Memory Management

“Multitasking without memory management is like having a party in a closet.”

– Charles Petzold. *Programming Windows 3.1*

“Programs expand to fill the memory that holds them.”

**Why memory management?**

- Process isolation
- Automatic allocation and management
- Support for modular programming
- Protection and access control
- Long term storage

**Memory management requirements**

- Process address space
  - Process runs in its private address space
  - In user mode, process refers to private stack, data, and code areas
  - In kernel mode, process refers to kernel data and code areas and uses a different private stack
  - Processes may need to access address space that is shared among processes
    - In some cases, this is achieved by explicit requests (shared memory)
    - In other cases, it may be done automatically by kernel to reduce memory usage (editors)
- Relocation
  - Available main memory shared by a number of processes
  - Programmer may not know of the other programs resident in memory while his code is executing
  - Processes are swapped in/out to maximize CPU utilization
  - Process may not get swapped back into the same memory location; need to *relocate* the process to a different area of memory
  - All memory references need to be resolved to correct addresses
- Protection
  - Processes need to be protected against unwanted interference by other processes
  - Requirement for relocation makes it harder to satisfy protection because the location of a process in memory is unpredictable
  - Impossible to check the absolute addresses at compile time
  - Problem of dealing with dynamic memory allocation through pointers
    - Avoid accessing code or data of OS or other processes, or branch into those
    - Need hardware support to protect processes from interfering with one another
- Sharing
- Logical organization
• Physical organization

Preparing a program for execution

• Development of programs
  – *Source* program
  – Compilation/Assembly to get *Object* program
  – Linking / Linkage editors to get *Relocatable load module*
  – Loading to get *Load module*

• Memory (RAM)
  – Physical viewpoint
    – A physical chip attached to the motherboard
  – Programmer’s viewpoint
    – Large array of words (or bytes)
    – Unique address of each word
    – CPU fetches from and stores into memory addresses
    – Is it right?
      • What if the compiler uses a register to store a variable?
      • What if it does not even allocate space for the variable (in case it is an unused variable)?
  – OS viewpoint
    – Both physical and programmer viewpoints are wrong
    – RAM is a computer subsystem comprising a variety of interacting physical components that, together, create a complex hierarchy of logical abstractions

• Memory hierarchy managers (in Unix and Windows)
  – Heap manager, or dynamic memory manager
    – Supports basic operations with memory blocks
    – Operations include block allocation, block release, changing size of allocated memory block, and so on
  – Virtual memory manager
    – At a lower level compared to heap manager
    – Virtual address spaces
      • Abstracts from physical addresses, allowing for arbitrary addresses to be assigned to memory cells
      • Allows several concurrently running applications to be loaded to same logical/virtual addresses, without any overlap of address space
      • Allows for complete separation of address space unless applications use shared memory/IPC
      • Kernel and memory management unit cooperate to find the actual physical location from the requested memory item
    – Virtual memory
      • Extension of the idea of virtual address space
      • Any memory cell can reside in both main memory and hard disk
      • Allows for practically unlimited amounts of memory to be allocated
      • Windows provides each process with 2GB of memory; Windows NT Enterprise Server does not impose any limit

• Physical view of memory hierarchy
Memory Management

- Processor does not interact directly with memory or hard disk
- Memory interaction is achieved through memory controller
- Disk interaction is achieved through disk controller
- Both memory and disk controllers are integrated into chipset
- System memory
  - Based on Dynamic RAM, slow
- Cache memory
  - High speed memory closer to the CPU
  - Typically provided as Static RAM while the main memory is Dynamic RAM, for the reason of cost
  - Not available to programmer
  - Contents cannot be directly read/modified
  - Cache controller
    - Manages cache memory, instead of it being managed by CPU
    - Responsible to accumulate important data within cache memory while clearing away unneeded data
    - Current CPUs may have cache controller integrated into them
  - L1 cache
    - Extremely fast (near CPU speed)
    - Reads in the order of 2-3 cycles (reads in RAM typically 30-80 cycles but can go as high as up to 250 cycles)
  - L2 cache
    - Contains data flushed out of L1 cache, therefore also termed as victim cache
    - Access times in the range of 9-15 cycles

• Instruction execution cycle
  - Fetch an instruction (opcode) from memory
  - Decode instruction
  - Fetch operands from memory, if needed
  - Execute instruction
  - Store results into memory, if necessary

• Memory unit sees only the addresses, and not how they are generated (instruction counter, indexing, direct)

• Address Binding
  - Binding – Mapping from one address space to another
  - Program must be loaded into memory before execution
  - Loading of processes may result in relocation of addresses
    - Link external references to entry points as needed
  - User process may reside in any part of the memory
  - Symbolic addresses in source programs (like $i$)
  - Compiler binds symbolic addresses to relocatable addresses, assumed to start at location zero
  - Linkage editor or loader binds relocatable addresses to absolute addresses
  - Types of binding
    - Compile time binding
      - Binding of absolute addresses by compiler
      - Possible only if compiler knows the memory locations to be used
· MS-DOS .com format programs

* Load time binding
  · Based on relocatable code generated by the compiler
  · Final binding delayed until load time
  · If change in starting address, reload the user code to incorporate address changes

* Execution time binding
  · Process may be moved from one address to another during execution
  · Binding must be delayed until run time
  · Requires special hardware

● Relocation

  – Compiler may work with assumed logical address space when creating an object module
  – Relocation – Adjustment of operand and branch addresses within the program
  – Static Relocation
    * Similar to compile time binding
    * Internal references
      · References to locations within the same program address space
      · Earliest possible moment to perform binding at the time of compilation
      · If bound at compile time, compiler must have the actual starting address of object module
      · Early binding is restrictive and rarely used, but may result in better run-time performance, specially in loops

```c
int x[5], i, j;
for (i = 0; i < n; i++)
  for (j = 0; j < 5; j++)
    x[j] = /* some computation */

/* Other part of loop using x */
```

* External references
  · References to locations within the address space of other modules or functions
  · More practical
  · All modules to be linked together must be known to resolve references
  · Linking loader
    · With static relocation, load module is not relocatable and the loader simply copies the load module into memory
  * Separate linkage editor and loader
    · More flexible
    · Starting address need not be known at linking time
    · Absolute physical addresses bound only at the time of loading
    · Relocatable physical addresses bound by relocating the complete module
    · The program gets relocated twice – once for linking and then for loading

– Dynamic Relocation

* All object modules kept on disk in relocatable load format
* Relocation at runtime immediately precedes each storage reference
* Invisible to all users (except for system programmers)
* Forms the basis for virtual memory
* Permits efficient use of main storage
* Binding of physical addresses can be delayed to the last possible moment
* When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded into memory
Relocatable linking loader can load the new routine if needed
Unused routine is never loaded
Useful to handle large amount of infrequently used code (like error handlers)

- Linking
  - Allows independent development and translation of modules
  - Compiler generates *external symbol table*
  - Resolution of external references
    - Chain method
      - Using chain of pointers in the module
      - Chain headed by corresponding entry in the external symbol table
      - Resolution by linking at the end (when the address is known) through external symbol table
      - Last element in the chain is NULL
      - External symbol table not a part of the final code
    - Indirect addressing
      - External symbol table a permanent part of the program
      - Transfer vector approach
      - Each reference is set to point to the entry in the external symbol table
      - External symbol table reflects the actual address of the reference when known
      - Linking faster and simpler than chaining method but two memory references required at run time to complete the reference

- Static Linking
  - All external references are resolved before program execution

- Dynamic Linking
  - External references resolved during execution
  - *Dynamically linked libraries*
    - Particularly useful for system libraries
    - Programs need to have a copy of the language library in the executable image, if no dynamic linking
    - Include a stub in the image for each library-routine reference
    - Stub
      - Small piece of code
      - Indicates the procedures to locate the appropriate memory resident library routine
      - Upon execution, stub replaces itself with the routine and executes it
      - Repeated execution executes the library routine directly
      - Useful in library updates or bug fixes
        - A new version of the library does not require the programs to be relinked
        - Library version numbers to ensure compatibility

