Memory Management

- Logical and Physical Address Spaces
- Contiguous Allocation
- Paging
- Segmentation
- Virtual Memory Techniques (Next Lecture)
Memory Management

- Memory density available for constant dollars tends to double every 18 months.

- Why bother about memory management?

- Parkinson’s Law:
  “Data expands to fill the space available for storage”

- Emerging memory-intensive applications

- Memory usage of evolving systems tends to double every 18 months.
CPU – Memory Performance Gap

“Moore’s Law”
Basic Concepts

- The memory access may take *multiple CPU clock cycles* to complete.

- The system must make sure that each process is confined to its own *address space* in memory.

- One simple implementation: *base register & limit register* pair.
Multi-step Processing of a User Program

- Source program
  - Compiler or assembler
    - Object module
      - Linkage editor
        - Load module
          - Loader
            - In-memory binary memory image
              - Dynamic linking
              - Dynamically loaded system library
                - System library
                  - Other object modules
                    - Compile time
                      - Load time
                        - Execution time (run time)
Virtual vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

- **Virtual (Logical) address** – generated by the CPU
- **Physical address** – address seen by the memory unit.

The user program generates logical addresses; it never sees the real physical addresses.

When are the physical and logical addresses equal? When are they different?
Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** If the memory location is known a priori, absolute code can be generated. Must recompile code if starting location changes.

- **Load time:** Must generate *relocatable* code if memory location is not known at compile time.

- **Execution time:** Binding delayed until run-time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g. *base* and *limit* registers).

- Logical and physical addresses are the same for compile-time and load-time binding.
Address Binding (Cont.)

PROGRAM

JUMP X

LOAD Y

DATA

X

1024

1824

2034

Y

PROGRAM

JUMP 1824

LOAD 2034

DATA

1824

2034

PROGRAM

JUMP 800

LOAD 1010

DATA

800

1010
Suppose we already have the physical address – what is the effective (average) memory access time for a system with a single-level cache memory?

- Cache hit ratio ($h_r$): the fraction of all memory accesses that are found in the cache memory
- Memory access time: $m$
- Cache memory access time: $c$

Effective memory access time:
$$h_r \times c + (1-h_r) \times (c + m)$$

Example: $c = 1$ ns; $m = 100$ ns; $h_r = 0.95$
Effective memory access time: 6 ns
Memory-Management Unit (MMU)

- Hardware device that maps logical address to physical address.

- In a simple MMU scheme, the value in the relocation (or, base) register is added to every address generated by a user process at the time it is sent to memory.
Fixed Partitions

- Divide the memory into fixed-size partitions
- The degree of multiprogramming is bounded by the number of partitions.
- When a partition is free, a process is selected from the input queue and is loaded into memory
Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process
  - Size of each partition is the same and fixed
  - How do we provide protection?

- Advantages
  - Easy to implement, fast context switch

- Issues & Limitations
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)
Fixed Partitions using relocation register

Diagram:
- CPU
- Logical address: 346
- Relocation register: 14000
- MMU
- Physical address: 14346
- Memory
Memory Allocation with Fixed-Size Partitions

Process 1
Process 2
Process 3
Process 4

Partition 1
Partition 2
Partition 3

Process 1
Process 3
Process 2
Variable Partitions

- Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.

- 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.

- Operating system maintains information about:
  a) allocated partitions
  b) free partitions (hole)
Variable Partitioning

- Processes are placed to the main memory in contiguous fashion

- Memory Protection: Relocation register scheme

- Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be smaller than the value in the limit register.
Variable Partitions

- Natural extension -- physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register
  - Why do we need the limit register? Protection
  - If (physical address > base + limit) then exception fault

- Advantages
  - No internal fragmentation: allocate just enough for process

- Issues & Limitations
  - External fragmentation: job loading and unloading produces empty holes scattered throughout memory
Use of Relocation and Limit Registers

Hardware Support for Relocation and Limit Registers

CPU

limit register

relocation register

<

yes

physical address

memory

no

trap: addressing error

GMU CS 571
Contiguous Allocation with Variable-Size Partitions

![Diagram with processes and main memory](image_url)
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 the entire list, unless ordered by size. Produces the smallest leftover hole.

- **Worst-fit**: Allocate the *largest* hole.
  - Produces the largest leftover hole.

