Memory Management

Alexandru Andrei
alean@ida.liu.se
phone: 013-322828, room: B 3D:439

Last on TTIT61

- Need for processes synchronization
- Race condition, critical section, atomic operations, mutual exclusion
- Critical section access protocols
- Busy waiting
- Hardware support
- Semaphores, monitors, conditions
- Deadlocks

A. Andrei, Process programming and operating systems, Memory management

Lecture Plan

1. What is an operating system? What are its functions?
   Basics of computer architectures. (Part I of the textbook)
2. Processes, threads, schedulers (Part II, chap. III-V)
3. Synchronization & Deadlock (Part II, chap. VI, VII)
4. Primary memory management. (Part III, chap. VIII, IX)
5. File systems and secondary memory management (Part IV, chap. X, XI, XII)

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
  - Page replacement
  - Thrashing

Compiling

```
0x00: pushl b
0x05: pushl val
0x0A: call add
0x0F: movl %eax,a
0x14: addl %esp, $8
.data
0x20: a
0x24: b
```

File1.c
```c
int val;
int add(int a, b) {
    return b = c;
}
```

File2.c
```c
extern int val;
int a, b
int main() {
    a = add(val, b);
}
```

A. Andrei, Process programming and operating systems, Memory management

Introduction

CPU

Fetch

Cache

Registers

Exec. unit

Main Memory

(RAM)

Disk

Speed

Fast

Slow

Compiling

```
0x00: pushl %ebp
0x01: movl %esp, %ebp
0x02: movl 12(%ebp), %eax
0x03: addl 8(%ebp), %eax
0x04: leave
0x08: ret
.data
0x20: val
```
Linking

```
0x00: pushl %ebp
0x05: pushl %ebp
0x0A: pushl add
0x0F: movl %eax.a
0x14: addl %esp, $8
.data
0x20: a
0x24: b
```

Compilation

- The compiler binds variables and code routines to memory addresses
- In programs consisting of different object modules, some variables and/or code routines could be left unbound

Binding

- Compile time binding
  - If the compiler knows where the process will reside in memory, absolute code can be generated, i.e. the addresses in the executable file are the physical addresses
- Load time binding
  - Otherwise, the compiler must generate relocatable code, i.e. code that may reside anywhere in memory
- Execution time binding
  - If the process can be moved during execution from one memory segment to another
  - Special hardware support needed

Relocatable Code

Virtual and Physical Addresses

```
CPU Virtual address MMU Physical address Mem
```

- MMU (memory management unit) – run-time mapping of virtual addresses to physical addresses
- Compile time or load time binding ⇒ virtual = physical

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
  - Page replacement
  - Thrashing
Swapping

- Programs must reside in memory in order to be executed
- But they may temporarily reside on a backing store (hard disk) in order to make room for other programs
- Moving from primary memory to backing store – swap out
- Moving from backing store to primary memory – swap in

- Context switching becomes extremely slow (0.5 sec)

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
- Page replacement
- Thrashing

Contiguous Memory Allocation

Allocation Schemes

- First-fit
  - Allocate the first hole that fits
- Best-fit
  - Allocate the smallest hole that fits (the one that leads to the smallest left-over)
- Worst-fit
  - Allocate the largest hole

Swapping

When we swap-in a process, do we have to put it in the same memory space as before it was swapped-out?

- Depends on the binding method.
  - Compile time and load time: Yes
  - Execution time: Not necessarily
### External Fragmentation

- All schemes in the contiguous allocation method suffer from **external fragmentation**:
  - The total amount of free space is greater than the space required by a new process, but the new process cannot be fitted because no hole is greater than the required space.

- First-fit and best-fit perform better than worst-fit, but even they may lead to 1/3 of the memory space to be unusable.

### Solutions to External Fragmentation

- **Compaction**
  - Move up (or down) such that all (or parts of) free memory is contiguous

- Impossible with compile time and load time binding

### Internal Fragmentation

- Assume that a hole is N bytes and we request N-x bytes, where x is very small.
- The hole that is left after allocation is x bytes.
- Management of a small holes take more memory than the holes themselves.