Memory alignment
- Choice between unaligned loads and unaligned stores
- Pick unaligned loads
  - Unaligned stores may overwrite data adjacent to target
  - Problem can be avoided by extra overhead

Implementation of memory management
• Done through memory tables to keep track of both real (main) as well as virtual (secondary) memory

• Memory management unit (MMU)
  – Identifies a memory location in ROM, RAM, or I/O memory given a physical address
  – Does not translate physical address
  – Physical address is provided by an address bus to initiate the movement of code or data from one platform device to another
  – Physical address is generated by devices that act as bus masters on the address buses (such as CPU)
  – Frame buffers and simple serial ports are slave devices that respond to the addresses

• In Unix, memory can be accessed as a device
  – Entails a larger overhead in doing so
  – You have to go through the I/O system through a system call to talk to memory, rather than doing it directly by one machine instruction
  – Useful only when trying to access otherwise protected memory
  – `ps` digs up information about other processes by reading `/dev/kmem` (kernel memory) and `/dev/mem` (physical memory)

• RAM usage in Unix
  – A portion of RAM dedicated to storing kernel image (code and static data structures)
  – Remaining portion handled by virtual memory system and used in three possible ways
    1. Kernel requests for buffers, descriptors, and other dynamic kernel data structures
    2. Process requests for generic memory areas and for memory mapping of files
    3. To get better performance from disks and other buffered devices by using as cache
  – Kernel often forced to use physically contiguous memory areas
    * Memory request could fail if there is enough memory available but not available as a contiguous chunk

**Simple Memory Management Schemes**

• Shortage of main memory due to
  – Size of many applications
  – Several active processes may need to share memory at the same time

• Fixed Partition Memory Management
  – Simplest memory management scheme for multiprogrammed systems
  – Divide memory into fixed size partitions, possibly of different size
  – Partitions fixed at system initialization time and may not be changed during system operation
  – Single-Partition Allocation
    * User is provided with a bare machine
    * User has full control of entire memory space
    * Advantages
      · Maximum flexibility to the user
      · User controls the use of memory as per his own desire
      · Maximum possible simplicity
      · Minimum cost
      · No need for special hardware
- No need for operating system software
  - Disadvantages
    - No services
    - OS has no control over interrupts
    - No mechanism to process system calls and errors
    - No space to provide multiprogramming
- Two-Partition Allocation
  - Memory divided into two partitions
    - Resident operating system
    - User memory area
  - Linux divides the memory into kernel space and user space
  - OS placed in low memory or high memory depending upon the location of interrupt vector
  - Need to protect OS code and data from changes by user processes
    - Protection must be provided by hardware
    - Can be implemented by using base-register and limit-register
- Loading of user processes
  - First address of user space must be beyond the base register
  - Any change in base address requires recompilation of code
  - Could be avoided by having relocatable code from the compiler
  - Base value must be \textit{static} during program execution
  - OS size cannot change during program execution
    - Change in buffer space for device drivers
    - Loading code for rarely used system calls
      - \textit{Transient} OS code
  - Handling transient code
    - Load user processes into high memory down to base register
      - Allows the use of all available memory
    - Delay address binding until execution time
      - Base register known as the \textit{relocation register}
      - Value in base register added to every address reference
      - User program never sees the real physical addresses
      - User program deals only with logical addresses
- Multiple-Partition Allocation
  - Necessary for multiprogrammed systems
  - Allocate memory to various processes in the wait queue to be brought into memory
  - Simplest scheme
    - Divide memory into a large number of fixed-size partitions
    - One process to each partition
    - Degree of multiprogramming bound by the number of partitions
    - Partitions allocated to processes and released upon process termination
    - Originally used by IBM OS/360 (MFT)
    - Primarily useful in a batch environment
  - Variable size partitions – Basic implementation
    - Keep a table indicating the availability of various memory partitions
    - Any large block of available memory is called a \textit{hole}
Memory Management

* Initially the entire memory is identified as a large hole
* When a process arrives, the allocation table is searched for a large enough hole and if available, the hole is allocated to the process
* Example
  - Total memory available – 2560K
  - Resident OS – 400K
  - User memory – 2160K

<table>
<thead>
<tr>
<th>Process</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td>600K</td>
<td>10</td>
</tr>
<tr>
<td>p₂</td>
<td>1000K</td>
<td>5</td>
</tr>
<tr>
<td>p₃</td>
<td>300K</td>
<td>20</td>
</tr>
<tr>
<td>p₄</td>
<td>700K</td>
<td>8</td>
</tr>
<tr>
<td>p₅</td>
<td>500K</td>
<td>15</td>
</tr>
</tbody>
</table>

- Set of holes of various sizes scattered throughout the memory
- Holes can grow when jobs in adjacent holes are terminated
- Holes can also diminish in size if many small jobs are present
- Problem of fragmentation
  * Division of main memory into small holes not usable by any process
  * Enough total memory space exists to satisfy a request but is fragmented into a number of small holes
  * Fragmentation is caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy that request
  * Possibility of starvation for large jobs

- Used by IBM OS/MVT (multiprogramming with variable number of tasks, 1969)
- Dynamic storage allocation problem
  * Selects a hole to which a process should be allocated
  * First-fit strategy
    - Allocate first hole that is big enough
    - Stop searching as soon as first hole large enough to hold the process is found
  * Best-fit strategy
    - Allocate the smallest hole that is big enough
    - Entire list of holes is to be searched
    - Search of entire list can be avoided by keeping the list of holes sorted by size
  * Worst-fit strategy
    - Allocate the largest available hole
    - Similar problems as the best-fit approach

- Memory Compaction
  * Shuffle the memory contents to place all free memory into one large hole
  * Possible only if the system supports dynamic relocation at execution time
  * Total compaction
  * Partial compaction
  * Dynamic memory allocation in C
· C heap manager is fairly primitive
· The *malloc* family of functions allocates memory and the heap manager takes it back when it is freed
· There is no facility for heap compaction to provide for bigger chunks of memory
· The problem of fragmentation is for real in C because movement of data by a heap compactor can leave incorrect address information in pointers
· Microsoft Windows has heap compaction built in but it requires you to use special memory handles instead of pointers
· The handles can be temporarily converted to pointers, after locking the memory so the heap compactor cannot move it

- Overlays
  · Size of process is limited to size of available memory
  · Technique of *overlaying* employed to execute programs that cannot be fit into available memory
  · Keep in memory only those instructions and data that are needed at any given time
  · When other instructions are needed, they are loaded into space previously occupied by instructions that are not needed
  · A 2-pass assembler

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>70K</th>
<th>Generate symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 2</td>
<td>80K</td>
<td>Generate object code</td>
</tr>
<tr>
<td>Symbol table</td>
<td>20K</td>
<td></td>
</tr>
<tr>
<td>Common routines</td>
<td>30K</td>
<td></td>
</tr>
<tr>
<td><strong>Total memory requirement – 200K</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Available memory – 150K</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Divide the task as into overlay segments

<table>
<thead>
<tr>
<th>Overlay 1</th>
<th>Overlay 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1 code</td>
<td>Pass 2 code</td>
</tr>
<tr>
<td>Symbol table</td>
<td>Symbol table</td>
</tr>
<tr>
<td>Common routines</td>
<td>Common routines</td>
</tr>
<tr>
<td><strong>Overlay driver (10K)</strong></td>
<td><strong>Overlay driver (10K)</strong></td>
</tr>
</tbody>
</table>

- Code for each overlay kept on disk as absolute memory images
- Requires special relocation and linking algorithms
- No special support required from the OS
- Slow to execute due to additional I/O to load memory images for different parts of the program
- Programmer completely responsible to define overlays

