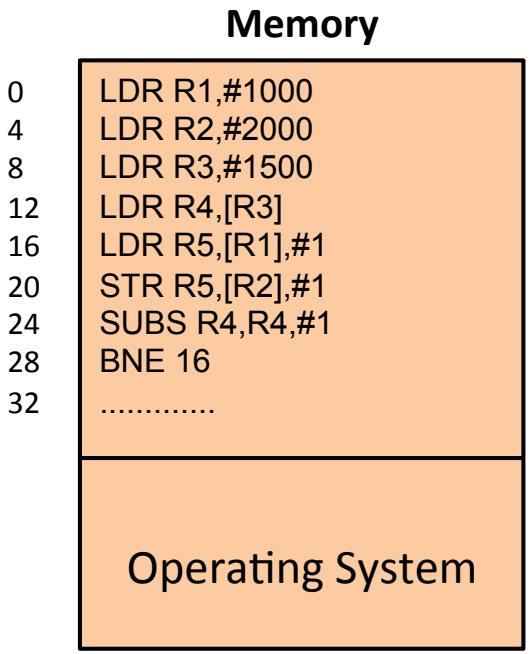
```
for (i=0; i < size; i++)
    B[i]=A[i];
```

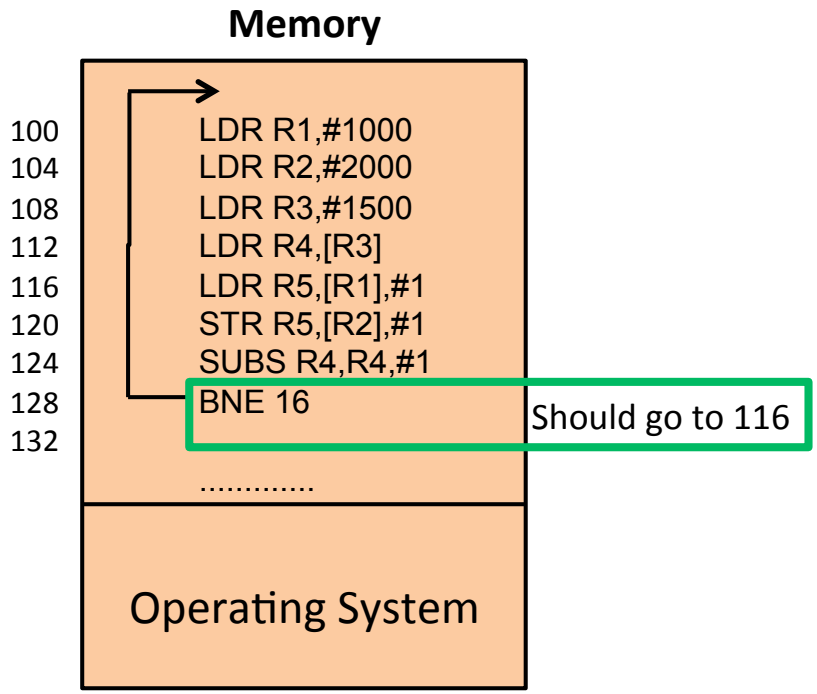
- Header size: 100 bytes
- Each assembly instruction occupies 4 bytes
- Origin array (`A`) starts at address 1000
- Destination array (`B`) starts at address 2000
- Array size in address 1500



Execution in a single-process OS

- The OS's code can be loaded in the upper address range (highest memory addresses)
- Program can be loaded in the lowest memory addresses
  - If it cannot be loaded starting at address 0, memory references must be readjusted



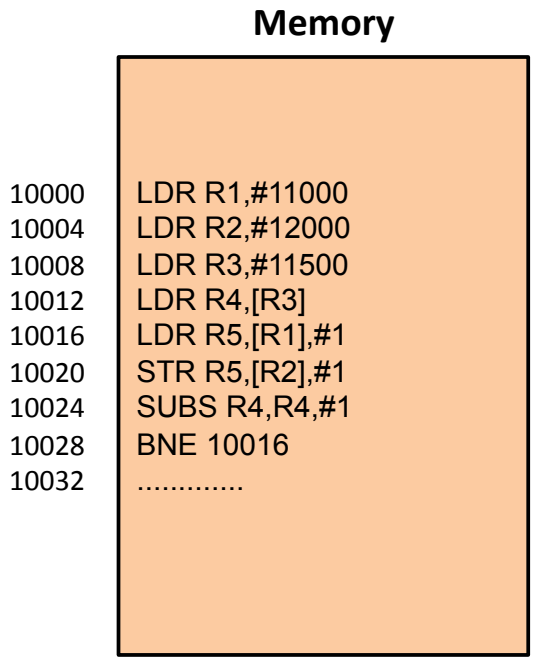


Relocation

- Translation of logical addresses into physical addresses
  - Logical addr.: memory address generated by the program
  - Physical addr.: address in the main memory range
- Required in multitasking-enabled OSes:
  - $Translation(PID, logical\_address) \rightarrow physical\_address$
- Relocation creates a private logical address space for each process
  - OS must be able to access the logical address space of any process (while the process is running)
- Two relocation schemes:
  - Software
  - Hardware

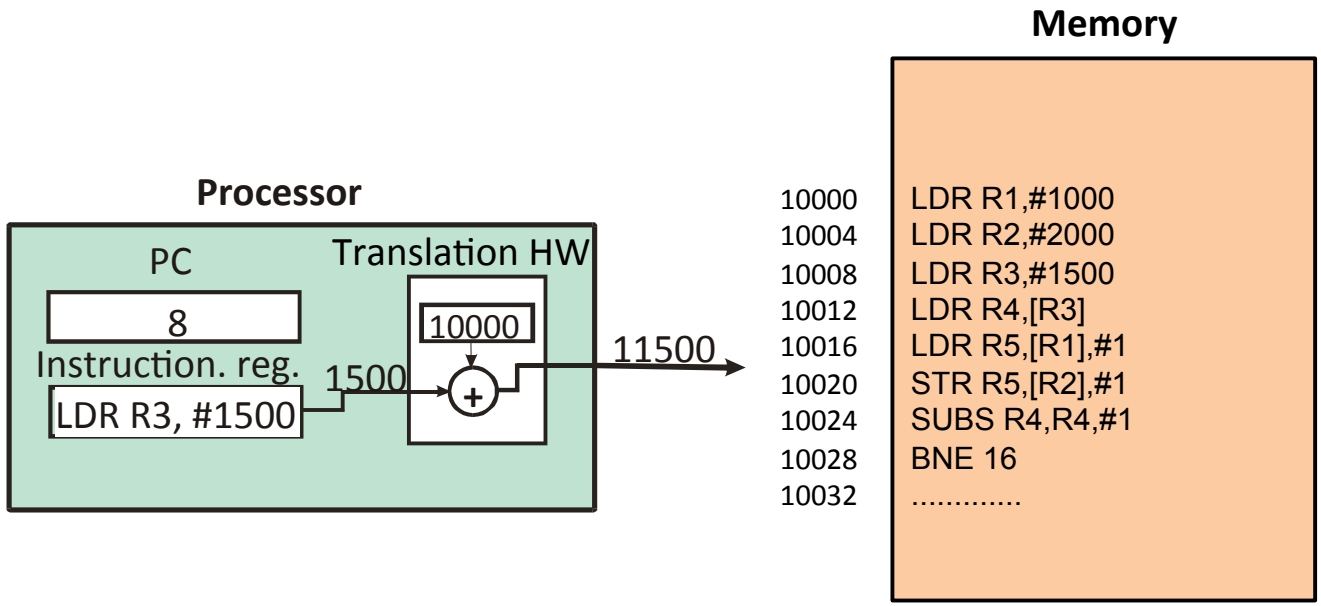
Software relocation

- Address translation performed when program is loaded into memory
- Code in memory different to that stored in the executable file
- Drawbacks:
  - By itself, it does not ensure protection
  - It does not allow to move the program to different memory locations at run time



Hardware relocation

- The hardware (MMU - *Memory Management Unit*) translates addresses at run time
- The program's code in the executable file is loaded into memory *as is*
  - The program/processor generates logical addresses
- OS in charge of:
  - "Maintaining" the translation function for each process
  - Tell the hardware which function must be applied for each process



Protection

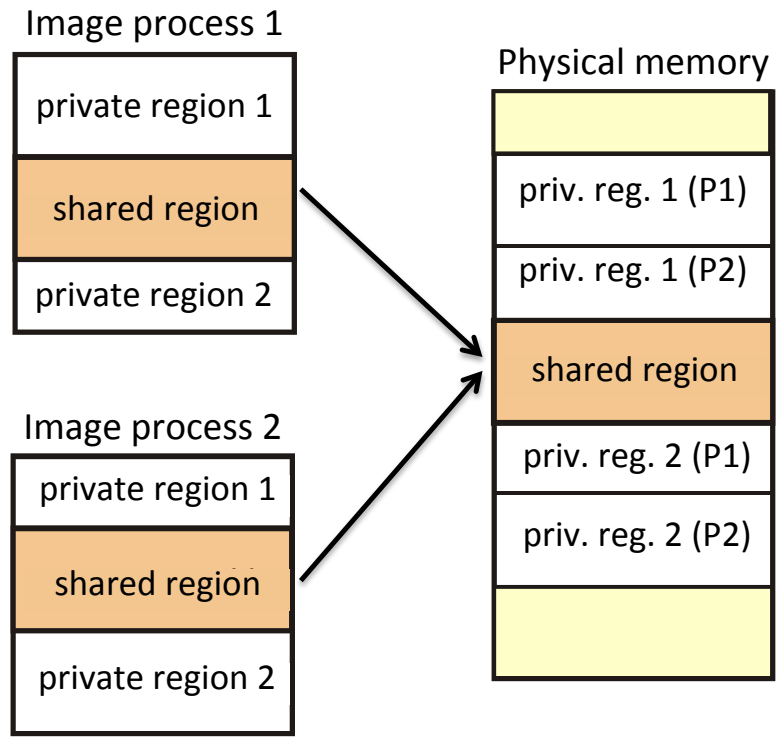
- Single-process environment: Protect the OS
- Multitasking environment: Also protect processes from one another