- **Next-fit**: Starts to search from the last allocation made, chooses the first hole that is big enough.

- First-fit and best-fit better than worst-fit in terms of storage utilization.
Fragmentation Problem

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; the unused memory is internal to a *partition*.
  - Contiguous allocation with fixed-size partitions has the internal fragmentation problem

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
  - Contiguous allocation with variable-size partitions has the external fragmentation problem
  - **50-percent rule:** Even with an improved version of First Fit, given $N$ allocated blocks, another $N/2$ blocks will be lost to external fragmentation! (on the average)
Memory Defragmentation

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - Must be careful about pending I/O before initiating compaction.
  - Problems?
Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.
Paging (Cont)

- Allows the physical address space of a process to be non-contiguous.
  - Divide physical memory into fixed-sized blocks called frames.
  - Divide logical memory into blocks of same size called pages.
  - Any page can go to any free frame

- A program of size $n$ pages, needs $n$ free frames
  - Set up a page table to translate logical to physical addresses.
  - External Fragmentation is eliminated.
  - Internal fragmentation is a problem.
Paging Example

Logical Address Space of Process A

Main Memory
Address Translation Scheme

- Observe: The simple limit/relocation register pair mechanism is no longer sufficient.
- m-bit logical 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.

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>d</td>
</tr>
<tr>
<td>m – n</td>
<td>n</td>
</tr>
</tbody>
</table>
Address Translation Architecture

![Diagram of address translation architecture]

- **CPU**
- **Logical address**
- **Page table**
- **Physical address**
- **Physical memory**

The diagram illustrates the process of translating a logical address into a physical address using a page table. The page table entries are used to map logical addresses to physical addresses within the memory space.
Paging Example

logical memory

page 0
page 1
page 2
page 3

page table

frame number

0 1 4
1 3
2
3 7

physical memory

0 page 0
1 page 0
2 page 2
3 page 2
4 page 1
5
6
7 page 3
Free Frames

Before allocation

After allocation
Hardware Support for Page Table

- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PTLR), if it exists, indicates the size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called translation look-aside buffer (TLBs).
- TLB is an associative, high speed memory.
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem

- Easy to swap out chunks of a program
  - All chunks are the same size
  - Pages are a convenient multiple of the disk block size
  - How do we know if a page is in memory or not?
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page

- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (next)

- Memory required to hold page table can be significant
  - Need one Page Table Entry (PTE) per page
  - 32 bit address space w/ 4KB pages = 220 PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)
Managing Page Tables

- We computed the size of the page table for a 32-bit address space with 4K pages to be 4MB
  - This is far far too much overhead for each process

- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)

- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if address space is sparse (internal fragmentation)

- Use another level of indirection: two-level page tables
Two-level Page Tables
An Example with 4-Byte PTEs
Two-Level Paging Example II

- A logical address (on 32-bit machine with 4KByte page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- A page table entry is 4 bytes
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- The logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Hashed Page Tables

- For large address spaces, a hashed page table can be used, with the hash value being the virtual page number.
- A page table entry contains a chain of elements hashing to the same location (to handle collisions).
- Each element has three fields:
  - the virtual page number,
  - the value of mapped page frame,
  - a pointer to the next element in the linked list.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table

logical address

p d

hash function

hash table

physical address

r d

physical memory

q s

p r
Another solution to avoid the problem of prohibitively large page tables (64-bit UltraSPARC and PowerPC).

The inverted page table has only one entry for each real page of memory. The system has only one page table.

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs. (Solution?)
Inverted Page Table Architecture
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Particularly important for time-sharing environments.

- **Private code and data**
  - Each process keeps a separate copy of the code and data.
Shared Pages (Example)
Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
  2. When a process accesses the page, the invalid PTE will cause a trap (page fault)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process

- But where does it put it? Has to evict something else
  - OS usually keeps a pool of free pages around so that allocations do not always cause evictions
Efficient Page Translation

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data

- Now two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory

- How can we use paging but also have lookups cost about the same as fetching from memory
  - Cache translations in hardware
  - Translation Look-aside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)
Associative Memory

- Associative memory – parallel search
  - Address translation (A’, A’’)
    - If A’ is in associative register, get frame # out.
    - Otherwise get frame # from page table in memory
    - Typically, TLB contains 64 – 1024 entries.
Translation Look-Aside Buffer

- Translation Lookaside Buffers
  - Translate virtual page #s into PTEs (not physical addresses)
  - Can be done in a single machine cycle

- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address

- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be “mapped”
  - Hit rates are therefore very important
Loading the TLBs

- Most address translations are handled using the TLB
  - >99% of translations, but there are misses (TLB miss)...