**Virtual memory**
· Large address space: Makes the system appears as if it has more memory than it actually has
· Protection: Each process has its own virtual address space, completely separated from other processes; areas of memory can be protected against writing allowing code and data to be protected against accidental overwriting by rogue applications
· Memory mapping: Map code image and data files into a process’ address space; file contents are linked directly into the virtual address space of a process
· Fair physical memory allocation: Fair share of physical memory for each process
· Shared virtual memory: Controlled sharing of code (bash shell, dynamically loaded libraries) and data
Principles of Virtual Memory

- Hide the real memory from the user
- Logical layer between the application memory requests and the hardware memory management unit
- Advantages
  - Several processes can be executed concurrently
  - Possible to run applications requiring more memory than what is available
  - Processes can execute a program whose code is only partially loaded in memory
  - Each process allowed to access a subset of available physical memory
  - Processes can share a single memory image of a library or program
  - Programs can be relocatable
  - Programmers do not have to worry about physical memory organization

- Process virtual address space handling
  - Kernel stores virtual address space as a list of memory area descriptors
  - Upon execution of a process through `exec(2)`, kernel assigns a virtual address space with memory areas for
    * Executable code of program
    * Initialized data of program
    * Uninitialized data of program
    * Initial program stack (user mode stack)
    * Executable code and data of needed shared libraries
    * Heap

- Copy-on-write
  - Upon `fork()`, kernel just assigns the parent’s page frames to the child address space, marking them read-only
  - When the parent or child tries to modify the page contents, an exception is raised
  - Exception handler assigns a new frame to the affected process and initializes it with the contents of original page

- Swapping
  - Remove a process temporarily from main memory to a backing store and later, bring it back into memory for continued execution
  - Swap-in and Swap-out
  - Suitable for round-robin scheduling by swapping processes in and out of main memory
    * Quantum should be large enough such that swapping time is negligible
    * Reasonable amount of computation should be done between swaps
  - Roll-out, Roll-in Swapping
    * Allows to preempt a lower priority process in favor of a higher priority process
    * After the higher priority process is done, the preempted process is continued
  - Process may or may not be swapped into the same memory space depending upon the availability of execution time binding
  - Backing store
    * Preferably a fast disk
    * Must be large enough to accommodate copies of all memory images of all processes
    * Must provide direct access to each memory image
    * Maintain a ready queue of all processes on the backing store
* Dispatcher brings the process into memory if needed
  – Calculation of swap time
    * User process of size 100K
    * Backing store – Standard head disk with a transfer rate of 1 MB/sec
    * Actual transfer time – 100 msec
    * Add latency (8 msec) – 108 msec
    * Swap-in + Swap-out – 216 msec
  – Total transfer time directly proportional to the amount of memory swapped
  – Swap only completely idle processes (not with pending I/O)

• Multiple Base Registers
  – Provides a solution to the fragmentation problem
  – Break the memory needed by a process into several parts
  – One base register corresponding to each part with a mechanism to translate from logical to physical address

• Paging
  – Permits a process’ memory to be noncontiguous
  – Avoids the problem of fitting varying-sized memory segments into backing store
  – First used in ATLAS computer in 1962
  – Physical memory is divided into a number of equal-sized contiguous blocks, called page frames
    * In the past, common block sizes were 512 bytes or 1K
    * Common block sizes have been increasing in size, with Solaris allowing for a frame size of 8K
    * Block size in Unix can be determined by the C library function getpagesize(3C), or user command pagesize(1)
  – Hardware requirements
    * Logical memory broken into blocks of same size as page frame size, called pages
    * To execute, pages of process are loaded into frames from backing store
    * Backing store divided into fixed size blocks of the same size as page or frame
    * Every address generated by the CPU divided into two parts
      • Page number \( p \)
      • Page offset \( d \)
    * Page number used as index into a page table
      • Page table contains the base address of each page in memory
    * Page offset defines the address of the location within the page
    * Page size \( 2^n \) bytes
      • Low order \( n \) bits in the address indicate the page offset
      • Remaining high order bits designate the page number
    * Example – Page size of four words and physical memory of eight pages
• Scheduling processes in a paged system
  
  – Each page an instance of memory resource
  – Size of a process can be expressed in pages
  – Available memory known from the list of unallocated frames
  – If the process’ memory requirement can be fulfilled, allocate memory to the process
  – Pages loaded into memory from the list of available frames
  – Example

<table>
<thead>
<tr>
<th>free-frame list</th>
<th>free-frame list</th>
<th>free-frame list</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>13</td>
<td>unused</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>unused</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>unused</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
<td>unused</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>unused</td>
</tr>
<tr>
<td>18</td>
<td>unused</td>
<td>15</td>
</tr>
<tr>
<td>19</td>
<td>new process</td>
<td>13</td>
</tr>
<tr>
<td>new process</td>
<td>19</td>
<td>page 1</td>
</tr>
<tr>
<td>page 0</td>
<td>20</td>
<td>page 0</td>
</tr>
<tr>
<td>page 1</td>
<td>21</td>
<td>page 1</td>
</tr>
<tr>
<td>page 2</td>
<td></td>
<td>page 2</td>
</tr>
<tr>
<td>page 3</td>
<td></td>
<td>page 3</td>
</tr>
</tbody>
</table>

new process page table

<table>
<thead>
<tr>
<th>page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

• No external fragmentation possible with paging
• Internal fragmentation possible – an average of half a page per process
• Paging allows programs larger than available main memory to be executed without using overlay structure, using a technique called demand paging
Memory Management

- Page size considerations
  - Small page size ⇒ More overhead in page table plus more swapping overhead
  - Large page size ⇒ More internal fragmentation
- Implementation of page table
  - Simplest implementation through a set of dedicated registers
    - Registers reloaded by the CPU dispatcher
    - Registers need to be extremely fast
    - Good strategy if the page table is very small (< 256 entries)
    - Not satisfactory if the page table size is large (like a million entries)
  - Page-Table Base Register
    - Suitable for large page tables
    - Pointer to the page table in memory
    - Achieves reduction in context switching time
    - Increase in time to access memory locations
  - Translation look-aside buffers (TLB)
    - Used to speed up linear address translation
    - When used for first time, the corresponding physical address is computed through slow access to page table in RAM
    - Physical address then stored in a TLB entry for use in further references
    - In multiprocessor systems, each CPU has its own local TLB
    - Unlike hardware cache, TLB entries do not need to be synchronized because processes on different CPUs may associate the same linear address with different physical ones

- Shared pages
  - Possible to share common code with paging
  - Shared code must be reentrant (pure code)
    - Reentrant code allows itself to be shared by multiple users concurrently
    - The code cannot modify itself and the local data for each user is kept in separate space
    - The code has two parts
      1. Permanent part is the instructions that make up the code
      2. Temporary part contains memory for local variables for use by the code
    - Each execution of the permanent part creates a temporary part, known as the activation record for the code
    - Requirements for a function to be classified as reentrant
      1. All data is allocated on the stack
      2. No global variables
      3. Function can be interrupted at any time without affecting its execution
      4. Function doesn’t call any other function that is not reentrant
  - Separate data pages for each process
  - Code to be shared – text editor, windowing system, compilers
  - Reentrant kernels
    - Unix kernels are reentrant ⇒ Several processes may be executing in kernel mode at the same time
    - Provided by writing reentrant functions (modify only local variables and do not alter global data structures)
    - Unix kernel may include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time

- Memory protection in paging systems
Memory Management

- Accomplished by protection bits associated with each frame
- Protection bits kept in page table
- Define the page to be read only or read and write
- Protection checked at the time of page table reference to find the physical page number
- Hardware trap or memory protection violation
- Page table length register
  - Indicates the size of the page table
  - Value checked against every logical address to validate the address
  - Failure results in trap to the OS

- Logical memory vs Physical memory
  - Logical memory
    - Provides user’s view of the memory
    - Memory treated as one contiguous space, containing only one program
  - Physical memory
    - User program scattered throughout the memory
    - Also holds other programs
    - Mapping from logical addresses to physical addresses hidden from the user
    - System could use more memory than any individual user
    - Allocation of frames kept in frame table

