ECE 7650 Scalable and Secure Internet Services and Architecture
---- A Systems Perspective

Part I: Operating system overview:

Memory Management
Hardware background

- The role of primary memory
  - Program must be brought (from disk) into memory and placed within a process for it to be run
  - Main memory and registers are only storage CPU can access directly
- The issue of Cost
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
Base and limit registers

A pair of **base** and **limit** registers define the logical address space.
Logical vs. physical address space

• The separation of logical address space from physical address space is central to proper memory management

  ➢ Logical address – is a memory location seen by an application program and is generated by the CPU; also referred to as virtual address.
    — User program can always start its address space from 0;
    — A compiler generates relocatable addresses for each module;
    — Linker or loader bind the addresses to absolute address from 0

  ➢ Physical address – address seen by the memory unit
    — The final mapping of logical address to physical address is done in execution time and can be changed during the period.
    — The user program deals with logical addresses; it never sees the real physical addresses

• The mapping of logical address to physical is supported by hardware (Memory-Management Unit (MMU))
Dynamic mapping using a register

![Diagram showing CPU, MMU, relocation register, logical address, physical address, and memory.]
Contiguous memory allocation

• Main memory usually into two partitions:
  ➢ Resident operating system, usually held in low memory with interrupt vector
  ➢ User processes then held in high memory

• Relocation registers used to protect user processes from each other, and from operating-system code and data
  ➢ Base register contains value of smallest physical address
  ➢ Limit register contains range of logical addresses – each logical address must be less than the limit register
  ➢ MMU maps logical address dynamically
HW address protection with base and limit registers
Multiple-partition allocation

- 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)
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

- 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 possible, and is done at execution time.
Paging

- The goal of paging
  - Both logical address space and physical address space of a process can be noncontiguous;
  - Process can be allocated physical memory dynamically according to the memory competition;

- The mechanism – dividing both spaces into small units
  - Divide physical memory into fixed-sized blocks called frames (size is power of 2)
    - usually between 512 bytes and 8,192 bytes
  - Divide logical memory into blocks of same size called pages
  - Keep track of all free frames
  - Map a free frame to a page upon program loading or on page fault exception

- Set up a per-process page table to translate logical to physical addresses
- Internal fragmentation
An illustration of paging

Virtual address space

Physical address space

- page belonging to process
- page not belonging to process
Address Translation Scheme

Virtual 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>

for given logical address space \( 2^m \) and page size \( 2^n \)
Paging Hardware

Diagram showing the process of converting logical addresses to physical addresses using a page table. The CPU generates a logical address, which is then mapped to a physical address through the page table. The physical address is then used to access memory.
Free Frames

Before allocation

After allocation
Implementation of Page Table

• Page table is kept in main memory
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PRLR) indicates size of the page table

• Reducing cost in looking up the table
  - 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 translation look-aside buffers (TLBs)
  - Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
Paging Hardware With TLB

- CPU
- Logical address
- Page number
- Frame number
- TLB
- TLB hit
- Physical address
- Page table
- Physical memory
Memory protection

- Memory protection implemented by associating protection bit with each frame

- Valid-invalid bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
Memory protection (cont’d)
Shared pages

- Shared data/code
  - One copy of read-only data shared among processes (i.e., library, shared data).

- Private data/code
  - Each process keeps a separate copy of the code and data
Multi-level page tables

• Reducing the amount of memory required for per-process page table.
  ➢ If a one-level page table was used, it needs $2^{20}$ entries (e.g., at 4 bytes per entry, 4MB of RAM) to represent the page table for each process, assuming a full 4GB linear address space is used.

