Previous class...

- **Uniprocessor policies**
  - FCFS, Shortest Job First
  - Round Robin
  - Multilevel Feedback Queue

- **Multiprocessor policies**
  - A MFQ per processor
  - A process has affinity with a processor
  - Exception: idle cores can steal processes
Background

- Main memory and registers are the only storage that a CPU can access directly
- Program must be brought (from disk) into memory for it to be run
Outline

• Fixed partitions
• Dynamic partitions
• Buddy system

• Segmentation
• Paging

Contiguous allocation:
Each process occupies a contiguous memory region in the physical memory

Non-contiguous allocation:
Each process occupies multiple memory regions scattered in the physical memory.
Fixed partitions

• The bounds of each partition are *fixed/predefined*

• Disadvantages:
  – Cause Internal Fragmentation when the allocated space is larger than the need
    • What is “*fragmentation*”? Small useless chunks
    • E.g., when you put a 13M process in the 16M partition, 3M space is wasted and it is called *internal fragmentation*
      • *Internal*: the wasted space is inside allocated space
  – The number of active processes is limited

• Analogy: street parking with meters
Fixed partitions - questions

• If 8M partitions are all used, where do you place a 7M process, and what is the size of the internal fragmentation?
  – The 12M partition is the best choice
  – The internal fragmentation is 5M

• How to resolve the severe internal fragmentation?
  – Dynamic partitions
Dynamic partitions

• Process is allocated exactly the memory it requires
• The partitions are dynamic: the number and locations of partitions are not fixed
• Analogy: street parking without meters
Dynamic partitions - example
Dynamic partition - disadvantage

• **External fragmentation**
  – Their total size is large enough to satisfy a request, but they are not contiguous, so cannot be used to service the request
  – *External*: fragmentation is outside allocated space

• **Solution: Compaction**
  – OS shifts processes so that they are contiguous; thus, free memory is together in one block
  – Program execution must be paused for relocation; waste CPU time
Placement Algorithms

• When there is more than one free block of memory of sufficient size, the system must decide which free block to allocate.

<table>
<thead>
<tr>
<th>Best-fit</th>
<th>First-fit</th>
<th>Next-fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>• chooses the block that is closest in size to the request</td>
<td>• scan memory from the beginning and chooses the first available block that is large enough</td>
<td>• scan memory from the location of the last placement and chooses the next available block that is large enough</td>
</tr>
</tbody>
</table>
Assume a 16M partition is requested
Questions

• Does Fixed Partitions Allocation has external fragmentation?
  – Zero
  – There are no small useless chunks outside a partition

• Does Dynamic Partitions Allocation has internal fragmentation?
  – Zero
  – The allocated partition size is as needed
Buddy System (or, Buddy Memory Management)

- **Fixed partitions** cause severe internal but zero external fragmentation, while **dynamic partitions** cause severe external but zero internal fragmentation.
- Buddy System is between the two
  - It causes acceptable internal and external fragmentation, and has good overall performance.
- Three properties
  - Split-based allocation
  - **Freelists**-based implementation
  - Coalescing-buddy-based deallocation
Buddy System – allocate the first block

• To begin, the entire space available for allocation is treated as a single block of size $2^U$.

• The request size is first rounded up as power of 2, denoted as $S$
  – E.g., 55 -> $2^6=64$; 120 -> $2^7=128$

• If $S = 2^U$, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size $2^{U-1}$.

• If $S = 2^{U-1}$, then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split into halves again. This process continues until the split block is equal to $S$ and it is allocated to the request.
### Buddy System – split-based allocation

<table>
<thead>
<tr>
<th>Request Size</th>
<th>Allocation</th>
<th>1 Mbyte block</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 K</td>
<td>A = 128K</td>
<td>128K</td>
</tr>
<tr>
<td></td>
<td>128K</td>
<td>256K</td>
</tr>
<tr>
<td></td>
<td>512K</td>
<td></td>
</tr>
<tr>
<td>240 K</td>
<td>A = 128K</td>
<td>128K</td>
</tr>
<tr>
<td></td>
<td>B = 256K</td>
<td>512K</td>
</tr>
<tr>
<td>64 K</td>
<td>A = 128K</td>
<td>64K</td>
</tr>
<tr>
<td></td>
<td>B = 256K</td>
<td>512K</td>
</tr>
<tr>
<td>256 K</td>
<td>A = 128K</td>
<td>64K</td>
</tr>
<tr>
<td></td>
<td>B = 256K</td>
<td>D = 256K</td>
</tr>
<tr>
<td></td>
<td>256K</td>
<td></td>
</tr>
<tr>
<td>Release B</td>
<td>A = 128K</td>
<td>64K</td>
</tr>
<tr>
<td></td>
<td>D = 256K</td>
<td>256K</td>
</tr>
<tr>
<td>Release A</td>
<td>128K</td>
<td>64K</td>
</tr>
<tr>
<td></td>
<td>D = 256K</td>
<td>256K</td>
</tr>
<tr>
<td>75 K</td>
<td>E = 128K</td>
<td>64K</td>
</tr>
<tr>
<td></td>
<td>D = 256K</td>
<td>256K</td>
</tr>
<tr>
<td>Release C</td>
<td>E = 128K</td>
<td>128K</td>
</tr>
<tr>
<td></td>
<td>D = 256K</td>
<td>256K</td>
</tr>
<tr>
<td>Release D</td>
<td>512K</td>
<td>D = 256K</td>
</tr>
<tr>
<td></td>
<td>256K</td>
<td></td>
</tr>
<tr>
<td>Release E</td>
<td></td>
<td>1 M</td>
</tr>
</tbody>
</table>
A Binary Tree Representation