- Segmentation
  - User prefers to view memory as a collection of variable-sized segments, like arrays, functions, procedures, and main program
  - No necessary order in the segments
  - Length of each segment is defined by its purpose in the program
  - Elements within a segment defined by their offset from the beginning of segment
    - First statement of the procedure
    - Seventeenth entry in the symbol table
    - Fifth instruction of the sqrt function
  - Logical address space considered to be collection of segments
  - A name and length for each segment
  - Address - Segment name and offset within the segment
    - Segment name to be explicitly specified unlike paging
  - The only memory management scheme available on Intel 8086
    - Memory divided into code, data, and stack segments
    - Included in x86 architecture to encourage programmers to split their code into logically related entities
  - Hardware for segmentation
    - Mapping between logical and physical addresses achieved through a segment table
    - Each entry in segment table is made up of
      - Segment base
      - Segment limit
    - Segment table can be abstracted as an array of base-limit register pairs
    - Two parts in a logical address
Memory Management

1. Segment name/number $s$
   - Used as an index into the segment table
2. Segment offset $d$
   - Added to the segment base to produce the physical address
   - Must be between 0 and the segment limit
   - Attempt to address beyond the segment limit results in a trap to the OS

- Implementation of segment tables
  - Kept either in registers or in memory
  - Segment table base register
  - Segment table length register
  - Associative registers to improve memory access time

- Protection and sharing
  - Segments represent a semantically defined portion of a program
  - Protection and sharing like paging
  - Possible to share parts of a program
    - Share the \texttt{sqrt} function segment between two independent programs

- Fragmentation
  - Memory allocation becomes a dynamic storage allocation problem
  - Possibility of external fragmentation
    - All blocks of memory are too small to accommodate a segment
  - Compaction can be used whenever needed (because segmentation is based on dynamic relocation)
  - External fragmentation problem is also dependent on average size of segments

- Paged segmentation
  - Used in the MULTICS system, and also in Linux
  - Page the segments
  - Separate page table for each segment
  - Segment table entry contains the base address of a page table for the segment
  - Segment offset is broken into page number and page offset
  - Page number indexes into page table to give the frame number
  - Frame number is combined with page offset to give physical address
  - MULTICS had 18 bit segment number and 16 bit offset
  - Segment offset contained 6-bit page number and 10-bit page offset
  - Each segment limited in length by its segment-table entry
    - Page table need not be full-sized; Requires only as many entries as needed
  - Linux has one memory segment for the kernel and one for each of the processes
    - Each process can access only its own memory segment
    - When you write a kernel module you access the kernel memory segment
      - Handled automatically by the system
    - To pass contents between user segment and kernel segment, kernel receives a pointer to a buffer which is in the process segment
      - Accessed by \texttt{put_user} and \texttt{get_user} macros (one character at a time)
      - Several characters can be transferred by using the macros \texttt{copy_to_user} and \texttt{copy_from_user}
      - For write function, you need to import data from user space
      - For read function, you do not need to transfer data because data is already in kernel space
– On an average, half a page of internal fragmentation per segment
– Eliminated external fragmentation but introduced internal fragmentation and increased table space overhead

Implementation of Virtual Memory

• Allows the execution of processes that may not be completely in main memory
• Programs can be larger than the available physical memory

• Motivation
  – The entire program may not need to be in the memory for execution
  – Code to handle unusual conditions may never be executed
  – Complex data structures are generally allocated more memory than needed (like symbol table)
  – Certain options and features of a program may be used rarely, like text-editor command to change case of all letters in a file

• Benefits of virtual memory
  – Programs not constrained by the amount of physical memory available
  – User does not have to worry about complex techniques like overlays
  – Increase in CPU utilization and throughput (more programs can fit in available memory)
  – Less I/O needed to load or swap user programs

• Demand Paging
  – Similar to a paging system with swapping
  – Rather than swapping the entire process into memory, use a “lazy swapper”
  – Never swap a page into memory unless needed
  – Distinction between swapper and pager
    * Swapper swaps an entire process into memory
    * Pager brings in individual pages into memory
  – In the beginning, pager guesses the pages to be used by the process
  – Pages not needed are not brought into memory
  – Hardware support for demand paging
    * An extra bit attached to each entry in the page table – *valid-invalid bit*
    * This bit indicates whether the page is in memory or not
    * Example

```
<table>
<thead>
<tr>
<th>Frame</th>
<th>Frame</th>
<th>Frame</th>
<th>Frame</th>
<th>Frame</th>
<th>Frame</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical memory</th>
<th>Logical memory</th>
<th>Logical memory</th>
<th>Logical memory</th>
<th>Logical memory</th>
<th>Logical memory</th>
<th>Logical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
</tr>
<tr>
<td>Table</td>
<td>Table</td>
<td>Table</td>
<td>Table</td>
<td>Table</td>
<td>Table</td>
<td>Table</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
</tr>
</tbody>
</table>
```

Physical memory
For memory-resident pages, execution proceeds normally

If page not in memory, a page fault trap occurs

Upon page fault, the required page brought into memory

- Check an internal table to determine whether the reference was valid or invalid memory access
- If invalid, terminate the process. If valid, page in the required page
- Find a free frame (from the free frame list)
- Schedule the disk to read the required page into the newly allocated frame
- Modify the internal table to indicate that the page is in memory
- Restart the instruction interrupted by page fault

Pure demand paging – Don’t bring even a single page into memory

Locality of reference

Performance of demand paging

- Effective access time for demand-paged memory
  - Usual memory access time \((m)\) – 10 to 200 nsec
  - No page faults ⇒ Effective access time same as memory access time
  - Probability of page fault = \(p\)
  - \(p\) expected to be very close to zero so that there are few page faults
  - Effective access time = \((1 - p) \times m + p \times \text{page fault time}\)
  - Need to know the time required to service page fault

What happens at page fault?

- Trap to the OS
- Save the user registers and process state
- Determine that the interrupt was a page fault
- Check that the page reference was legal and determine the location of page on the disk
- Issue a read from the disk to a free frame
  - Wait in a queue for the device until the request is serviced
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame
- While waiting, allocate CPU to some other process
- Interrupt from the disk (I/O complete)
- Save registers and process state for the other user
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show that the desired page is now in memory
- Wait for the CPU to be allocated to the process again
- Restore user registers, process state, and new page table, then resume the interrupted instruction

Computation of effective access time

- Bottleneck in read from disk
  - Latency time – 8 msec
  - Seek time – 15 msec
  - Transfer time – 1 msec
  - Total page read time – 24 msec
- About 1 msec for other things (page switch)
Memory Management

- Average page-fault service time – 25 msec
- Memory access time – 100 nanosec
- Effective access time
  \[ (1 - p) \times 100 + p \times 25,000,000 \]
  \[ = 100 + 24,999,900 \times p \]
  \[ \approx 25 \times p \text{msec} \]

- Assume 1 access out of 1000 to cause a page fault
  * Effective access time – 25 µsec
  * Degradation due to demand paging – 250%
- For 10% degradation
  \[ 110 > 100 + 25,000,000 \times p \]
  \[ 10 > 25,000,000 \times p \]
  \[ p < 0.0000004 \]

Reasonable performance possible through less than 1 memory access out of 2,500,000 causing a page fault

Page Replacement

- Limited number of pages available in memory
- Need to optimize swapping (placement and replacement)
- Increase in multiprogramming through replacement optimization
- Increase in degree of multiprogramming \(\Rightarrow\) overallocation of memory
- Assume no free frames available
  - Option to terminate the process
    * Against the philosophy behind virtual memory
    * User should not be aware of the underlying memory management
  - Option to swap out a process
    * No guarantee that the process will get the CPU back pretty fast
  - Option to replace a page in memory
- Modified page-fault service routine
  - Find the location of the desired page on the disk
  - Find a free frame
    * If there is a free frame, use it
    * Otherwise, use a page replacement algorithm to find a victim frame
    * Write the victim page to the disk; change the page and frame tables accordingly
  - Read the desired page into the (newly) free frame; change the page and frame tables
  - Restart the user process
- No free frames \(\Rightarrow\) two page transfers
- Increase in effective access time
- **Dirty bit**
  - Also known as modify bit
  - Each frame has a dirty bit associated with it in hardware
Memory Management