• Multiple-level page table
  ➢ Break up the logical address space into multiple-level page tables
  ➢ Outer/up level page table must be set up for each process
  ➢ However, lower-level page tables can be allocated on demand
Two-level page-table example
Two-level paging example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 12-bit page offset

Thus, a logical address is as follows:

\[
\begin{array}{ccc}
\text{page number} & \text{page offset} \\
p_i & p_2 & d \\
10 & 10 & 12 \\
\end{array}
\]

where \(p_i\) is an index into the outer page table, and \(p_2\) is the displacement within the page of the outer page table.
Three-level Paging Scheme

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit.
- A C compiler might create these segments:
  - The code, global variables, the heap, the stack, and the C library
Logical View of Segmentation

user space

physical memory space
Segmentation Architecture

• Logical address consists of a two tuple:
  <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 program;
  segment number \( s \) is legal if \( s < \text{STLR} \)
Segmentation Architecture (Cont’d)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

- Protection bits associated with segments; code sharing occurs at segment level

- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
Example of Segmentation

logical address space

subroutine
segment 0

stack
segment 3

symbol table

Sqrt
segment 1

main program
segment 2

\begin{tabular}{|c|c|}
\hline
limit & base \\
\hline
0 & 1000 1400 \\
1 & 400 6300 \\
2 & 400 4300 \\
3 & 1100 3200 \\
4 & 1000 4700 \\
\hline
\end{tabular}

\newpage

physical memory
Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - which produces linear addresses
  - Linear address given to paging unit
    - which generates physical address in main memory
    - Segmentation and paging units form MMU
Logical to physical address translation in Pentium

Diagram:
- CPU
- Logical address
- Segmentation unit
- Linear address
- Paging unit
- Physical address
- Physical memory

Tables:
- Logical address
- Selector
- Offset
- Descriptor table
- Segment descriptor
- 32-bit linear address

Notes:
- local descriptor table (LDT)
- global descriptor table (GDT)
- Page table
Pentium Paging Architecture

Linear/virtual address

<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>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
Three-level Paging in Linux

<table>
<thead>
<tr>
<th>global directory</th>
<th>middle directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
</table>

(linear address)

- global directory
- middle directory
- page table
- offset

CR3 register

page frame
Background of Virtual Memory

• Virtual memory – a technique that allows the execution of processes that are not completely in memory, so that many processes can stay in memory simultaneously to allow multiprogramming.

• Enabling technique: separation of user logical memory from physical memory.
  ➢ Only part of the program needs to be in memory for execution
  ➢ Logical address space can therefore be much larger than physical address space
  ➢ Allows for more efficient process creation

• Virtual memory can be implemented via:
  ➢ Demand paging
  ➢ Demand segmentation
Virtual memory that is larger than physical memory
Shared data/code using virtual memory
Copy-on-write

• Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - Looks like each have their own copy, but postpone actual copying until one is writing the data
  - If either process modifies a shared page, only then is the page copied
  - COW:
    – Don’t copy pages, copy PTEs – now have 2 PTEs pointing to frame
    – Set all PTEs read-only
    – Read accesses succeed
    – On Write access, copy the page into new frame, update PTEs to point to new & old frame

• COW allows more efficient process creation as only modified pages are copied

• Free pages are allocated from a pool of zeroed-out pages
COW example: Before Process 1 Modifies Page C
COW example: After Process 1 Modifies Page C
Demand Paging

• Demand paging: bring a page into memory only when it is needed
  ➢ Commonly used in VM system;
  ➢ Less I/O needed
  ➢ Less memory needed
  ➢ Efficient memory allocation to support more users

• When a logical address is presented to access memory
  ➢ Check if the address is in the process’s address space;
  ➢ If yes, traverse the page table, check corresponding entry at each level of the table;
  ➢ There is a present bit in each entry to indicate if its pointed next-level table is in memory
  ➢ If the present bit is unset, then OS needs to handle page fault
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory (e.g., using mmap() system call)

- A page-sized portion of the file is read from the file system into a physical page and mapped to a process’s address space. Subsequent memory address to this address space is equivalent to reads/writes to/from the corresponding region in the file. Demand paging is still applied.

- Simplifies file access by treating file I/O through memory rather than read(), write() system calls

- Also allows several processes to map the same file allowing the pages in memory to be shared
Steps in Handling a Page Fault

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Physical memory
5. Reset page table
6. Restart instruction

load M

operating system

reference
What happens if there is no free frame?