- Who places translations into the TLB (loads the TLB)?
  - Software loaded TLB (OS)
    - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)
  - Hardware (Memory Management Unit)
    - Must know where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
Paging Hardware With TLB

CPU

logical address

p d

page number frame number

TLB

TLB hit

f d

physical address

physical memory

page table

TLB miss

p

f
Translation Look-Aside Buffer

- When we want to add a new entry to a full TLB, one entry must be replaced.

- Some entries can be wired down.

- Some TLB’s store address-space identifiers (ASIDs)

- What if TLB does not support separate ASIDs?
Effective Access Time Re-visited

- In the absence of paging mechanism, the effective memory access time is a function of the cache memory / main memory access times and cache hit ratio.

- How does the paging affect the memory access time with or without TLB?
Effective Access Time Re-visited

Example: Assume that in the absence of paging, effective memory access time is 100 nanoseconds (computed through the cache memory hit ratio and cache memory/main memory access times).

Assume that Associative (TLB) Lookup is 20 nanoseconds TLB Hit ratio – percentage of times that a page number is found in TLB. Assume TLB Hit ratio is 95%

Effective Access Time (EAT):

\[ EAT = 0.95 \times 120 + 0.05 \times 220 = 125 \text{ nanoseconds} \]
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become \(<\text{segment #}, \text{offset}>\)
  - Units of memory from user’s perspective

- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process

- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
    Segments named by #, used to index into table
Segmentation Architecture

Logical address consists of a pair:

\(<\text{segment-number, offset}>,\)

Segment table – maps two-dimensional physical addresses. Each table entry has:

- base – contains the starting physical address where the segments reside in memory.
- limit – specifies the length of the segment.

Segment-table base register (STBR) points to the segment table’s location in memory.

Segment-table length register (STLR) indicates number of segments used by a process

segment number \( s \) is legal if \( s < \text{STLR} \).
Segmentation Hardware

CPU

s

<

no

yes

<

limit
base

segment table

trap: addressing error

physical memory
Example of Segmentation

![Diagram of logical address space with segments and a segment table showing limits and bases.]

- Subroutine
- Stack
- Symbol table
- Segment 0
- Segment 1
- Segment 2
- Segment 3
- Segment 4

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
</tr>
</tbody>
</table>

Physical memory:
- Segment 0: 1400
- Segment 1: 4700
- Segment 2: 4300
- Segment 3: 3200
- Segment 4: 5700
Segmentation

- Each segment represents a semantically defined portion of the program.
  - All entries in a segment are likely to be used in the same way.
  - Segment-table entry will contain the protection info.

- Sharing
  - Segments are shared when entries in the segment tables of two different processes point to the same physical location.

- Memory Allocation → external fragmentation

- Segmentation versus Paging
Segmentation

- Extensions
  - Can have one segment table per process
    - Segment #s are then process-relative (why do this?)
  - Can easily share memory
    - Put same translation into base/limit pair
    - Can share with different protections (same base/limit, different protection)

- Problems
  - Cross-segment addresses
    - Segments need to have same #s for pointers to them to be shared among processes
  - Large segment tables
    - Keep in main memory, use hardware cache for speed
Segmentation with Paging: Intel Pentium

Logical address → Segmentation unit → Linear address → Paging unit → Physical address → Physical memory

Logical address

Selector

Offset

Descriptor table

Segment descriptor

32-bit linear address
The Pentium architecture allows a page size of either 4 KB or 4 MB.

For 4 KB pages, a two-level paging scheme is used.

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

- The ten high-order bits of the linear address reference an entry in the outermost page table (page directory).
- One entry in the page directory is Page Size flag.
  - If set, it points to a 4 MB page: the inner page table is by-passed and the 22 low-order bits in the linear address refer to the offset in the 4-MB frame.
VMemory Summary

- Virtual memory
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses

- Various techniques
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user’s perspective
  - Combine paging and segmentation to get benefits of both