– Dirty bit is set if the page has been modified or written into
– If the page is selected for replacement
  * Check the dirty bit associated with the frame
  * If the bit is set write it back into its place on disk
  * Otherwise, the page in disk is same as the current one

• Page replacement algorithms
  – Aim – to minimize the page fault rate
  – Evaluate an algorithm by running it on a particular string of memory references and compute the number of page faults
  – Memory references string called a reference string
  – Consider only the page number and not the entire address
  – Address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

– 100 byte to a page
– Reference string

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

– Second factor in page faults – Number of pages available
– More the number of pages, less the page faults
– FIFO Replacement Algorithm
  * Associate with each page the time when that page was brought in memory
  * The victim is the oldest page in the memory
  * Example reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

* With three pages, page faults as follows:

<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>7</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>0</th>
<th>0</th>
<th>7</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* May reduce a page that contains a heavily used variable that was initialized a while back
* Bad replacement choice ⇒ Increase in page fault rate
* Consider another reference string

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

* With three pages, the page faults are:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

* Belady’s Anomaly – For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases
Memory Management

- Optimal Page Replacement Algorithm
  * Also called OPT or MIN
  * Has the lowest page-fault rate of all algorithms
  * Never suffers from Belady’s Anomaly
  * “Replace the page that will not be used for the longest period of time”
  * Example reference string
    7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  * With three pages, page faults as follows:
    | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 7 |
    |---|---|---|---|---|---|---|---|---|
    | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
    | 1 | 1 | 3 | 3 | 3 | 1 | 1 |
  * Guarantees the lowest possible page-fault rate of all algorithms
  * Requires future knowledge of page references
  * Mainly useful for comparative studies with other algorithms

- LRU Page Replacement Algorithm
  * Approximation to the optimal page replacement algorithm
  * Replace the page that has not been used for the longest period of time
  * Example reference string
    7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  * With three pages, page faults as follows:
    | 7 | 7 | 7 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 |
    |---|---|---|---|---|---|---|---|---|---|---|---|---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 0 |
    | 1 | 1 | 3 | 3 | 2 | 2 | 2 | 2 | 7 |
  * Implementation may require substantial hardware assistance
  * Problem to determine an order for the frame defined by the time of last use
  * Implementations based on counters
    - With each page-table entry, associate a time-of-use register
    - Add a logical clock or counter to the CPU
    - Increment the clock for every memory reference
    - Copy the contents of logical counter to the time-of-use register at every page reference
    - Replace the page with the smallest time value
    - Times must be maintained when page tables are changed
    - Overflow of the clock must be considered
  * Implementations based on stack
    - Keep a stack of page numbers
    - Upon reference, remove the page from stack and put it on top of stack
    - Best implemented by a doubly linked list
    - Each update expensive but no cost for search (for replacement)
    - Particularly appropriate for software or microcode implementations

- Stack algorithms
  - Class of page replacement algorithms that never exhibit Belady’s anomaly
  - Set of pages in memory for \( n \) frames is always a subset of the set of pages in memory with \( n + 1 \) frames

- LRU Approximation Algorithms
  - Associate a reference bit with each entry in the page table
  - Reference bit set whenever the page is referenced (read or write)
– Initially, all bits are reset
– After some time, determine the usage of pages by examining the reference bit
– No knowledge of order of use
– Provides basis for many page-replacement algorithms that approximate replacement

– Additional-Reference-Bits Algorithm
  * Keep an 8-bit byte for each page in a table in memory
  * At regular intervals (100 msec), shift the reference bits by 1 bit to the right
  * 8-bit shift-registers contain the history of page reference for the last eight time periods
  * Page with lowest number is the LRU page
  * 11000100 used more recently than 01110111
  * Numbers not guaranteed to be unique

– Second-Chance Algorithm
  * Basically a FIFO replacement algorithm
  * When a page is selected, examine its reference bit
  * If reference bit is 0, replace the page
  * If reference bit is 1, give the page a second chance and proceed to select the next FIFO page
  * To give second chance, reset the reference bit and set the page arrival time to current time
  * A frequently used page is always kept in memory
  * Commonly implemented by a circular queue
  * Worst case when all bits are set (degenerates to FIFO replacement)
  * Performance evaluation with reference string 7, 0, 1, 2, 0, 3, 0, 4, 2 the reference bit as -1 or -0 at each time instance.

+ - The bits got reset and there was no page fault.
++ - Page 0 got reference bit changed back to 1; no page fault.
* - Used FIFO replacement by replacing the oldest page, whose reference bit is 0.

– LFU Algorithm
  * Keep counter of number of references made to each page
  * Replace the page with the smallest count
  * Motivation – Actively used page has a large reference count
  * What if a page is heavily used initially but never used again
  * Solution – Shift the counts right at regular intervals forming a decaying average usage count

– MFU Algorithm
  * Page with smallest count was probably just brought into memory and is yet to be used
  * Implementation of LFU and MFU fairly expensive, and they do not approximate OPT very well

– Using used bit and dirty bit in tandem
  * Four cases
    · (0,0) neither used nor modified
    · (0,1) not used (recently) but modified
    · (1,0) used but clean
    · (1,1) used and modified
  * Replace a page within the lowest class
  * Also called MESI protocol (Modified, Exclusive, Shared, Invalid)

Allocation of Frames
Memory Management

- No problem with the single user virtual memory systems
- Problem when demand paging combined with multiprogramming
- Minimum number of frames
  - Cannot allocate more than the number of available frames (unless page sharing is allowed)
  - As the number of frames allocated to a process decreases, page fault rate increases, slowing process execution
  - Minimum number of frames to be allocated defined by instruction set architecture
    - Must have enough frames to hold all the different pages that any single instruction can reference
    - The instruction itself may go into two separate pages
  - Maximum number of frames defined by amount of available physical memory
- Allocation algorithms
  - Equal allocation
    * m frames and n processes
    * Allocate \( m/n \) frames to each process
    * Any leftover frames given to free frame buffer pool
  - Proportional allocation
    * Allocate memory to each process according to its size
    * If size of virtual memory for process \( p_i \) is \( s_i \), the total memory is given by \( S = \sum s_i \)
    * Total number of available frames \( m \)
    * Allocate \( a_i \) frames to process \( p_i \) where
      \[
      a_i = \frac{s_i}{S} \times m
      \]
    * \( a_i \) must be adjusted to an integer, greater than the minimum number of frames required by the instruction set architecture, with a sum not exceeding \( m \)
    * Split 62 frames between two processes – one of 10 pages and the other of 127 pages
    - First process gets 4 frames and the second gets 57 frames
  - Allocation dependent on degree of multiprogramming
  - No consideration for the priority of the processes

Thrashing

- Number of frames allocated to a process falls below the minimum level ⇒ Suspend the process’s execution
- Technically possible to reduce the allocated frames to a minimum for a process
- Practically, the process may require a higher number of frames than minimum for effective execution
- If process does not get enough frames, it page-faults quickly and frequently
- Need to replace a page that may be in active use
- Thrashing – High paging activity
- Process thrashing if it spends more time in paging activity than in actual execution
- Cause of thrashing
  - OS monitors CPU utilization
  - Low CPU utilization ⇒ increase the degree of multiprogramming by introducing new process
  - Use a global page replacement algorithm
May result in increase in paging activity and thrashing
- Processes waiting for pages to arrive leave the ready queue and join the wait queue
- CPU utilization drops
- CPU scheduler sees the decrease in CPU utilization and increases the degree of multiprogramming
- Decrease in system throughput
- At a certain point, to increase the CPU utilization and stop thrashing, we must decrease the degree of multiprogramming
- Effect of thrashing can be reduced by local replacement algorithms
  * A thrashing process cannot steal frames from another process
  * Thrashing processes will be in queue for paging device for more time
  * Average service time for a page fault will increase
  * Effective access time will increase even for processes that are not thrashing
- Locality model of process execution
  * Memory references tend to cluster
  * Clusters change over long time but for short term, a process works in a limited amount of memory/cluster
  * Technique to guess the number of frames needed by a process
  * As a process executes, it moves from locality to locality
  * Locality – Set of pages that are generally used together
- Allocate enough frames to a process to accommodate its current locality