Minimal requirements:

1. The translation function must create non-overlapping logical address spaces for different processes
2. Every single address generated by running programs must be validated
   - Validation performed by HW $\rightarrow$ raises exceptions when necessary
   - The OS takes care of exception handling

Shared memory

- Logical addresses from 2 or more processes correspond to the same physical address
- The OS manages shared memory
- Benefits of shared memory:
  - Very efficient/fast communication mechanism among processes
  - Processes running the same program may share the code (avoid wasting space)
    - The OS performs this optimization automatically if the underlying memory allocation scheme allows it
- Requires non-contiguous memory allocation



Support for regions

- Process memory image is not homogeneous
  - Set of regions with different features
    - Code, stack, data, ...
  - Example: Code region cannot be modified
- Memory image is dynamic
  - Regions may need resizing at run time (e.g., stack)
  - Regions can be created/destructed
  - Non-allocated regions exist (*gaps*)
- OS must maintain a table of regions for each process
- Memory manager must provide the necessary support for regions:
  1. Detect not allowed accesses to a region
  2. Detect accesses to gaps
  3. Maintain gaps in the memory region so that other regions can grow

Optimize memory usage

- Maximize the degree of multiprogramming
- Some memory is typically "wasted" due to:
  - Unusable remains (fragmentation)
  - Tables for the memory manager
- Good trade-off → Paging
  - Virtual memory enables to increase the degree of multiprogramming
  - Bigger tables (bigger pages) lead to lower fragmentation

Optimal, yet unfeasible, memory allocation scheme

0	Address #50 from process 4
1	Address #10 from process 6
2	Address #95 from process 7
3	Address #56 from process 8
4	Address #0 from process 12
5	Address #5 from process 20
6	Address #0 from process 1
	.....
	.....
N-1	Address #88 from process 9
N	Address #51 from process 4

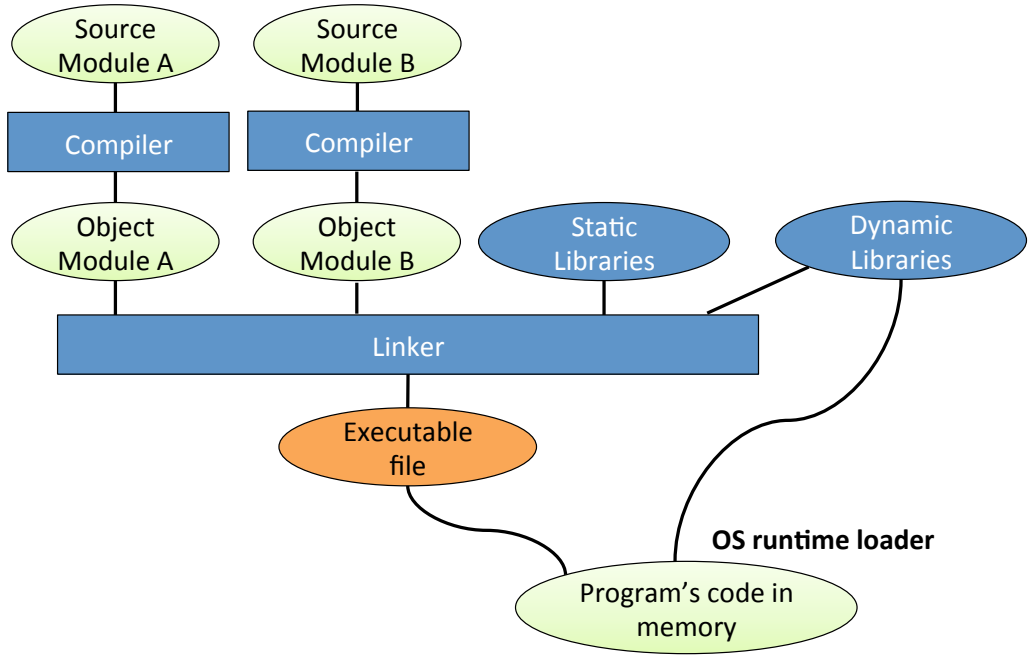
Huge process memory images

- Modern applications require bigger and bigger memory images
- Issue solved thanks to **Virtual Memory**
  - Size of process memory image can exceed the amount of physical memory available in the machine
- In the old days, the *overlaying* technique was used:
  - Program divided into phases (*modules*) executed one after another
  - Only one module is loaded in memory at a time
  - Each module performs its associated processing and then loads the next module into memory
  - Not transparent to the programmer $\rightarrow$ labor-intensive task

Contents

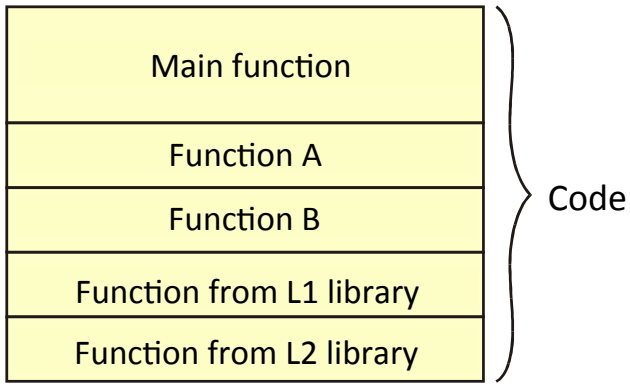
Building an executable file

- Application: set of modules typically written in a high-level language
- Build executable for application: Compilation + Linking
- Compilation:
  - Generates object module
  - Resolves references within each source module
- Linking:
  - Resolves references across object modules
  - Resolves references to library symbols
  - Generates final executable file



Static libraries

- Collection of related object modules that export a set of global symbols (functions, variables, ...)
- **Static linking**: the *linker* creates a self-contained executable by linking the object modules from both the program and the library



Drawbacks of static linking

- Big executable files
- Library code replicated in multiple executable files and in memory
- Executable file must be rebuilt every time the library is updated

Dynamic libraries

- Linking and load at run time
  - For each dependant dynamic library, the executable contains:
    1. Name of the library
    2. Load routine to be executed at run time
  - The load routine is invoked right before the first time a library symbol is referenced

Advantages

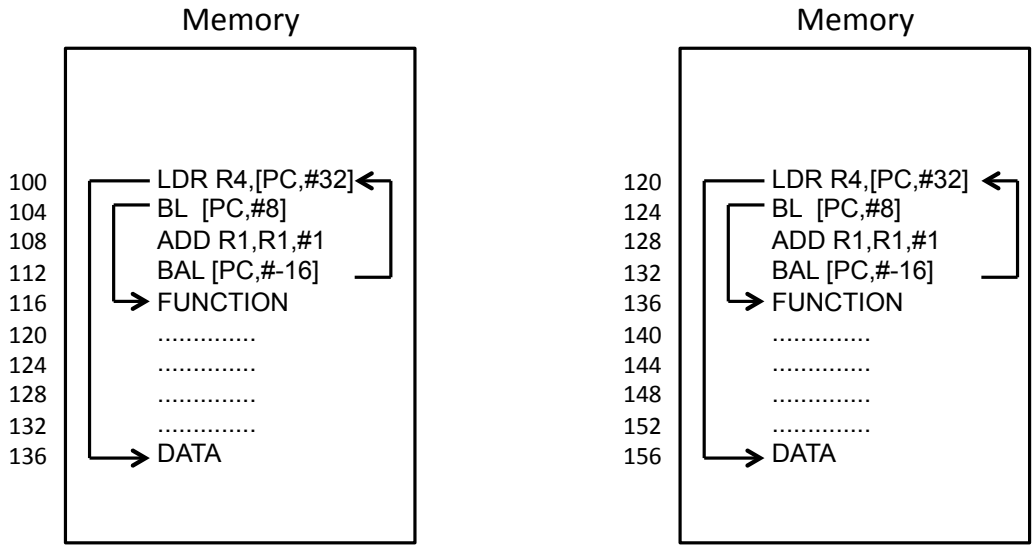
- Reduced executable size
  - Code of library functions stored in library file only
- Processes share code of the same dynamic library
- Changes made to the library that do not alter the library's ABI do not require rebuilding executables

Disadvantages

- Executable is not self-contained
- Increases execution time due to loading and linking
  - Acceptable, advantages totally pay off

Sharing dynamic libraries among processes

- A shared library may contain internal references (e.g. `f()` calls `g()`)
- How to reference private variables and functions inside the library?
  1. Reserve a fixed logical address range for the library beforehand
     - Not a flexible approach
     - Assigning non-overlapping ranges to different libraries may be difficult
  2. Let any library to be assigned different logical address ranges in different processes
     - Makes loading multiple libraries in a process memory image easier
     - Position-Independent Code (PIC) must be generated for the library



Reminder: useful commands

Commands

- `/sbin/ldconfig -p`
  - Shows a list with all loaded libraries
- `ldd`
  - Print shared library dependencies

Dynamic libraries: `ldd` Example

greetings.c

```c
#include <stdio.h>

int main (void) {
  char name[100];
  printf("Enter your name: ");

  if (scanf("%s", name) != 1) {
      printf("Error or EOF \n");
      return 1;
  } else {
      printf("Hi %s!\n", name);
      return 0;
  }
}
```

Terminal

```
osuser@debian:~$ gcc -g greetings.c -o greetings
osuser@debian:~$ ldd greetings
    linux-vdso.so.1 =>  (0x00007fff39a35000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f80e3b95000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f80e3f3d000)
osuser@debian:~$
```