- Divide the memory space into *blocks* of size BS.
- Allocate the M bytes, where M=BS×⌈R/BS⌉, where R is the requested amount and BS is the block size.

- Not all bytes of the last block may be used ⇒ internal fragmentation.

### Outline

- **Binding**
- **Swapping**
- **Contiguous memory allocation**
- **Paging**
- **Segmentation**
- **Virtual memory**
- **Page replacement**
- **Thrashing**

### Paging

- Memory management scheme that permits non-contiguous allocation.

- Physical memory is divided in fixed-size blocks, called *frames*.
- Virtual address space is divided in blocks of the same size as a frame, called *pages*.
- Backing store is also divided in frames.
- When a process is to be executed, it is loaded into available memory frames of the backing store.
- A page table contains the base address of each page in physical memory.

---

**Paging Model of Logical and Physical Memory**

<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>Frame Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
<td>0</td>
</tr>
<tr>
<td>page 1</td>
<td>1</td>
</tr>
<tr>
<td>page 2</td>
<td>2</td>
</tr>
<tr>
<td>page 3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
<td>frame 0</td>
</tr>
<tr>
<td>page 1</td>
<td>frame 1</td>
</tr>
<tr>
<td>page 2</td>
<td>frame 2</td>
</tr>
<tr>
<td>page 3</td>
<td>frame 3</td>
</tr>
</tbody>
</table>

**Physical Memory**

- page 0
- page 1
- page 2
- page 3
Process Page Tables

Paging

- Memory has $2^N$ bytes $\Rightarrow$ N bits in the address
- A page has $2^n$ bytes $\Rightarrow$ Page offset has n bits
- There are $2^{N-n}$ pages $\Rightarrow$ N-n bits required to select a page
- One entry in the page table contains N-n bits
- The page table is stored using $2^{N-n} \times \left\lceil \frac{(N-n)}{8} \right\rceil$ bytes
- E.g.:
  - Memory is 1GB ($N=30$), page size is 4kB ($n=12$)
  - There are 256k pages
  - Page table occupies 768kB
- If the page size is 8kB ($n=13$), the page table occupies 3072kB

Hardware Support for Paging

- If page tables are very small (up to 256 entries), they could be stored in registers of the CPU (very fast access)
- These registers must be reloaded upon a context switch

- Typically, we have much more than 256-entry page tables $\Rightarrow$ page tables are kept in memory
- There exists a page table base register (PTBR) that contains the base address of the page table
- Only PTBR is reloaded upon a context switch $\Rightarrow$ much faster context switch, but...
- Slower access to the page table

Translation Look-Aside Buffer

E.g.: phys = pageTable[virt / pageSize] * pageSize + virt % pageSize

Paging

- paging eliminates external fragmentation but suffers from internal fragmentation
- Wasted memory = half a page per process $\Rightarrow$ we would like small page sizes
- On the other hand, small page size lead to large page tables and larger management overhead

- Typically, we have page sizes of 4-8kB
TLB

- Is a hardware device
- Is an associative memory, i.e. it is not addressed by address but by data
- Instead of “show me the house at number 10”, you say “show me the house where the Simpson’s live”, i.e. instead of “give the 5th entry in the TLB”, you say “give me the entry in the TLB that corresponds to virtual page X”

- What happens to the context of the TLB upon a context switch?

TLB

- Upon a TLB miss, a new page table entry is copied in the TLB
- If all TLB entries are occupied, one of them must be replaced
- Replacement algorithms: Random, Least Recently Used (LRU), Round-Robin, etc.