- Working-Set Model
  - Based on the presumption of locality
  - Uses a parameter $\Delta$ to define working-set window
  - Set of pages in the most recent $\Delta$ page reference is the working set
  - Storage management strategy
    * At each reference, the current working set is determined and only those pages belonging to the working set are retained
    * A program may run if and only if its entire current working set is in memory
  - Actively used page $\in$ working set
  - Page not in use drops out after $\Delta$ time units of non-reference
  - Example
    * Memory reference string
      
      \[
      \begin{array}{lllllllllll}
        2 & 6 & 1 & 5 & 7 & 7 & 7 & 5 & 1 & \Delta_{t_1} \\
        6 & 2 & 3 & 4 & 1 & & & & & \\
        2 & 3 & 4 & 4 & 4 & 3 & 4 & 4 & 4 & \Delta_{t_2} \\
        1 & 3 & 2 & 3 & 4 & 4 & 3 & 4 & 4 & 4 \\
        \ldots
      \end{array}
      \]
    * If $\Delta = 10$ memory references, the working set at time $t_1$ is $\{1, 2, 5, 6, 7\}$ and at time $t_2$, it is $\{2, 3, 4\}$
  - Accuracy of working set dependent upon the size of $\Delta$
  - Let $WSS_i$ be the working set size for process $p_i$
  - Then, the total demand for frames $D$ is given by $\sum WSS_i$
  - Thrashing occurs if $D > m$
  - OS monitors the working set of each process
    * Allocate to each process enough frames to accommodate its working set
Memory Management

- Enough extra frames ⇒ initiate another process
- \( D > m \) ⇒ select a process to suspend
- Working set strategy prevents thrashing while keeping the degree of multiprogramming high
- Optimizes CPU utilization
- Problem in keeping track of the working set
  * Can be solved by using a timer interrupt and a reference bit
  * Upon timer interrupt, copy and clear the reference bit value for each page

- Page-fault frequency
  - More direct approach than working set model
  - Measures the time interval between successive page faults
  - Page fault rate for a process too high ⇒ the process needs more pages
  - Page fault rate too low ⇒ process may have too many frames
  - Establish upper and lower-level bounds on desired page fault rate
  - If the time interval between the current and the previous page fault exceeds a pre-specified value, all the pages not referenced during this time are removed from the memory
  - PFF guarantees that the resident set grows when page faults are frequent, and shrinks when the page fault rate decreases
  - The resident set is adjusted only at the time of a page fault (compare with the working set model)

- Prepping
  - Bring into memory at one time all pages that will be needed
  - Relevant during suspension of a process
  - Keep the working set of the process
  - Rarely used for newly created processes

- Page size
  - Size of page table
    * Smaller the page size, larger the page table
    * Each process must have its own copy of the page table
  - Smaller page size, less internal fragmentation
  - Time required to read/write a page in disk
  - Smaller page size, better resolution to identify working set
  - Trend toward larger page size
    * Intel 80386 – 4K page
    * Motorola 68030 – variable page size from 256 bytes to 32K

- Program structure
  - Careful selection of data structures
  - Locality of reference

Memory addressing in x86 architecture

- Logical address
  - Included in machine language instructions to specify an address
Memory Management

Embodies the x86 segmented architecture that forces programmers to divide their programs into segments
Each address specified as segment and offset

- Linear/Virtual address
  - 32-bit unsigned integer, to address up to 4GB
  - Typically represented in hex notation

- Physical address
  - Address of a location on chip
  - Sent over the memory bus
  - Represented as 32-bit or 36-bit unsigned integers

- MMU transforms a logical address into a linear address using a hardware circuit called a segmentation unit
- A second hardware circuit called a paging unit transforms linear address into physical address

- Multiprocessor systems
  - All CPUs access RAM chips concurrently
  - But read/write operations on a RAM chip must be performed serially
  - Achieved by a memory arbiter between the bus and every RAM chip
    - Grant access to CPU if chip is free and delay access if chip is busy with another PE
    - Uniprocessor systems may include memory arbiter to resolve access between CPU and DMA
    - Not of much concern to programmer because it is managed by hardware

Segmentation in hardware

- Address translation in x86 architecture
  - Real mode
    - Maintains processor compatibility with older models (pre-80286)
    - Allows OS to bootstrap
  - Protected mode

- Segment selector and segmentation register
  - Two parts in logical address
    1. 16-bit segment identifier or segment selector
    2. 32-bit segment offset
  - Segment selector format
    - **Bit 0-1** Requestor privilege level (RPL)
    - **Bit 2** Table indicator (TI)
    - **Bit 3-15** Index
  - Six segment registers to hold segment selectors
    - **cs** Code segment register; program instructions
      - Also includes a 2-bit field to indicate the current privilege level (CPL) of CPU
      - Level 0 is the highest
      - Linux uses only level 0 (kernel mode) and level 3 (user mode)
    - **ss** Stack segment register; current program stack
    - **ds** Data segment register; global and static data
es, fs, gs  General purpose; arbitrary data segments

- Segment descriptors
  - 8-byte structure to describe segment characteristics
  - Stored in global descriptor table (GDT) or local descriptor table (LDT)
  - One GDT, with each process having its own copy of LDT
  - Address and size of GDT are contained in gdt descriptor register
  - Address and size of LDT in use are contained in ldt descriptor register
  - Fields in segment descriptor
    Base  Linear address of first byte of segment
    G  Granularity flag
      * 0 indicates the segment size in bytes
      * Otherwise, it is multiples of 4096 bytes
    Limit  Offset of last cell in segment; segment size
      * \( G = 0 \Rightarrow 1B \leq \text{size} \leq 1MB \)
      * \( GI = 0 \Rightarrow 4KB \leq \text{size} \leq 4GB \)
    S  System flag
      * 0 indicates system segment to hold critical data structures such as local descriptor table (LDT)
      * Otherwise, it holds user code or data segment
    Type  Segment type and access rights
    DPL  Descriptor privilege level
      * Restricts access to segment
      * Minimal CPU privilege level to access the segment
      * 0 for kernel mode privilege; 3 for user mode
    P  Segment-present flag
      * 0 if segment is not in main memory
      * Linux sets this flag to 1 because it never swaps out whole segments to disk
    D or B  Flag to indicate whether segment contains code or data
      * 0 if addresses used as segment offsets are 16-bit long
      * 1 if addresses used as segment offsets are 32-bit long
    AV  Ignored by Linux
- Code Segment Descriptor
  - May be included in GDT or LDT
  - The S flag is set (non-system segment)
- Data segment descriptor
  - May be included in GDT or LDT
  - The S flag is set (non-system segment)
  - Stack segments are implemented using generic data segments
- Task state segment descriptor (TSSD)
  - Used to save contents of processor registers
  - Appears only in GDT
  - Corresponding Type field has value 11 or 9 depending on whether the process is currently executing on CPU
  - The S flag is cleared (system segment)
- Local descriptor table descriptor (LDTT)
  - Refers to a segment containing an LDT
Memory Management

- Appears only in GDT
- Corresponding Type field has value 2
- The $S$ flag is cleared (system segment)