• Use page replacement – find some page (aka victim page) in memory and evict it out of memory.

• Page replacement supports virtual memory
  ➢ large virtual memory can be provided on a smaller physical memory

• Performance goal – want an algorithm which will result in minimum number of page faults (miss ratio)

• In addition, use modify (dirty) bit to reduce overhead of page transfers
  ➢ only modified pages are written to disk
Basic Page Replacement

1) Find the location of the desired page on disk

2) Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use its page replacement algorithm to select a victim frame (if the frame is dirty, it needs to be written back)

3) Bring the desired page into the (newly) free frame; update the page and frame tables

4) Restart the process
Page Replacement Algorithms

• Objective: lowest page-fault rate

• Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

• The reference string is like:

   1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Graph of Page Faults Versus The Number of Frames
FIFO Page Replacement

- FIFO: Evict oldest page:
  - Problem: completely ignores usage pattern
    - first pages loaded are often frequently accessed

<table>
<thead>
<tr>
<th>reference string</th>
<th>page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 2 2 4 4 4 0 0 0 7 7 7</td>
</tr>
<tr>
<td></td>
<td>0 0 0 3 3 3 2 2 2 1 1 1 1 0 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0 0 0 3 3 3 2 2 2 2 2 1</td>
</tr>
</tbody>
</table>
Belady’s Anomaly

For some page replacement algorithms, the page fault rate may increase as the number of page frames increases.

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(1) Using 3 page frames, there are 9 page faults
(2) Using 4 page frames, there are 10 page faults
Optimal Algorithm

- The MIN algorithm: replace the page that is accessed the farthest in the future, e.g. that won’t be accessed for the longest time

- Problem: don’t know what are the future references
  - Used for measuring how well a practical algorithm performs
Least Recently Used (LRU) Algorithm

- LRU: Evict least-recently-used page
- LRU is designed according to the temporal locality
  - A page recently accessed is likely to be accessed again in the near future
- Good performance if past is predictive of future.
- Major problem: would have to keep track of “recency” on every access, either timestamp, or move to front of a list
  - infeasible
LRU Algorithm Implementation

• Stack implementation – keep a stack of page numbers in a double link form:
  
  ➢ Page referenced:
    — move it to the top
    — requires 6 pointers to be changed
  
  ➢ No search for replacement

• Use of a stack to record the most recent page references

```
reference string
4 7 0 7 1 0 1 2 1 2 7 1 2

2
1
0
7
4

stack before a

2
1
0
7
4

stack after b

7
2
1
0
4

a

b
```
LRU Approximation Algorithms

• Hardware support: reference bit
  ➢ With each page associate a bit, initially = 0
  ➢ When page is referenced bit set to 1
  ➢ Replace the one which is 0 (if one exists)
    – We do not know the order, however

• Second chance algorithm
  ➢ Need reference bit
  ➢ Clock replacement
  ➢ If page to be replaced (in clock order) has reference bit = 1 then:
    – set reference bit 0
    – leave page in memory
    – replace next page (in clock order), subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

(a) circular queue of pages

(b) circular queue of pages
# Replacement Examples

<table>
<thead>
<tr>
<th>Page address stream</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK</td>
<td>2*</td>
<td>3*</td>
<td>3*</td>
<td>3*</td>
<td>5*</td>
<td>4*</td>
<td>3*</td>
<td>2*</td>
<td>2*</td>
<td>2*</td>
<td>3*</td>
<td>3*</td>
</tr>
</tbody>
</table>

F = page fault occurring after the frame allocation is initially filled
Other Replacement Algorithms

- **LFU Algorithm**: Keep a counter of the number of references that have been made to each page and replaces page with smallest count.

- **MRU Algorithm**: replace most-recently-used page.
Global vs. Local Allocation

• Local replacement
  ➢ OS selects a victim page from only the set of frames allocated to the process that has the page fault
  ➢ Pro: One process can hold its allocated frames without worrying about interference from others.
  ➢ Con: the number of allocated memory cannot be dynamically adjusted according to changing memory demands of various processes.