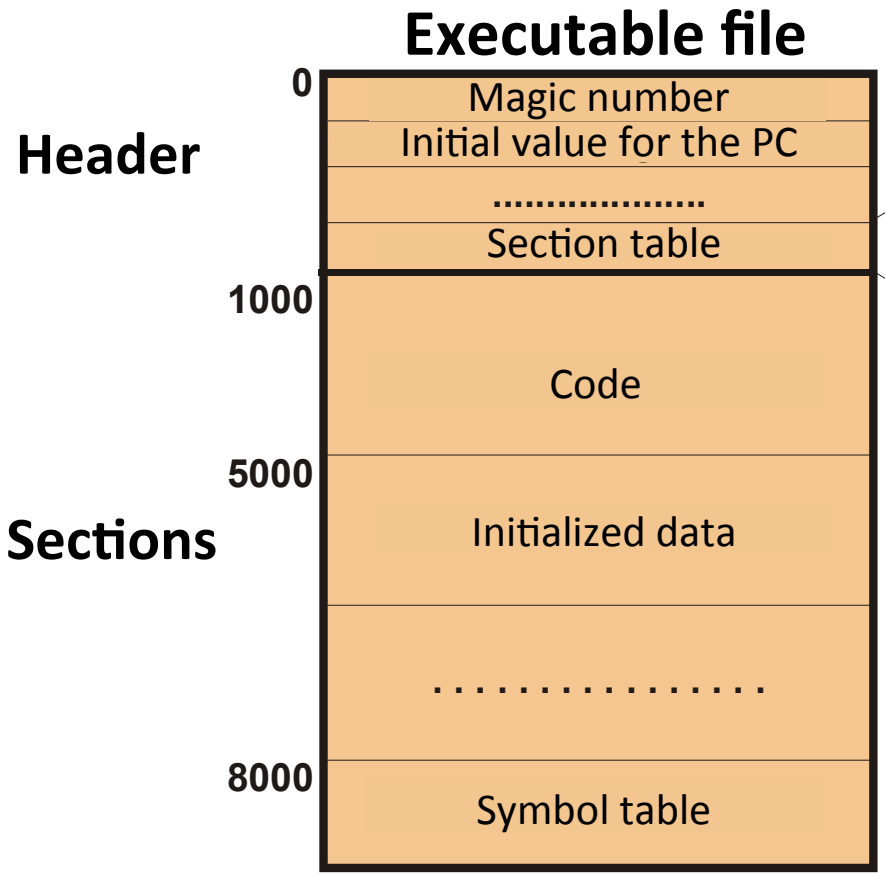
Executable file format

- In UNIX *Executable and Linkable Format* (ELF)

Structure (summary)

1. Header
   - Magic number that identifies the file as an executable (0x7f+'ELF')
   - Program's entry point (initial value for the PC)
   - Section table
2. Sections
   - Code (`.text`)
   - Initialized data (`.data`)
   - Uninitialized data (`.bss`)
     - Just the size is specified
   - Symbol table for debugging (`.debug`)
   - Information of dynamic libraries (`.dynamic`, `.dynstr`, `.dynsym`)

Executable file format



	Offset	Size
Code	1000	4000
Init. data	5000	1000
Uninit. data	------	500
.....	.....	.....
Symbol table	8000	1000

Global variables vs. local variables

Global variables (with section)

- Static allocation
  - Created upon process creation
  - Exist throughout the program's execution
- Fixed address in memory and in the executable

Local variables and parameters (without a section)

- Dynamic allocation
  - Created when the associated function is invoked
  - Destroyed when returning from the function
- The address is calculated at runtime
- Recursion: several instances of the same variable

Global variables vs. local variables

Example

```
int x=8;      /* Global variable with an initial value */
int y;        /* Global variable without initial value */
int j=5;      /* Initializaton and memory allocation for global var. */

int f(int t) {   /* Parameter */
 int z;          /* Local variable */
  ...
}

int main() {
  ...
  f(4);
  ...
}
```

Reminder: useful commands

Commands

- `nm` / `objdump`
  - list symbols/display information from object files and executable files
  - Typical options in `objdump`: -S, -t, -f, -h

nm example

```
osuser@debian:~$ nm -Dsv greetings
                 w __gmon_start__
                 U __isoc99_scanf
                 U __libc_start_main
                 U printf
                 U puts
osuser@debian:~$
```

Reminder: objdump

objdump example

```
osuser@debian:~$ objdump -x greetings
greetings:     file format elf64-x86-64
greetings
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004004c0
...
Sections:
Idx Name Size VMA LMA
File off Algn
..
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
...
 13 .text         0000020c  00000000004004c0  00000000004004c0  000004c0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 15 .rodata       0000002f  00000000004006d8  00000000004006d8  000006d8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
...
 21 .dynamic      000001e0  00000000006007f8  00000000006007f8  000007f8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 24 .data         00000010  0000000000600a18  0000000000600a18  00000a18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 25 .bss          00000008  0000000000600a28  0000000000600a28  00000a28  2**2
                  ALLOC
```

Process memory image

- The Process Memory Image (PMI) is a **set of memory regions** that store all what is necessary for a program's execution
  - Also known as *process address space*
- Each region is an independent contiguous zone
- Each region has data and (OS-managed) metadata associated with it:
  - Start address and initial size
  - Backing: where its initial content is stored if any (e.g., executable file)
  - Protection: RWX
  - Shared/private usage
  - Fixed/Dynamic size (growing direction $\uparrow$ / $\downarrow$)

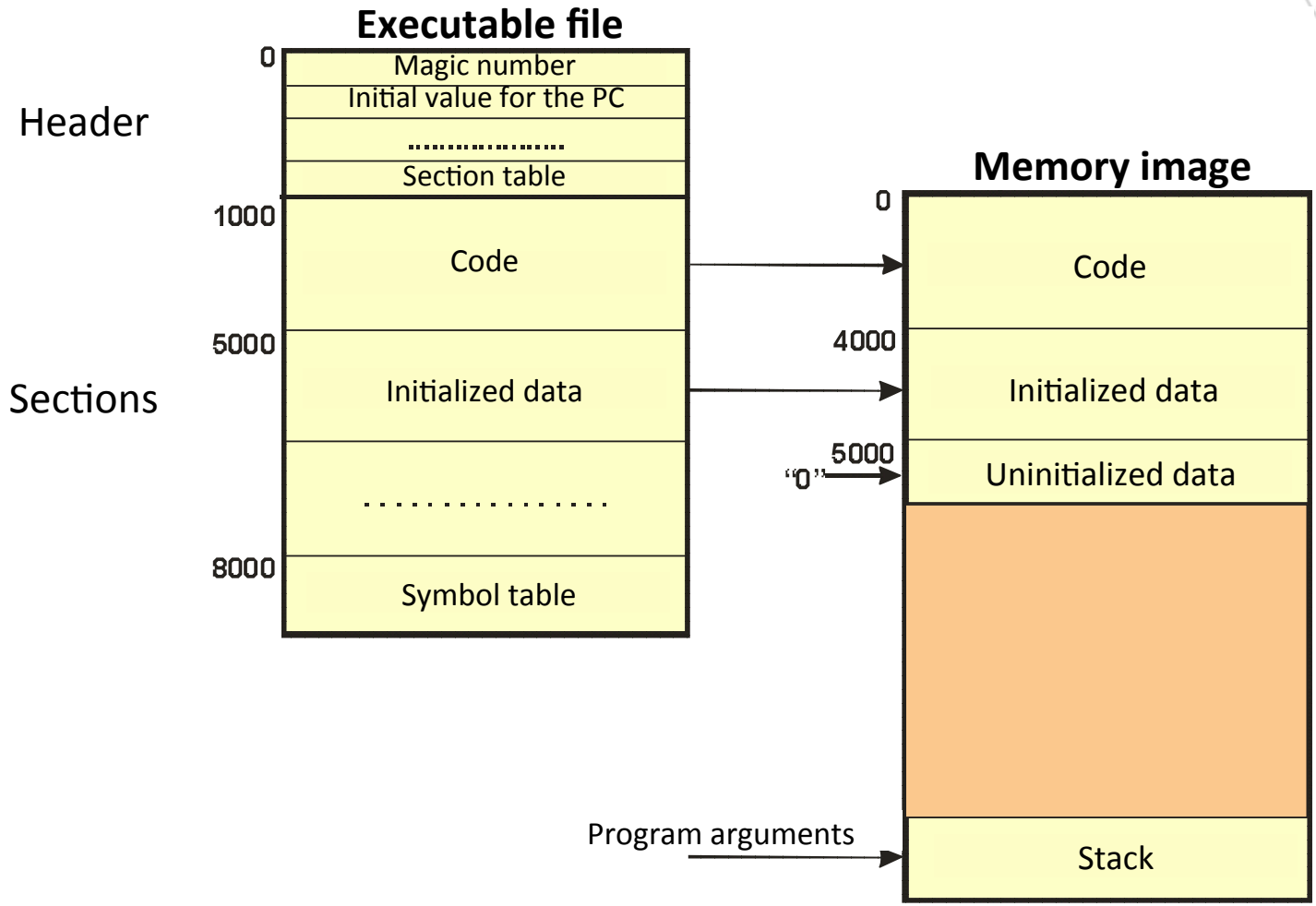
Creating memory image from executable file

- Program execution (`exec()`): Create memory image from executable
  - Some sections in the exec. file → regions in the initial memory image

Basic regions

1. **Code**
   - Shared, RX, Fixed size, Backed by executable file
2. **Initialized data**
   - Private, RW, Fixed size, Backed by executable file
3. **Uninitialized data**
   - Private, RW, Fixed size, No backing (zero fill)
4. **Stack**
   - Private, RW, variable size, No backing
   - Grows towards lower memory addresses
   - Initial stack content: program arguments (`execvp()`)

Creating memory image from executable file



Additional regions in memory image

- While +a process runs, new regions may be created
  - The process memory image has a dynamic nature