- Fast access to segment descriptors
  - Logical address consists of a 16-bit segment selector and a 32-bit offset
  - Segmentation registers store only the segment selector
  - Additional nonprogrammable register in x86 for each of the six programmable segmentation registers

- Each register contains the 8-byte segment descriptor specified by the segment selector contained in the corresponding segmentation register
- When a segment selector is loaded in a segmentation register, the corresponding segment descriptor is loaded from memory into the matching nonprogrammable register
- This allows for address translation in the segment without accessing the GDT or LDT in main memory
  - GDT/LDT are accessed only when the contents of segmentation register change

- Segment selector fields
  - **index** Segment descriptor entry in GDT or LDT
  - **TI** Table indicator; specifies if segment descriptor is in GDT (TI=0) or LDT (TI=1)
  - **RPL** Requestor privilege level; Current privilege level of CPU

- Segmentation unit
Memory Management

Segmentation in Linux

- Used in a limited way
- Treats segmentation and paging as redundant
  - Segmentation assigns a different linear address space to each process
  - Paging maps the same linear address space into different physical address spaces
- Linux prefers paging to segmentation
  - Simpler memory management when all processes use the same segment register values (same set of linear addresses)
  - Portability to a wide range of architectures
    - RISC architectures have limited support for segmentation
- 2.6 kernel uses segmentation only when required by the x86 architecture
- Processes in user mode use user code segment and user data segment
- Processes in kernel mode use kernel code segment and kernel data segment
- Segment descriptor fields for the four main Linux segments

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>G Limit</th>
<th>S</th>
<th>Type</th>
<th>DPL</th>
<th>D/B</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>user code</td>
<td>0x000000000</td>
<td>1</td>
<td>0xffff</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>user data</td>
<td>0x000000000</td>
<td>1</td>
<td>0xffff</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kernel code</td>
<td>0x000000000</td>
<td>1</td>
<td>0xffff</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kernel data</td>
<td>0x000000000</td>
<td>1</td>
<td>0xffff</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Management

- Linear address ranges from 0 to the addressing limit of $2^{32} - 1$
  - All processes may use the same logical address in both user and kernel mode
  - Also, logical address and linear address is the same (both start at 0)
  - Current privilege level (CPL) indicated by requestor privilege level (RPL) of segment selector stored in cs register
  - Change in CPL requires a corresponding change in other segmentation registers
    * CPL = 3 (user mode) changes ds register to user data segment, and ss register to user mode stack
    * CPL = 0 (kernel mode) changes ds register to kernel data segment, and ss register to kernel mode stack
  - Saving a pointer to an instruction or data structure does not require saving the segment selector component of logical address because the current segment selector is contained in the ss register

- Linux GDT
  - Only one GDT in uniprocessor systems but one GDT per CPU on multiprocessor systems
  - GDTS stored in cpu_gdt_table array
  - Each GDT has 18 segment descriptors and 14 null/unused/reserved entries
  - 18 segment descriptors point to
    * Four user and kernel code and data segments
    * A task state segment (TSS)

Paging in hardware

- Paging unit translates linear addresses into physical addresses
- Checks the requested access type against the access rights of linear address
- Invalid memory access leads to page fault exception
- Page: Linear addresses are grouped in fixed-length intervals called pages
- Contiguous linear addresses within a page are mapped into contiguous physical addresses
- Paging in x86 is enabled by setting the PG flag of a control register named cr0; if PG is cleared, linear addresses are interpreted as physical addresses
- Regular paging
  - Page size standardized to 4KB
  - 32-bits of a linear address are divided into
    1. Directory – Most significant 10 bits
    2. Table – Intermediate 10 bits
    3. Offset – Least significant 12 bits
  - Translation from linear to physical address involves two steps, based on a type of translation table
    1. Page directory
    2. Page table
      * Reduces the amount of RAM required for per-process page table
      * Simple one-level page table will require up to $2^{20}$ entries; 32-bit addresses will yield 4MB of RAM for the page table for each process
      * Two-level scheme requires page tables only for the virtual memory regions actually used by process
      * 1024 entries in each of page directory and page table
      * Same structure for the entries in page directory and page table
Present flag  Valid/invalid page

Field containing 20 most significant bits of physical address  Each page frame has 4KB capacity; the 12 least significant bits of the physical address are always zero; if the field refers to page directory, page frame contains a page table; if it refers to a page table, page frame contains a page of data

Accessed flag  Same as refer bit; paging unit never resets this flag, always done by OS

Dirty flag  Applies to page table entries only; set each time a page is written into; paging unit never resets this flag, always done by OS

Read/Write flag  Access rights for the page or page table

User/supervisor flag  Privilege level required to access the page

PCD and PWT flags  Controls the way the page or page table is handled by hardware cache

Global flag  Applies to page directory entries; if set, the entry refers to a 2MB or 4MB long page frame

Extended paging

– Introduced in Pentium
– Allows page frames to be 4MB instead of 4KB
  * Enabled by setting the Page Size flag of a page directory entry
– Used to translate large contiguous linear address ranges into physical address
– Kernel does not need intermediate page tables and saves memory and preserves TLB entries

Physical Address Extension paging mechanism

– Amount of RAM supported by a processor limited by the number of address pins connected to the bus
  – Intel CPUs, up to Pentium, used 32-bit physical address
– Allowed for up to 4GB of RAM but linear space requirements of use mode processes limited the kernel to 1GB of RAM
– Larger servers required more RAM and Intel increased the number of address pins from 32 to 36, starting with Pentium Pro
  * Increased the addressable RAM to 64GB
  * Required new paging mechanism to translate a 32-bit linear address into a 36-bit physical address
  * Leads to Physical Address Extension

• Activated by setting the PAE flag in cr4 control register
  – PS (page size) flag in page directory enables large page size (2MB when PAE is enabled)

• New paging mechanism
  – 64GB of RAM split into $2^{24}$ page frames
  – Physical address field of page table expanded from 20 to 24 bits
  – Page table entry of 36 bits: 12 flag bits + 24 physical address bits
    * New page table entry size doubled to 64 bits
    * New 4KB PAE page table includes 512 entries instead of 1024
  – New level of page table called page directory pointer table (PDPT) with four 64-bit entries
    * PDPTs stored in the first 4GB of RAM and aligned to multiple of 32 bytes
    * Base address of PDPT can be represented in 27 bits in cr3 control register
  – PAE does not enlarge the user space of a process; a process in user mode can only address up to 4GB
  – PAE allows the kernel to exploit up to 64GB of RAM, and hence increases the number of processes in the system

• Hardware cache
  – CPU clock rates of GHz vs DRAM access time in hundreds of cycles
  – CPU may be held back while executing instructions involving memory reference
  – Hardware cache reduces speed mismatch between CPU and RAM
  – Based on principle of locality, for both code and data
  – New unit called line
    * A few dozen contiguous bytes that are transferred in burst mode between slow DRAM and fast on-chip static RAM used to implement caches
    * Cache divided into subsets of lines
      · Direct-mapped cache: a line in main memory always stored at the exact same location in cache
      · Fully associative cache: Any line in memory can be stored at any location in cache
      · N-way set associative cache: Any location can be stored in any one of N lines of cache
  – Cache unit inserted between paging unit and main memory
* Includes both a hardware cache memory and a cache controller
* Cache memory stores the actual lines of memory
* Cache controller stores an array of entries, one entry for each line of cache memory
  - Entry includes a tag and a few flags to describe the status of line
  - Tag has bits to allow the cache controller to recognize the memory location mapped by line
  - Bits of physical address split into three groups: tag, cache controller subset index, and offset within the line
  - If a line with the same tag as high-order bits of address is found, CPU has a cache hit; otherwise, it has a cache miss
* Cache hit
  - Cache controller behaves differently for different access
  - For read, controller simply transfers data from cache line to CPU register (no RAM is accessed)
  - For write, there are two basic strategies
    - Write-through – Controller writes into both RAM and cache, effectively switching off cache for write
    - Write-back – Cache line is updated and RAM is left unchanged; RAM is updated only when CPU executes an instruction requiring a flush of cache or when a FLUSH hardware signal occurs
* Cache miss
  - Cache line is written to memory and correct line fetched from RAM
* Cache snooping
  - Separate hardware cache for every core in multiprocessor systems
  - When a CPU modifies its hardware cache, it checks if the same data is contained in other CPUs’ cache; if so, it must notify the other CPU to update its cache value
  - Done at the hardware level and of no concern to kernel