• Global replacement
  ➢ OS selects a victim frame from the set of all frames; one process can take a frame from another
  ➢ Pro: has greater system throughput and is more commonly used.
  ➢ Con: One process can excessively lose its frames, leading to thrashing
Thrashing

- A scenario: access to a page causes a page fault, system evicts another page from its frame, and next access goes to just-evicted page which must be brought in
- Worst case: a phenomenon called thrashing
  - leads to constant swap-out/swap-in
  - 100% disk utilization, but no process makes progress
    - CPU mostly idle
Locality in a Memory-Reference Pattern

• Temporal locality: if a page is accessed, it is likely to be accessed soon.
• Due to existence of locality, in a certain period of time only limited number of pages are touched.
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instruction

- $WSS_i$ (working set of process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies in time)

  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \sum WSS_i \equiv$ total demand frames

- if $D > m \Rightarrow$ Thrashing (m is total physical frames)
  - then swap/suspend/kill one of the processes?
Keeping Track of the Working Set

• Approximate with interval timer + reference bits
  
  ➢ Example: $\Delta = 10,000$
    — Timer interrupts after every 5000 time units
    — Keep in memory 2 bits for each page
    — Whenever a timer interrupts, copy and set the values of all reference bits to 0
      — If one of the bits in memory $= 1 \Rightarrow$ page in working set

• Why is this not completely accurate?
  
  ➢ Working set is a property inherent to a process;
  
  ➢ In a multiprogramming environment, this approach cannot accurately track WS of a particular process.
Can Thrashing be Prevented?

• Why is there thrashing?
  - Process does exhibit locality, but the locality is simply too large
  - Processes individually fit & exhibit locality, but in total they are too large for the system to accommodate all

• Possible solutions:
  - Buy more memory
    — ultimately have to do that
    — The reason why thrashing is nowadays less of a problem than in the past – still OS must have strategy to avoid worst case
  - Let OS decide to kill processes that are thrashing
    — Linux kills a process if many trials of searching of a free page frame for the process fail

• Approach to prevent thrashing in a multiprogramming environment
  – token-ordered LRU
How Thrashing Develops in the kernel?

- In the global replacement, the LRU algorithm selects an LRU page for replacement throughout the entire user memory space of the system.

- When the total memory demands of the concurrently running processes exceed available user memory space to a certain degree, the system starts thrashing --- none of the programs are able to establish their working sets.

- Each process is treated equally by the LRU policy, but none can proceed once thrashing happens.
The performance under thrashing

**Dedicated Executions**

- Memory shortage 42%.
- The time of first spike of gcc is extended by 65 times,
- The time of a stair of vortex is extended by 14 times

**Concurrent Executions**
The False LRU Page

- A page frame of a process becomes a replacement candidate under the LRU algorithm if the page has not been used for a certain period of time.

- There are two conditions under which a page is not accessed by its owner process:
  1) the process does not need to access the page;
  2) the process is conducting page faults (sleeping) so that it is not able to access the page although it might have done so without the page faults.

- We call the LRU pages generated on the first condition true LRU pages, and those on the second condition false LRU pages.

- These false LRU pages are produced by the time delay of page faults, not by the access delay of the program.
  ➢ The LRU principle is not maintained.

- However, LRU page replacement implementations do not discriminate between these two types of LRU pages, and treats them equally!
Thrashing Prevention: Token-order LRU

• The basic idea:

  ➢ Set a token in the system.

  ➢ The token is taken by one of the processes when page faults occur.

  ➢ The system eliminates the false LRU pages from the process holding the token to allow it to quickly establish its working set.

  ➢ The token process is expected to complete its execution and release its allocated memory as well as its token.

  ➢ Other processes then compete for the token and complete their runs in turn.

• By transferring privilege among processes in thrashing from one to another, the system can reduce the total number of false LRU pages and to transform the chaotic order of page usages to an arranged order.

• The policy can be designed to allow token transferred more intelligently among processes to address issues such as fairness and starvation.
Comparison for gcc+vortex (42% memory shortage)

Without Token

Token-ordered LRU