Additional regions

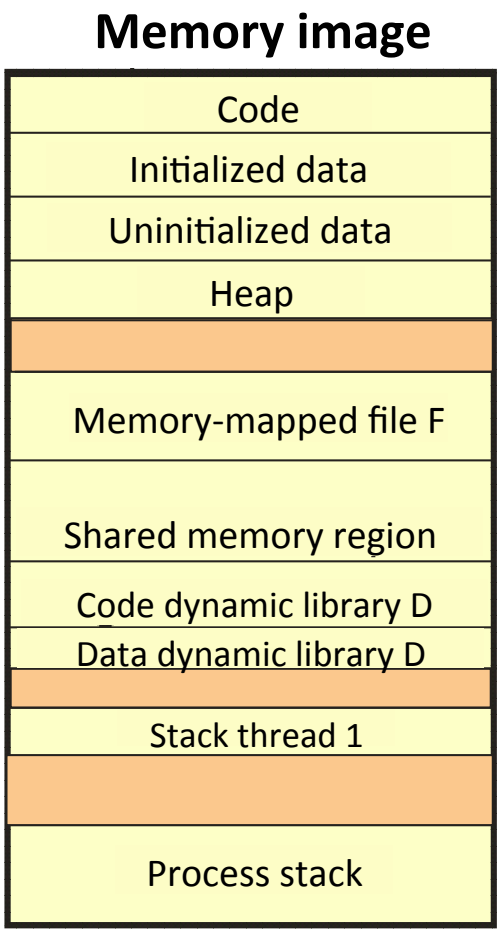
- **Heap region**
  - To support dynamic memory allocation (`malloc()` in C)
  - Private, RW, Variable size, No backing (zero fill)
  - Grows towards higher memory addresses
- **Regions for dynamic libraries**
  - Regions created to hold code and data for each dynamic library
- **Thread stack**
  - For each thread created in the process, a separate stack region exists
  - Same features as process stack region

Additional regions in memory image

Additional regions

- **Shared memory**
  - Region associated with shared memory between processes
    - Must be allocated explicitly by the programmer
  - Private, Variable size, No backing (zero fill)
  - Protection information established upon creation
- **Memory-mapped file** (`mmap()`)
  - Region associated with a file mapped onto memory
  - Variable size, Backed by file
  - Protection/Sharing information established upon mapping

Region features



Region	Backing	Protection	Shared/Priv	Size
Code	File	RX	Shared	Fixed
Init. data	File	RW	Private	Fixed
Uninit. data	None	RW	Private	Fixed
Stacks	None	RW	Private	Variable
Heap	None	RW	Private	Variable
Mapped file	File	User specified	Shared/Priv	Variable
Shared mem.	None	User specified	Shared	Variable