### Page frame management

- Intel Pentium can use two different page sizes: 4KB and 4MB
- Linux adopts the smaller 4KB page frame size as standard memory allocation unit
* Page fault exceptions issued by paging circuitry are easily interpreted; if page does not exist, memory allocator must find a free 4KB frame and assign it to the process
* Transfer of data is more efficient when the smaller size is used

### Physical memory layout in Linux

- Kernel builds a physical addresses map during initialization
  - Address ranges usable by kernel
  - Unavailable address ranges due to hardware devices, I/O shared memory, or the page frames containing BIOS data
- Reserved page frames
  - Unavailable physical address ranges
  - Ones containing kernel code and initialized data structures
  - Reserved page frames cannot be dynamically assigned or swapped to disk
- Kernel installed in RAM starting at address 0X00100000 (second MB)
- Typical kernel configuration loaded in less than 3MB of RAM
- What about the first MB in RAM?
  - Page frame 0 used by BIOS to store system hardware configuration detected during the *Power-On Self Test*
Memory Management

- Physical addresses in the range [0X000A0000, 0X000FFFFF] reserved for BIOS routines and to map internal memory of ISA graphics cards
  * Contains the area from 640KB to 1MB that is reserved in PCs

- Boot sequence
  - Kernel queries BIOS to find physical memory size
  - Kernel executes `machine_specific_memory_setup()` function to build physical addresses map
  - All page frames with numbers from 0X9F (`LOWMEMSIZE()`) to 0X100 (`HIGH_MEMORY()`) are marked as reserved

- Address spaces
  - System bus connects CPU to main memory
    - CPU connected to hardware peripherals through other buses
    - Memory space corresponding to hardware peripherals is called I/O space
  - CPU can access both system space memory and I/O space memory
  - I/O controllers can access system memory indirectly with some help from CPU
  - A CPU may have separate instructions to access memory and I/O space
    * Read a byte from I/O address 0x3F0 into register X
  - CPU controls system hardware peripherals by reading and writing into their registers in I/O space

- Non-Uniform Memory Access (NUMA)
  - Supported in Linux after 2.6 kernel
  - Access times for different memory locations from a given CPU may vary
  - Physical memory is partitioned into several nodes
  - Time to access pages within a single node is the same but may be different for two different CPUs
  - For every CPU, kernel tries to minimize the number of accesses to costly nodes by selecting the node to store most frequently used memory references
  - Physical memory inside each node is split up into several zones

Buddy system algorithm

- Used to resolve external fragmentation (used in Linux)
- Group free frames into 11 lists of blocks that contain groups of $2^0 - 2^{10}$ contiguous frames, respectively
- Largest request of 1024 frames corresponds to a chunk of 4MB of contiguous RAM
- Physical address of the first frame in a multiple of group size
  - Address of a 16-frame block is a multiple of $16 \times 2^{12}$
  - $2^{12}$ is the regular page size
- Example: Request for 1MB segment of memory
  - 1MB == 256 frames
  - Check if there is a free block in the 256-frame list; yes? we are done
  - No? Look for next larger block (512-frames)
  - If a 512-frame block is found, kernel allocates 256 of the 512 frames to the request and inserts the remaining 256 into the list of 256-frame blocks
Memory Management

– If not, look for the next larger block (1024-frames)
– If a 1024-frame block is found, kernel allocates 256 frames to the request, inserts the last 512 frames into the list of 512-frames and inserts the remaining 256 frames into the list of 256-frame blocks
– If not, the algorithm gives up and signals an error

• Releasing blocks of page frames
  – Kernel attempts to merge pairs of free buddy blocks of size $b$ into a single block of size $2b$
  – Blocks are considered buddies if
    * Both blocks have the same size, say $b$
    * They are located in contiguous physical addresses
    * The physical address of the first page frame of the first block is a multiple of $2 \times b \times 2^{12}$
  – If algorithm succeeds in merging released blocks, it doubles $b$ and tries again to create even bigger blocks

• Memory allocation in Linux
  – Kernel-mode memory allocation
    * Kernel is highest-priority component of OS
    * A request for dynamic memory by kernel is only made when needed and should not be deferred
    * Kernel functions are assumed to be error-free; kernel does not need to insert any protection against programming errors
  – User-mode memory allocation
    * Request for dynamic memory is considered non-urgent
    * A process is unlikely to address all the pages of code when an executable binary is loaded
    * A request by process for dynamic memory (malloc) does not mean that the process needs all the memory right away
    * As a rule, kernel tries to defer allocating dynamic memory to user-mode processes
    * User programs cannot be trusted; kernel must be prepared to catch all addressing errors caused by processes in user mode
    * When a user process asks for memory, it does not get additional page frames; instead it gets the right to use a new range of linear addresses which become part of its address space; this interval is called a memory region

• Process address space
  – All linear addresses that the process is allowed to use
  – Different set of linear addresses for each process
  – Kernel may dynamically modify a process address space by adding or removing intervals of linear addresses
  – Intervals of linear addresses specified by memory regions
    * Initial linear address, aligned to 4K
    * Length of memory region, multiples of 4K
    * Access rights
  – Memory regions are released upon exec and a new set of regions assigned to the process

• Creating a process address space
  – Kernel invokes a function that sets up all page tables and memory descriptors of the new process
  – Traditional processes inherit the address space of the parent process
    * Pages stay shared as long as they are only read (COW approach)
  – Lightweight processes use the address space of their parent process
    * Address space is never duplicated
Memory Management

- Created considerably faster than the normal processes
- Parent and child must coordinate their accesses carefully

- Kernel thread processes
  - Have no virtual memory
  - Run in kernel mode in physical address space

- Swapping out and discarding pages in Linux
  - Performed by kernel swap daemon (kswapd)
    - Runs on kernel thread
    - Makes sure that there are enough free pages in the system for efficient memory management
    - Started by kernel init process and waits for the kernel swap timer to expire periodically
  - kswapd uses two variables to manage memory:
    - free_pages_high: If the number of free pages is more than this variable, kswapd does nothing
      - Tracks the number of pages being written to the swap area in the variable nr_async_pages
      - nr_async_pages incremented when a page is queued waiting to be written into swap; decremented when the write to swap device is completed
    - free_pages_low
      - When the number of pages falls below either of the free_pages variables, kswapd attempts to reduce the number of physical pages being used by
        1. Reducing the size of buffer and page caches
        2. Swapping out System V shared memory pages
        3. Swapping out and discarding pages
      - If number of free pages is less than free_pages_low, kswapd tries to free 6 pages, otherwise, it tries to free 3 pages
        kswapd remembers the method it used to free physical pages and on subsequent runs, starts trying to free pages using last successful method
      - kswapd looks at each process in the system to see if it is a good candidate for swapping
        After locating a victim, it goes through all of its virtual memory regions to check areas that are not shared or locked

- Swap algorithm in Linux
  - Based on page aging
  - Each page has a counter
  - Pages age when they are unused and rejuvenate on access; kswapd swaps out only the old pages
  - Upon allocation, a page gets an initial age of 3
  - Upon access, the age is increased by 3 to a maximum of 20
  - Every time kswapd runs, pages are aged by decrementing 1 from the age
  - If the page is old (age = 0), kswapd will process it further
  - If enough of the swappable process’ pages have been swapped out, kswapd will sleep again
  - The next time, kswapd wakes up, it will consider the next process in the system
  - This allows kswapd to nibble away at each process’ physical pages until the system is again in balance