Advantages of Paging

- No external fragmentation
- Possibility of page sharing among processes ⇒ higher degree of multiprogramming

Shared Pages

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

Shared Pages Example
Advantages of Paging

- No external fragmentation
- Possibility of page sharing among processes ⇒ higher degree of multiprogramming
- Better memory protection
  - Other information such as read/write/execute permissions may be inserted in the page table
  - Memory addresses that are outside the memory space of the process may be detected (the infamous segmentation faults)

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
  - Page replacement
  - Thrashing

Segmentation

- Programmer views the programs more like a collection of segments
- Segmentation is a memory management scheme that supports this view

Logical View of Segmentation

Segmentation

- Segmentation is a technique that partitions the memory in logically related data units.
  - Code, stack, heap, etc.
- Users view memory as a collection of variable-sized segments, with no necessary ordering among them.
  - Virtual address: <Segment #: Offset>
- Different segments can grow or shrink independently, without affecting each other.
  - Natural extension of variable-sized partitions
Advantages of Segmentation

- Protection
  - Data regarding segment permissions may be added to the segment table
- Possibility of sharing some segments (code from libraries for example) ⇒ higher degree of multiprogramming
- Simplifies the handling: growing or shrinking

Paging vs. Segmentation

<table>
<thead>
<tr>
<th></th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>Fixed (4KB to 64KB)</td>
<td>Variable</td>
</tr>
<tr>
<td>Linear address space</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Memory addressing</td>
<td>One word (page number + offset)</td>
<td>Two words (segment + offset)</td>
</tr>
<tr>
<td>Replacement</td>
<td>Easy (all same size)</td>
<td>Difficult (find where segment fits)</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>Internal</td>
<td>External</td>
</tr>
<tr>
<td>Disk traffic</td>
<td>Efficient (optimized for page size)</td>
<td>Inefficient (may have small or large transfers)</td>
</tr>
<tr>
<td>Transparent to the programmers?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Paging vs. Segmentation

| Can the total address space exceed the size of physical memory? | Yes | Yes |
| Can codes and data be distinguished and separately protected? | No | Yes |
| Can tables whose size fluctuates be accommodated easily? | No | Yes |
| Is sharing of codes between users facilitated? | No | Yes |
| Why was this technique invented? | To get a large linear address space without having to buy more physical memory | To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection |

Segmentation on Top of Paging

Virtual Memory

- A technique that allows the execution of processes that may not be completely in memory
- Thus, programs can be larger than the available physical memory
- Difference between virtual memory and swapping

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
  - Page replacement
  - Thrashing
Why Virtual Memory?

- Often code for handling unusual error conditions is never executed (because the errors do not occur)
- Arrays, lists, and tables are often over-sized (we declare int a[100][100] but we will use only 10x10 elements)
- Some features of the program are used rarely. Even if we used all the code, we would not use all of it at the same time
- No more confined to the size of the physical memory
- Faster loads because the OS does not load the entire program
- Each process would take less physical space ⇒ higher CPU utilization, higher throughput, no response time or turnaround time penalty

Demand Paging

- Similar to a paging system with swapping
- Processes are initially loaded on the backing store
- When the process is dispatched, the dispatcher swaps it in
- But not all of the process, the dispatcher uses a lazy swapper that swaps only the necessary pages
- Correct terminology: not a swapper anymore, but a pager. Swappers swap entire processes

Valid and Invalid Pages

- Let us consider that the program wants to access address X, which belongs to page P.
- How does the OS know if page P is in physical memory or on the backing store?
- Page table contains an “invalid” bit, which is set if
  - The page does not belong to the address space of the process, or
  - The page is not in physical memory
- A page in physical memory is called memory-resident

Page Fault

- Access to an invalid page causes a page fault trap
- It has to be handled by the OS

Page Fault Handling

- The instruction that causes the page fault has not executed at the time of the page fault (because it does not have the operands)
- When a trap occurs, the execution switches to OS code
  - The values of the registers at the time of the page fault are stored in the PCB of the process that caused the page fault
  - The OS finds a free frame in the physical memory
  - It brings the page from the disk to the found frame
  - It resets the invalid bit (now the page is valid)
  - The process resumes in the same state as before the page fault (i.e. with the causing instruction not yet executed)
**Page Fault Handling**

- **load M**: Load M into memory.
- **Reference**: Reference the page.
- **page is on backing store**: The page is on the backing store.
- **Trap**: Trap to OS.
- **Memory**: Memory.
- **Page table**: Page table.
- **Backing store**: Backing store.
- **reset**: Reset the page.
- **being in**: Being in memory.