Proc file system

Terminal 1

```
osuser@debian:~$ ./greetings
Enter your name:
```

Terminal 2

```
osuser@debian:~$ ps -a | grep greetings
 7644 pts/5    00:00:00 greetings
osuser@debian:~$ cat /proc/7644/maps
00400000-00401000 r-xp 00000000 00:11 324                /home/osuser/greetings
00600000-00601000 rw-p 00000000 00:11 324                /home/osuser/greetings
7f34830e2000-7f3483263000 r-xp 00000000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483263000-7f3483463000 ---p 00181000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483463000-7f3483467000 r--p 00181000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483467000-7f3483468000 rw-p 00185000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483468000-7f348346d000 rw-p 00000000 00:00 0
7f348346d000-7f348348d000 r-xp 00000000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348366d000-7f3483670000 rw-p 00000000 00:00 0
7f3483688000-7f348368c000 rw-p 00000000 00:00 0
7f348368c000-7f348368d000 r--p 0001f000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348368d000-7f348368e000 rw-p 00020000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348368e000-7f348368f000 rw-p 00000000 00:00 0
7fff4de1b000-7fff4de3c000 rw-p 00000000 00:00 0          [stack]
7fff4de8a000-7fff4de8b000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Operations on regions

- The OS may alter a process memory image during the process's life-cycle

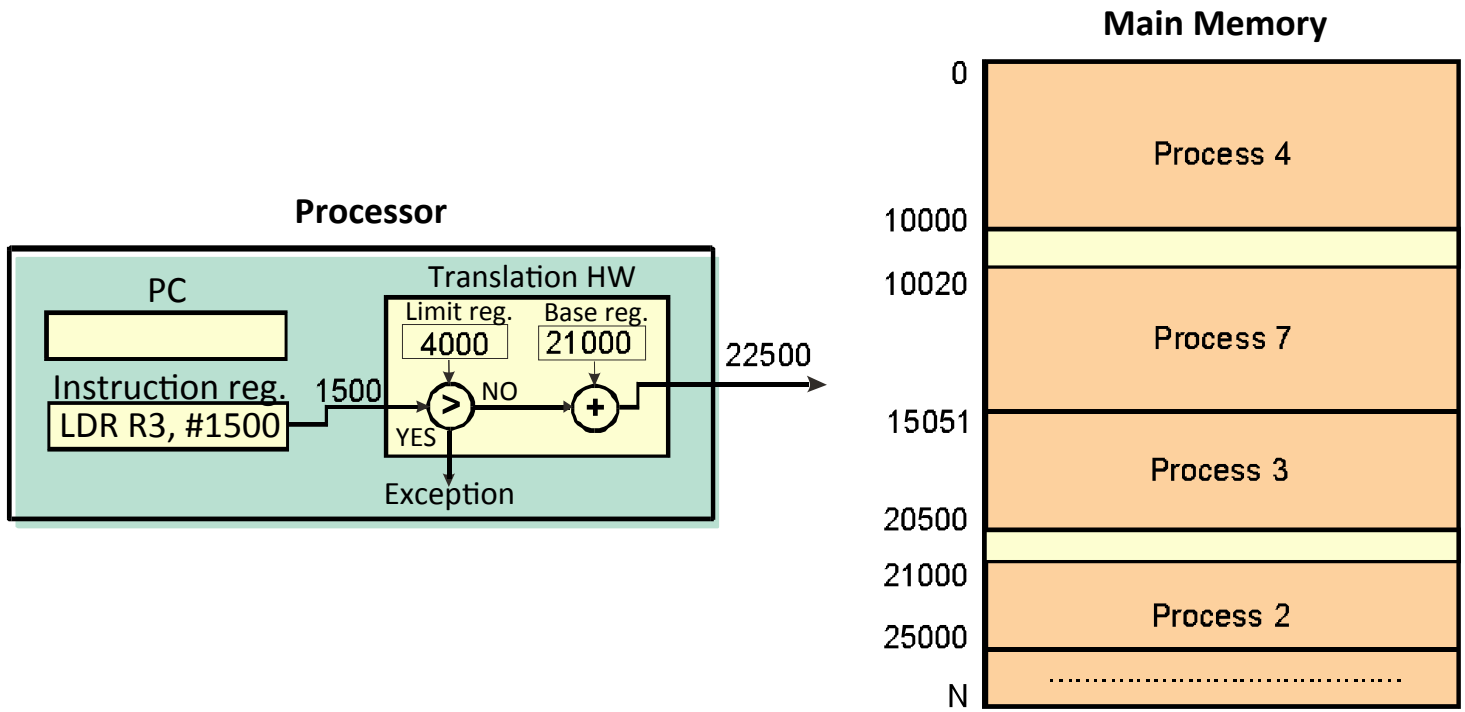
Typical operations

- **Create region**
  - Implicitly when creating the initial image or by a explicit request of the program at run time (e.g., memory-map a file with `mmap()`)
- **Destroy region**
  - Implicitly when the program terminates or upon explicit request of the program (e.g., `munmap()`)
- **Resize region**
  - Implicitly for the stack or upon explicit request of the program (*heap* with `malloc()`)
- **Duplicate region**
  - When creating a new process with `fork()`

Contents

Contiguous memory allocation (I)

- Memory image of each process stored entirely (as is) in a contiguous region of main memory
- Hardware support required: Base register and Limit register
  - Registers accesible only from the OS privilege level (kernel mode)



Contiguous memory allocation (II)

OS maintains information on:

1. Value for the base/limit registers for each process in the PCB
   - The appropiate value for the physical registers is updated on every context switch
2. Memory occupancy status
   - Data structures to keep track of allocated/unallocated regions
   - Also per-process table to keep track of regions inside the process memory image

This scheme is subject to external fragmentation

- Small unallocated "gaps" may exist between allocated regions
- Potential solution: compaction $\rightarrow$ costly operation

Memory allocation policy

- Which free gap must be used to store the memory image of a new process?

Available policies

- **First fit:** Allocate the first gap that is big enough to satisfy the request
- **Best fit:** Allocate the smallest gap that is big enough to satisfy the request
  - List of gaps sorted in ascending order by size
- **Worst fit:** Allocate the biggest gap found
  - List of gaps sorted in descending order by size

- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization

Contiguous allocation: Operations on regions

- When process is created, a fixed-size memory chunk is assigned to it
  - Big enough to hold initial regions (code, data)
  - Gaps must be left inside enabling to resize/create new regions
  - Selecting an appropriate size is challenging
    - If too big, space would be wasted
    - If too small, the process may run out of memory
- Operations:
  - Create/destroy/resize use the per-process region table to manage the process memory image
  - Duplicating a region entails creating a new region and copying the contents
    - Memory cannot be shared among processes due to hardware limitations

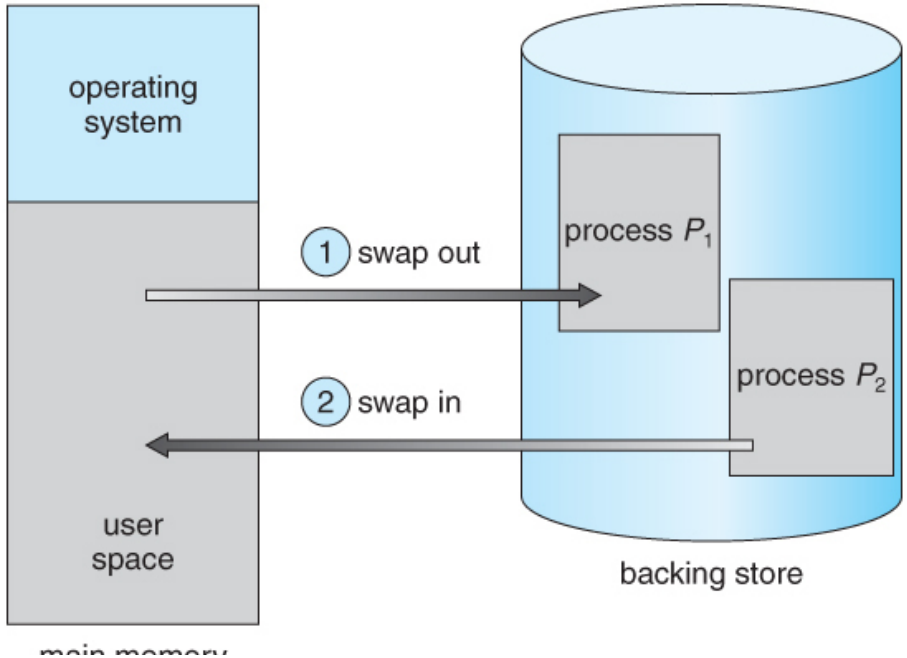
Contiguous memory allocation: summary

Does it meet the typical memory-management requirements?

- Private address space for each process:
  - via limit/base registers
- Protection:
  - via limit/base registers
  - Registers accesible only from the OS privilege level
- Shared memory:
  - not possible
- Support for regions in process memory image:
  - Each region cannot be assigned a different access permission mask (RWX)
  - Wrong accesses to gaps inside the memory image or stack overflow situations cannot be detected
- Maximize memory utilization and support for huge memory images
  - Poor memory utilization due to external fragmentation
  - Huge memory images are not supported

Swapping (I)

- *What to do if we cannot store all processes simultaneously in main memory?* $\rightarrow$ swapping
- Used in early versions of the UNIX OS
- **Swap area**: disk partition or file that stores process memory images
  - Process memory image in swap area $\rightarrow$ suspended process
  - When pure swapping is used, a process cannot run if its memory image is not loaded entirely in memory



Swapping (II)

Swap out

- When all active processes do not fit in memory, at least one process must be evicted from memory by copying its memory image to the *swap* area
- Diferent criteria to select the process to be swapped out:
  - Factor in process priorities
  - Favor processes in the "ready" state over those in the "waiting" state
  - Do not swap out a process if a DMA operation is being performed on the process memory image
- There is no need to copy the entire memory image when process is swapped out:
  - The code is in the executable file
  - Gaps in the memory image do not contain useful information

Swapping (III)

Swap in

- When there is enough room in main memory again, a process can be *swapped in* by loading the memory image from the swap area
- Swap in performed as well when a process has spent a certain amount of time in the swap area
  - In this scenario, another process must be *swapped out* first

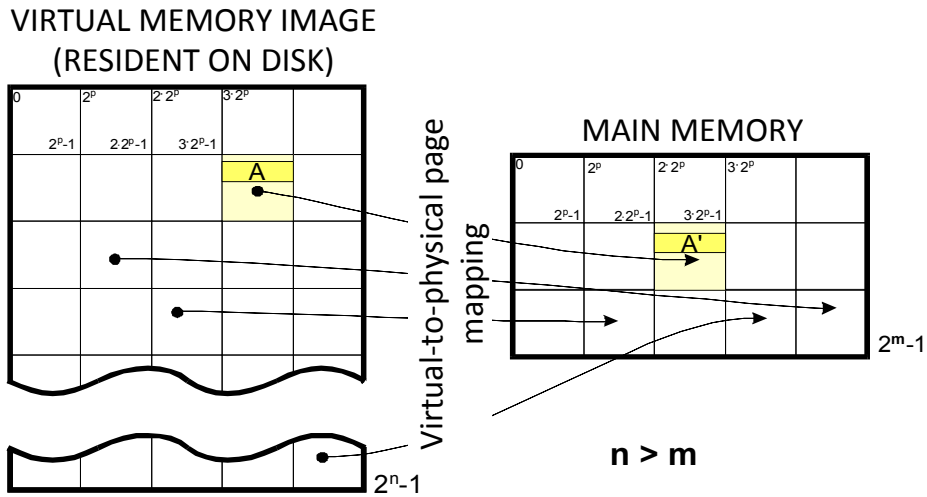
Swapping (IV)

- Policy to allocate space on *swap area*
  - *With preallocation*: space is assigned when process is created
  - *Without preallocation*: space is assigned when process is swapped out
- In order for a process to run, its memory image must be stored completely in main memory (MM)
  - Degree of multiprogramming depends on the size of MM and the size of the memory image of active processes
- More general solution $\rightarrow$ Virtual memory

Contents

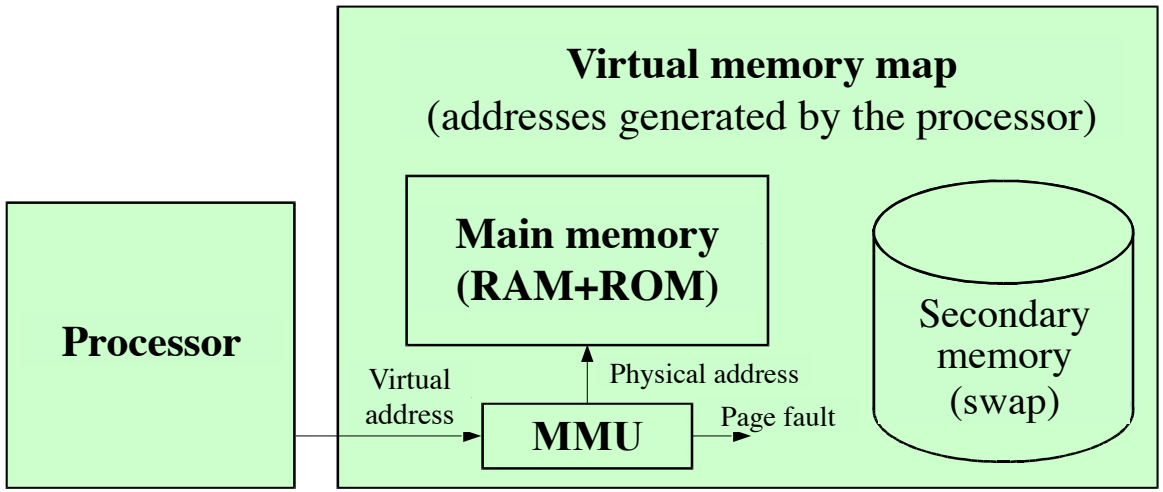
Virtual memory basics (I)

- User processes and the OS generate virtual addresses as they run
  - Virtual memory addresses of a process belong to its **virtual address space** (memory image)
- The virtual address space of a process is divided into *pages*
  - Page: minimal allocation unit in VM (e.g. 4KB)
- Part of the memory image of a process is stored in MM (main memory) and the rest is stored on disk (*swap area* or secondary memory)
- The OS is responsible for loading the "necessary" pages from the process's memory image as the process runs



Virtual memory basics (II)

- Each process uses and generates virtual addresses
  - Virtual address → page number + offset (within the page)
- The MMU (*Memory Management Unit*) translates virtual addresses into physical addresses
  - If the referenced page is in MM → Access the corresponding physical address
  - If the referenced page is not in MM → The MMU generates a *page fault* exception
    - The OS handles the page fault by transferring the page from disk (*swap*) into MM

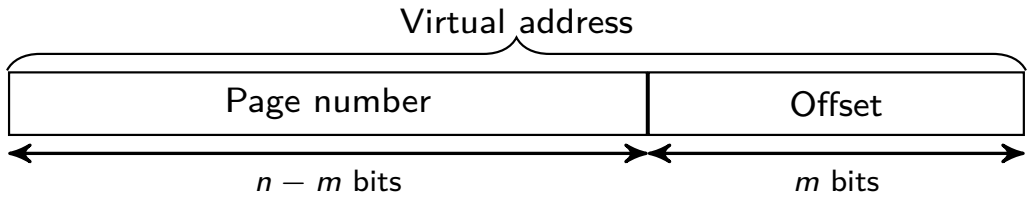


Virtual memory basics (III)

- Page are transferred between both levels (*swap* and MM)
  - From secondary mem. into MM: on demand
  - From MM onto secondary memory: by eviction
- Virtual memory is effective because processes exhibit *locality of reference*
  - Spatial locality (e.g., page belonging to the code region)
  - Temporal locality (e.g., multiple references to the same variable)
  - Most processes use a reduced part of its memory image during a given period of time
    - To maximize performance, the OS strives to ensure that the used part of the memory image (*working set*) is stored completely in MM (*resident set*)
- Benefits:
  - Increases the degree of multiprogramming
  - Enables the execution of programs whose memory image does not fit in MM

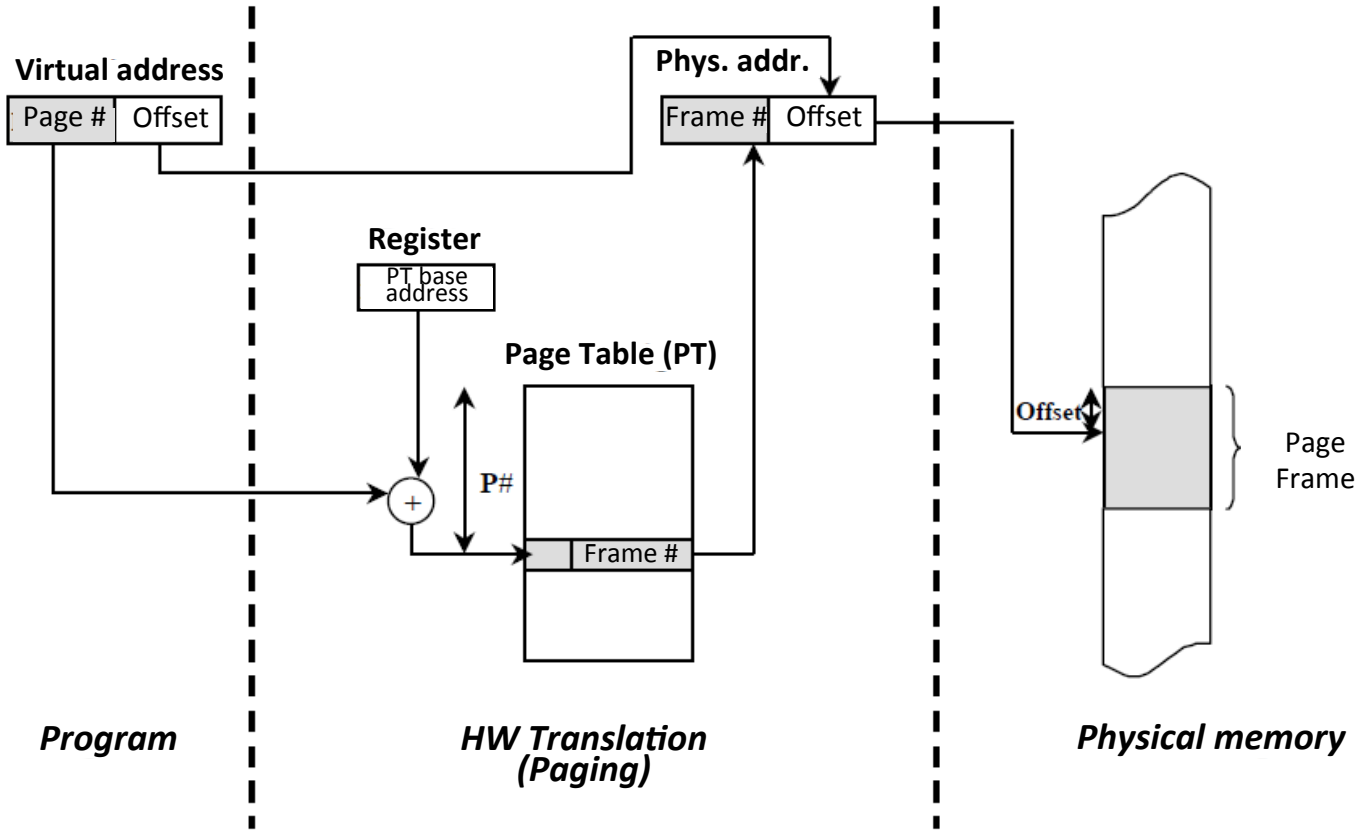
Virtual memory: hardware aspects

- Non-contiguous allocation scheme
- Minimal allocation unit: page
  - Page size (in bytes) is a power of two
  - Process memory image divided into pages
  - Main memory divided into *page frames* (Size of page frame=page size)
- Logical address (or Virtual address): page number and offset
  - Virtual addresses of n bits, $Page\_Size = 2^m$ bytes



- Page table (PT), one for each process
  - Stores the physical location (frame) of each page in the process
  - MMU uses the PT to translate virtual addresses into physical addresses
  - PT stored in main memory

Address translation with page tables

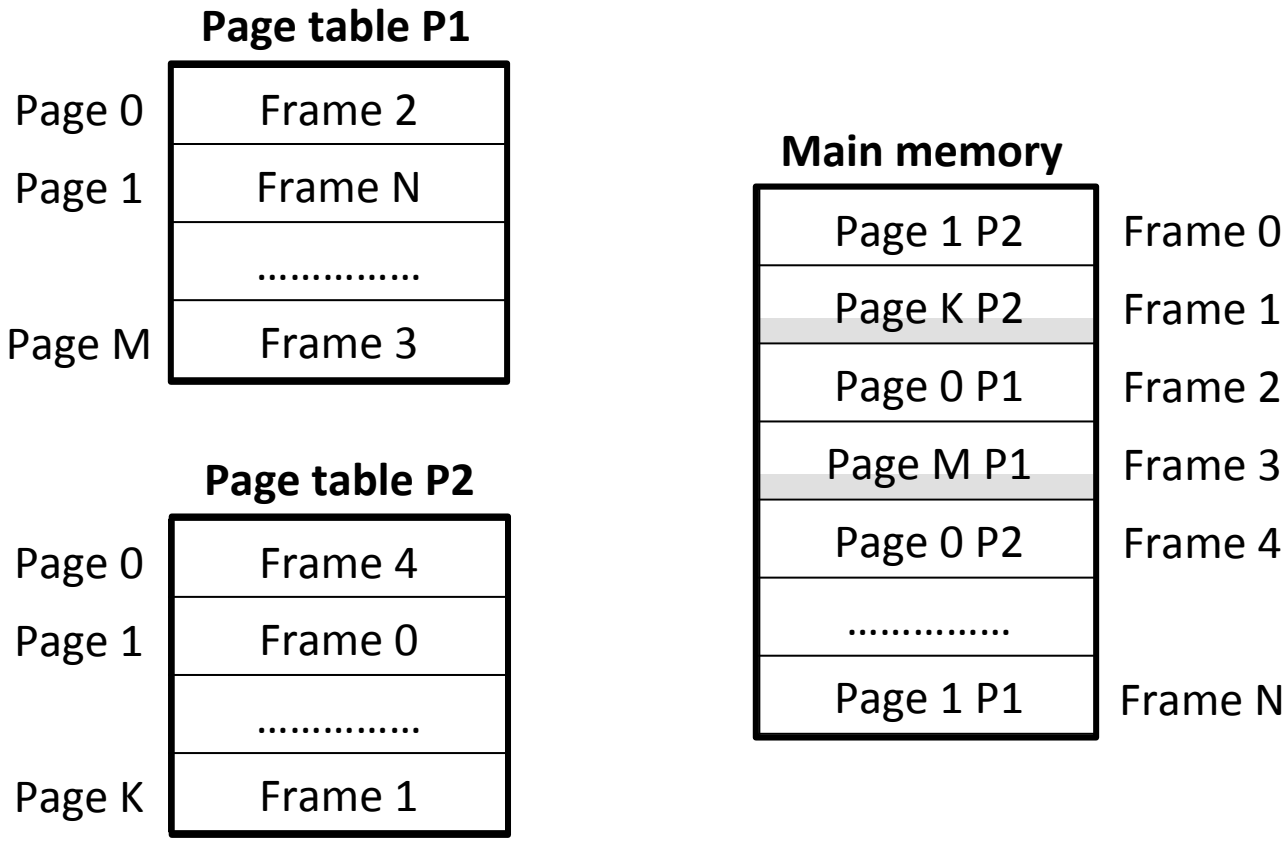


Information in a page table entry

- Page frame number
- Present bit (P)
  - Indicates whether the page is loaded in MM or not
  - Accessing a non-present page (P=0) triggers a page fault
- Protection information: WX (*Write/eXecute*)
  - If operation not allowed → Exception
- Privilege level: User/Kernel
- Reference (or access) bit (*Ref*)
  - MMU activates this bit when the page is accessed for the first time since it was brought to MM
- Modified (or dirty) bit (*Mod*)
  - MMU activates this bit when a process writes on the page
- Cache disable bit

Internal fragmentation with paging

- *Internal fragmentation* occurs when the amount of memory assigned to a process exceeds the amount of memory required
  - Potential space wasted inside allocated pages



Page size

Affects several conflicting factors:

- Power of 2 and multiple of the I/O block size
- Benefits from small pages:
  - Low internal fragmentation
  - It is easier to adjust allocated memory to the process working set
- Benefits from big pages:
  - Smaller page tables
  - Better performance for page transfers between MM and disk (*swap*)
- Good choice for the page size: between 2KB and 16KB
  - In current UNIX-like OSes, check page size with: `getconf PAGESIZE`

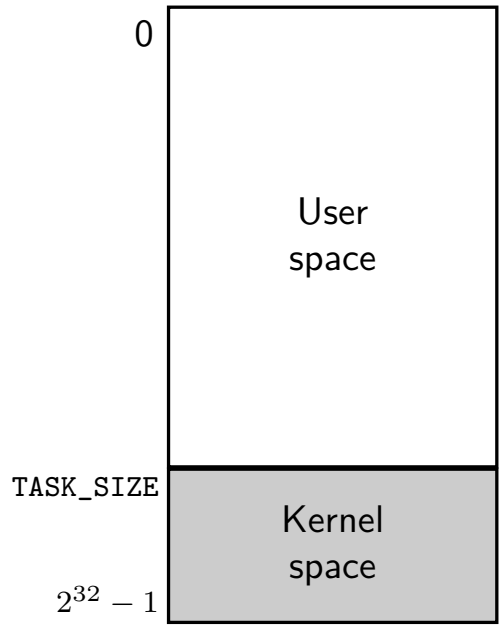
OS management

The OS maintains:

1. Page table for each process
   - When context switch takes place, the OS "tells" the MMU which page table to use
   - HW maintains a per-CPU register that stores the physical base address of the page table associated with the currently running process
     - Register can be accessed at the OS privilege level only
     - Example: In x86, the CR3 register is used (Control register #3)
2. Page table for the OS itself
   - In general, these pages are accessible only when running in kernel mode
3. Page frame table
   - State of each page frame (free/used)
4. Per-process region table

Pages and virtual address space

- 2 types of pages exist in a process address space:
  - User pages: store code/data of the process
  - System pages: store code/data of the OS
- For simplicity, the OS divides the process's virtual address space into *kernel space* (for system pages) and *user space* (for user pages)
- When the process runs instructions in user mode it can only access virtual addresses that fall in the user space range (reference user pages)
  - The MMU prevents the process from accessing system pages
- When the process invokes a system call (and enters kernel mode) the OS may access both spaces



Pages and virtual address space

Features of this organization

- Each process has a private page table (PT) associated with it but ...
  - Kernel-space region of the PT is the same across processes
  - Threads of the same process share the PT
  - The OS and the currently running process share the PT
    - In general, user processes cannot access *system pages*
- Simplifies system call implementation and exception handling
  - The active page table stays the same during changes in the processor mode
  - Makes it possible to translate user space pointers passed as a parameter to *syscalls*

Paging: Summary