**Outline**

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
  - Page replacement
  - Thrashing

**Page Replacement**

- **Example:**
  - Consider that the memory has 40 frames
  - A process has 10 pages, but uses on average only 5
  - Without virtual memory, we can fit 4 processes
  - With virtual memory, we can fit 8 processes

- We over-allocate memory
  - Let us fit 6 processes in memory (30 pages on average + 10 pages to spare)
  - Each of the 6 processes may, for some data sets, want to access all of its 10 pages. Then we need 60 pages instead of the 40 available
  - Increasingly likely with growing degree of multiprogramming

- **Two page transfers ⇒ double page fault performance penalty (swap out+swap in)**

- **Use a dirty bit**: a page is dirty if its contents has been modified since it has been swapped in. Thus, the memory-resident page is different from the one on the backing store

- **If the victim is not dirty, it does not need to be swapped out, because the copies in memory and on the backing store are identical ⇒ just one page transfer (copy on write)**

- **Two problems:**
  - Frame allocation algorithm
  - Page replacement algorithm
    - Such that the number of page faults is minimized

- **Page replacement can be**
  - Local: the victim is a frame belonging to the process
  - Global: the victim can be any frame
Frame Replacement

- A certain process uses many pages in time
- Not all of them are needed in the memory simultaneously
- Minimize the number of pages allocated for a process
- Equal allocation
- Proportional allocation

Page Replacement

- Page replacement can be
  - Local: the victim is a frame belonging to the process
  - Global: the victim can be any frame

Page Replacement Algorithms

- FIFO
- Belady's anomaly
- Optimal Page Replacement
- LRU Page Replacement
  - Second Chance
- Counting-Based Page Replacement

Outline

- Binding
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- Virtual memory
- Page replacement
  - Thrashing

Thrashing

- Denotes an intense paging activity
- If a process pages more than executes, then it is thrashing
- It happens when we allocated a small number of frames to a process
- If it page faults, it needs to replace one of its frames, but because there are few frames, most likely the victim will be needed again in short time

Thrashing

- Number of page faults \( \Rightarrow \) CPU utilization
- Degree of multiprogramming \( \Rightarrow \) by the scheduler \( \Rightarrow \) number of page faults

Number of page faults: \( \Rightarrow \) CPU utilization
Degree of multiprogramming: \( \Rightarrow \) by the scheduler: \( \Rightarrow \) number of page faults
Thrashing

- Avoid this domino effect by local replacement algorithms, i.e. the victim is a frame belonging to the process that causes the page fault
- It does not cause other processes to page fault
- However, if the process starts thrashing, paging will be slow for the other processes too \(\Rightarrow\) increased effective access time
- In order to avoid thrashing, processes must have as many frames as they need
- How do we find out how many frames they need?

Working Set Model

- Locality model says that a process execution moves from locality to locality
- A locality is a set of pages that are actively used together
- A working set is composed of the last \(\Delta\) accessed frames, where \(\Delta\) is the working set window

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 5, 6, 7</td>
<td>3, 4</td>
</tr>
</tbody>
</table>

- If the sum of the working set sizes of all processes exceeds the number of frames \(\Rightarrow\) thrashing

Working Set Model

- OS monitors the working set sizes
- If the sum of working set sizes exceeds the number of available frames, the OS selects one process to suspend

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
- If actual rate too low, process loses frame
- If actual rate too high, process gains frame

Summary

- Binding
  - Compile time, load time, execution time
- Swapping
- Contiguous memory allocation
  - External fragmentation
- Paging
  - Internal fragmentation, sharing, protection
- Segmentation
  - External fragmentation, sharing, protection
- Virtual memory
  - Page replacement
  - Thrashing

Reading

- Memory Management: Silberschatz & Galvin & Gagne, Part III
  - Main memory: Chapter 8: 8.1-8.4, 8.6-8.7
  - Virtual memory: Chapter 9: 9.1-9.6