Does it meet the typical memory-management requirements?

- Private address space for each process:
  - Via page tables
- Protection:
  - Via page tables
- Shared memory:
  - Entries in PTs of two or more processes refer to the same page frame (several virtual addresses → the same physical address)
- Support for regions in process memory image:
  - Protection bits: WX and User/Kernel
  - *Present* bit: Gaps do not occupy space in MM
- Maximize memory utilization and support for huge memory images
  - Size of process memory image can exceed the amount of physical memory available
- *Issue:* Higher cost in space than contiguous allocation due to larger OS-managed tables
  - The price that comes from much more functionality

Issues associated with page tables

Efficiency:

- For each logical access, two accesses to main memory (one after another) are required:
  - Entry in PT + actual data or instruction from process
- Solution: Address translation cache (TLB)

Storage cost:

- Linear page tables can be huge
  - Example: 4KB pages, 32-bit virtual addresses, 4-byte PT entries
    - PT size: $2^{20} * 4 = 4$MB/process
- Solution (imposed by HW):
  - Multilevel page tables
  - Inverted page tables

Page fault handling

Exception handling

- HW stores the virtual address that caused the page fault in a register
- If invalid address → The process is killed by a signal
- If there are no free page frames, check the frame table:
  - Select victim page: page P, stored in frame F
    - Sets P as invalid (not present) in corresponding PT
  - If P has been modified (Mod bit active)
    - Transfer P to secondary mem. (*swap*)
- If a free frame exists (it was freed up or existed before):
  - Load new page in frame F
  - Update entry in corresponding page table
    - Present bit=1, Frame # = F
  - Set frame F as occupied in frame table

- Handling a page fault require up to 2 disk accesses in the worst case

VM management policies

Page replacement policies:

- What page to evict when page fault occurs and no free frames exist?
- Local replacement scheme
  - A page frame belonging to the process that caused the fault must be allocated
- Global replacement scheme
  - Any occupied frame is eligible for replacement

Page allocation policies:

- How are page frames assigned to the various processes?
  - Static or dynamic allocation

Page replacement policies

- Goal: Minimize page fault rate
- Each replacement policy has a local and a global variant:
  - local scope: criteria applied to resident pages of the process
  - global scope: criteria applied to all resident pages (all page frames)
- Analyzed replacement policies
  - Optimal
  - FIFO
  - Clock (or Second-Chance)
  - LRU
- Any policies can be used along with *page buffering* techniques

Optimal algorithm

- Criteria: the OS evicts the page whose next use will occur farthest in the future $\rightarrow$ oracle required
- Interesting for analytical studies
- Example:
  - Physical memory featuring 4 page frames
  - References: *a b g a d e a b a d e g d e*

Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a	a	a	a	a	a	a	a	a	a	a	g	g	g
1		b	b	b	b	b	b	b	b	b	b	b	b	b
2			g	g	g	e	e	e	e	e	e	e	e	e
3					d	d	d	d	d	d	d	d	d	d

6 faults

FIFO policy

- Criteria: the oldest resident page is evicted
- Simple implementation:
  - Resident pages maintained in FIFO order → the page at the front of the queue is replaced
  - Ref bit is not used by the algorithm
- Poor performance:
  - Old resident pages can be still referenced frequently
  - Replacement criteria does not factor in how often a page was accessed
- Example:
  - Physical memory featuring 4 page frames
  - References: *a b g a d e a b a d e g d e*

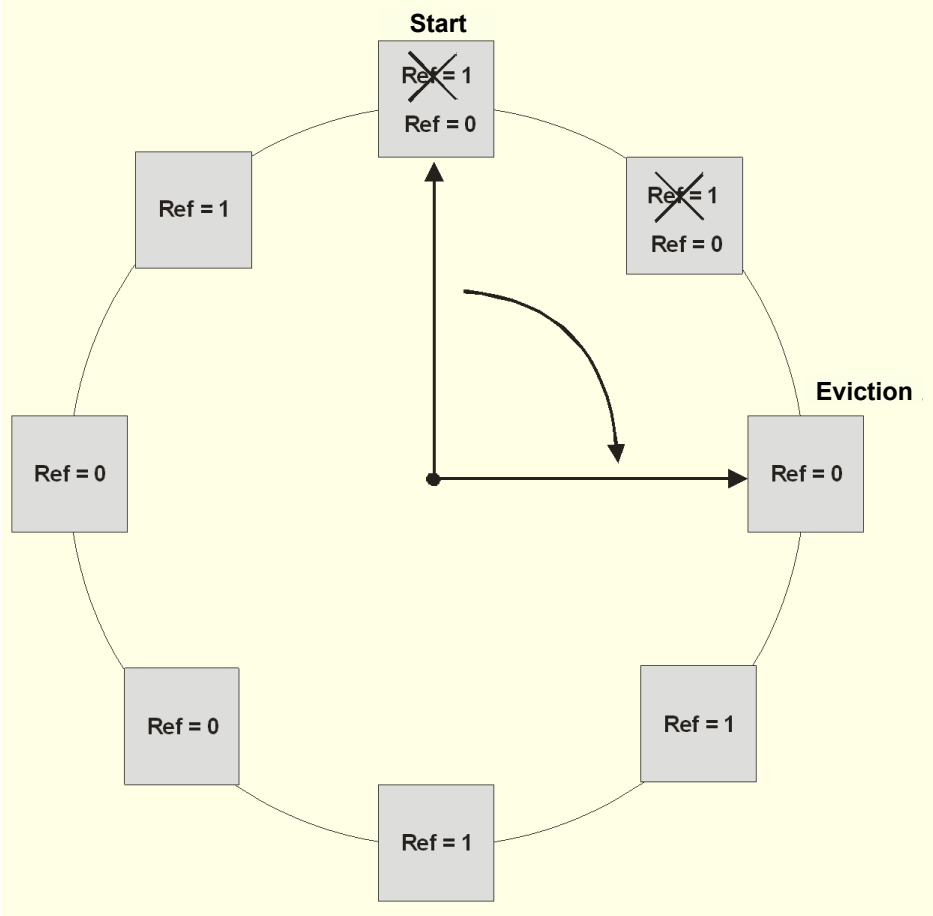
Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a_1	a	a	a	a	e_6	e	e	e	e	e	e	d_{13}	d
1		b_2	b	b	b	b	a_7	a	a	a	a	a	a	e_{14}
2			g_3	g	g	g	g	b_8	b	b	b	b	b	b
3					d_5	d	d	d	d	d	d	g_{12}	g	g

10 faults

Clock policy (aka Second-Chance)

- FIFO + use of reference (Ref) bit
- Criteria:
  - If page selected by FIFO has the Ref bit disabled, the page is replaced
  - If Ref bit enabled $\rightarrow$ give it a second chance
    - Ref bit is cleared
    - Page enqueued at the back of the FIFO queue
    - Apply the selection criteria to the next page (front of the queue)
- It can be implemented by using a circular list with a reference to the first page on the list:
  - Represented as a clock where the "clock's hand" points to the first page on the list

Clock policy (aka Second-Chance)



Clock policy (aka Second-Chance)

- Example:
  - Physical memory featuring 4 page frames
  - References: *a b g a d e a b a d e g d e*

Ref	**a**	**b**	**g**	**a**	**d**	**e**		**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	↓a_0	↓a_0	↓a_0	↓a_1	↓a_1	a_0	a_0	a_1	a_1	a_1	a_1	a_1	a_0	a_0	a_0
1		b_0	b_0	b_0	b_0	↓b_0	e_0	e_0	e_0	e_0	e_0	e_1	e_0	e_0	e_1
2			g_0	g_0	g_0	g_0	↓g_0	↓g_0	b_0	b_0	b_0	b_0	g_0	g_0	g_0
3					d_0	d_0	d_0	d_0	↓d_0	↓d_0	↓d_1	↓d_1	↓d_0	↓d_1	↓d_1

7 faults

LRU policy

- Criteria: replace least recently used page
- Takes temporal locality into consideration
- Strict implementation is difficult (approximate implementations exist):
  - It would require a specific MMU
- Possible implementation with HW support:
  - The MMU maintains a global counter (incremented every so often)
  - Each PT entry features a *timestamp*
  - When a page is referenced, the MMU copies the counter value into the *timestamp* field in the corresponding PT entry
  - Replacement: Page with smallest *timestamp*

LRU policy

- Example:
  - Physical memory featuring 4 page frames
  - References: *a b g a d e a b a d e g d e*

Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a_1	a	a	a_4	a	a	a_7	a	a_9	a	a	a	a	a
1		b_2	b	b	b	e_6	e	e	e	e	e_{11}	e	e	e_{14}
2			g_3	g	g	g	g	b_8	b	b	b	g_{12}	g	g
3					d_5	d	d	d	d	d_{10}	d	d	d_{13}	d

7 faults

Page buffering

- Worst case scenario when handling a page fault
  - 2 disk accesses
- Alternative: maintain a pool of free page frames always
  - Page fault: always uses a free frame (no replacement)
- If number of free frames $<$ threshold
  - "Paging daemon" applies the replacement algorithm continuously:
    - unmodified pages moved to the list of free frames
    - modified pages moved to the list of modified frames
      - when updated on disk they are moved to the free page list
      - page write back can be performed in batches (better performance)
- If a page is referenced when is in the free/modified list
  - page fault retrieves page directly from the list (no I/O)
  - can improve the behavior of "bad" replacement policies

Locking pages in memory

- Sometimes it is necessary to lock pages in memory so that they are not paged out
- Applied to system pages (OS pages)
  - Memory management is simpler for the OS if OS pages are locked
- Pages are also locked when there is an active DMA operation on a page
- Some systems offer services enabling user processes to lock one or more pages of its memory image
  - Suitable for real time processes
  - Can afffect system performance
  - In POSIX, `mlock()` service

Fixed allocation scheme

- A fixed number of frames (resident set) are assigned to the process upon creation
- May depend on the process's features
  - Size, priority,...
- Scheme does not adapt to different program phases
- Relatively predictable behavior
- The replacement policy must be applied locally
- Underlying architecture imposes a minimal number of assigned frames:
  - For example: instruction `MOVE /DIR1, /DIR2`
    - requires at least 3 page frames
    - instruction and both operands must be resident in memory to execute the instruction successfuly

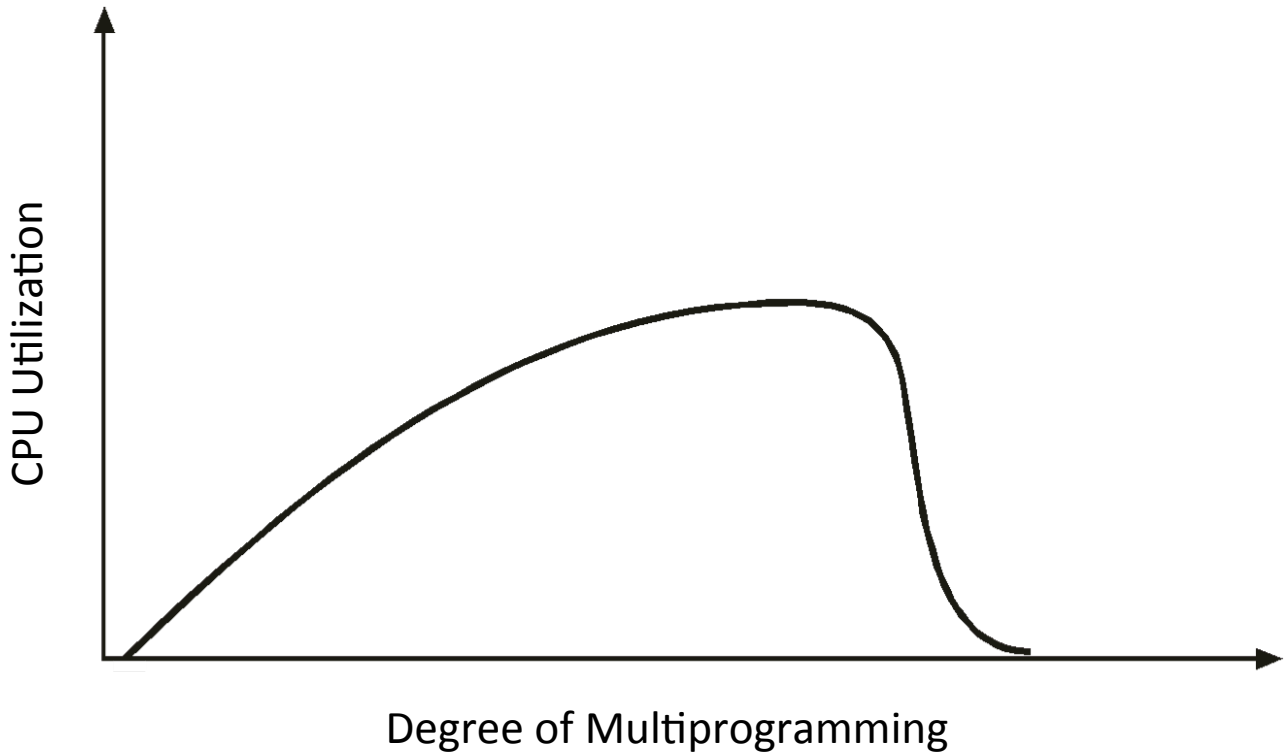
Dynamic allocation scheme

- Number of assigned frames varies with the program behavior (and possibly depends on the behavior of the other processes)
- Dynamic allocation + local replacement scope
  - Process increases/decreases its resident set depending on its behavior
  - Relatively predictable behavior
- Dynamic allocation + global replacement scope
  - Processes compete for page frames
  - Rather unpredictable behavior

Thrashing

- Excessive paging operations are taking place in a process or in the whole system
- With static allocation: *Thrashing* in P_i
  - If resident set of P_i < working set of P_i
- With dynamic allocation: *Thrashing* in the system
  - If number of available frames < $\sum$ resident set size across processes
  - CPU utilization drops rapidly
  - Processes in the "waiting" state most of the time (wait queues of paging device)
  - Solution (similar to that of *swapping*): load control
    1. Reduce the degree of multiprogramming
    2. Suspend one process (or more), thus freeing up their associating page frames
  - Problem: How to detect this situation?

Thrashing in the system

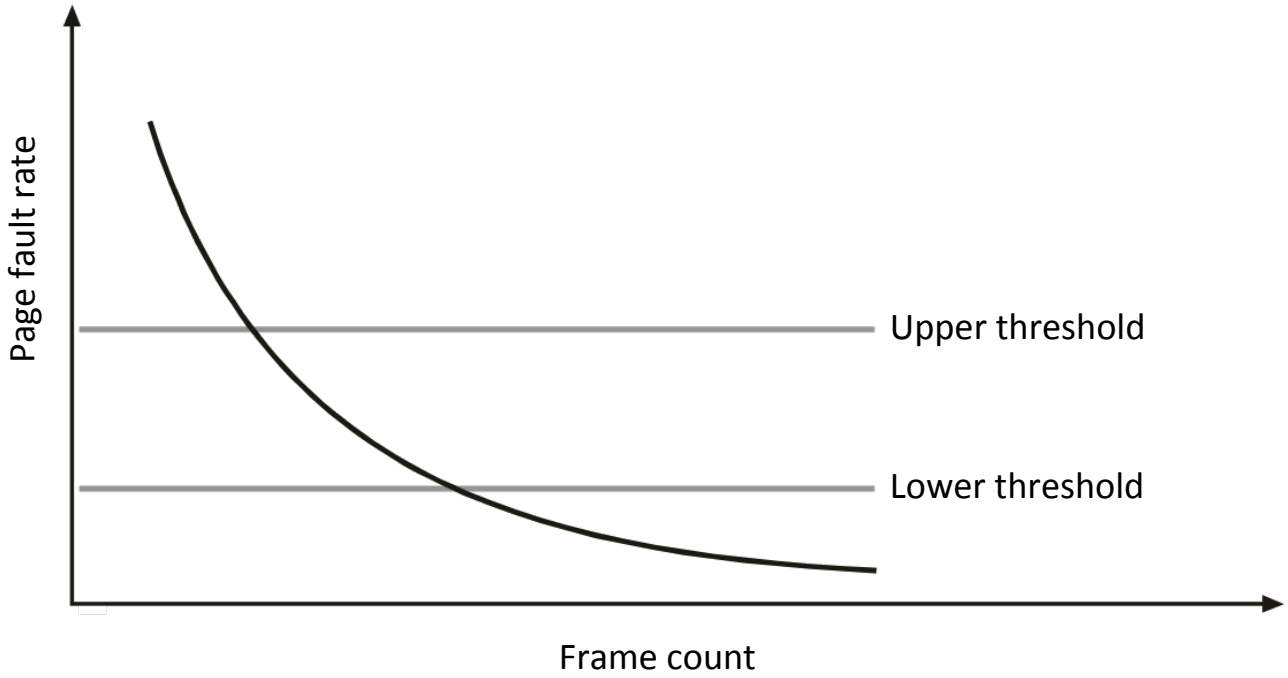


Working set strategy

- Try to adapt memory allocation to the working set (WS) of a process
  - WS approximated with pages referenced by the process in the last N references
- If working set dwindles down, some frames are freed up
- If working set grows, new frames are allocated to the process
  - If not enough frames: swap out process
  - The process is swapped in again when enough free frames exist to accomodate the process working set
- Dynamic allocation scheme with local replacement scope
- Strict implementation is difficult
  - Requires a specific MMU
- Aproximate algorithms exist:
  - Strategy based on the page fault frequency (PFF):
    - Control the page fault rate of a process

Page fault frequency (PFF) algorithm

- If PFF $<$ lower threshold some frames are freed up
- If PFF $>$ upper threshold new frames are allocated
  - If no free frames exist, some processes are swapped out



Load control and global replacement

- Global replacement algorithms do not control thrashing
  - Not even the optimal!
  - Need to operate along with a load control algorithm
- Example: UNIX 4.3 BSD
  - Global replacement with clock algorithm
    - Variant with two clock hands
  - Page buffering used
    - "Paging daemon" controls pool of free frames
  - If number of free frames $<$ threshold
    - "Paging daemon" applies replacement
  - If number of free frames is often below the threshold:
    - "Swapper" process suspends processes

Contents

Memory management on Linux

In Most OSes 2 memory managers coexist

1. Virtual memory manager (for user processes)
2. Memory manager for the kernel (dynamic memory allocation)

User-space management on Linux

Paging system for memory management

- In 32-bit architectures a contiguous 3GB region of the process address space is reserved for user-space addresses and; the remaining 1GB zone is reserved for kernel addresses
- The organization of page tables varies with the processor architecture (imposed by the hardware)
- The kernel allocates a set of contiguous pages using the *buddy allocator*
- Replacement algorithm used: Variant of the clock policy with two hands
- Heap management: the library of the programming language used is responsible for this task
  - The OS assists the library by exposing a reduced set of system calls

Heap management for C programs on Linux

- The heap-management library requests a new page to the OS when more memory is needed by the program
  - `brk()` system call
- Generally, unneeded memory is not returned to the OS
- The assignment must be very efficient to cater to the enormous number of requests:
  - A "lazy buddy" algorithm is used, which relies on multiple lists of variable-sized gaps
    - Contiguous gaps are not merged into a single gap
- For multithreaded programs, versions with multiple heaps exist
  - Enable to reduce bottlenecks in accessing data structures used to maintain the heap state

Kernel-space management on Linux

Dynamic memory allocation in the kernel (Slab Allocator)

- For data structures and buffers created by the kernel
- A set of contiguous page frames is reserved when the OS boots. The allocator takes care of managing that memory chunk at run time
- Object-based allocator: maintains caches of frequently used objects (e.g., process descriptors)
  - The kernel creates and destroys objects of the same type, thus preventing unneeded deallocations and initialization