• **Buddies**: sibling nodes, i.e., two nodes that share the parent node in the binary tree representation
Buddy System – free-lists-based implementation

A free list is a linked list data structure used to connect all unallocated storage chunks. It is quick and easy to allocate and unallocate a storage chunk.

<table>
<thead>
<tr>
<th>Order</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 2^0 x 64k = 64k</td>
<td></td>
</tr>
<tr>
<td>1: 2^1 x 64k = 128k</td>
<td></td>
</tr>
<tr>
<td>2: 2^2 x 64k = 256k</td>
<td></td>
</tr>
<tr>
<td>3: 2^3 x 64k = 512k</td>
<td></td>
</tr>
<tr>
<td>4: 2^4 x 64k = 1024k</td>
<td></td>
</tr>
</tbody>
</table>

One 2^2 block is allocated.
Buddy System used in Linux

- The Buddy System is used in Linux (and many other OSes) to manage the physical memory.
Buddy System – coalescing-buddy-based deallocation

- Whenever a block is freed, it tries to coalesce with its buddy. The coalescing procedure is recursive
Buddy System - fragmentation

• Internal fragmentation?
  – Worst case ~ 50%
  – E.g., request 128.01K, which is rounded up to 128x2K

• External fragmentation
  – Still exists, as coalescing can only occur between buddies
  – Best-fit is always used: freelists-based implementation allows you to find the best-fit chunk *quickly*
  – By applying restriction on splitting, you will not see small mini tiny useless holes; e.g., previously we limit the smallest block as 64k
Working Set

• **Working Set**: the amount of memory that a process references in a given time interval
  – You can roughly understand it as “the memory regions that are currently used”

• Usually a very small portion of the entire memory is requested by a process

• **Analogy**: the seat you currently use is your “working set”, while during your study at Temple you need much more space: library, dining, lab, classroom seat, etc.
Limitations of contiguous allocation

• Does contiguous allocation exploit the small working set?
  – No, as a contiguous memory block is allocated to meet the maximum need of a process; it means that a process cannot run unless its maximum need is met
  – But actually only a small portion of its maximum need is really accessed in a given time interval
Working Set - example

• With contiguous allocation, N processes reside in memory. Assume the memory requested by each process is equal and the working set is \( \frac{1}{4} \) of the requested memory.

• At most, how many active processes can be serviced by the main memory in theory if only the working set of each process resides in memory?
  – 4N
Big picture

- Fixed partitions
- Dynamic partitions
- Buddy system
- Segmentation
- Paging

Contiguous allocation:
Each process occupies a contiguous memory region in the physical memory.

Non-contiguous allocation:
Each process comprises multiple memory regions scattered in the physical memory.
Swapping

• When the free memory runs low, swap area in the disk is used
• All or part of a process’s data is swapped temporarily out of memory to the swap area, and then brought back into memory for continued execution
• Swapping is found on many systems (i.e., UNIX, Linux, and Windows)
Segmentation

- A program is divided into segments
- A segment can be a procedure, a stack, a global array, etc.

- Only segments that correspond to the current working set reside in the main memory.
- Others are put at disk, and can be swapped into main memory when needed
Disadvantage of Segmentation

• When a process exits, its segments leave “holes” of varying sizes in main memory
  – Similar to dynamic partitions, external fragmentation is still severe

• Inefficient handling of growing segments, such as heap and stack
  – Reserving a large memory space leads to severe internal fragmentation, while reserving a small space will result in repetitive reallocation when it grows
Paging

• Main memory is divided into equal fixed-size chunks, called *page frames*, that are relatively small
• A process is divided into small fixed-size chunks, called *pages*, of the same size
  – A Page refers to a chunk of address space
• The pages of a process can be stored in separated page frames in main memory
• *Any page can be put at any page frame*
Logical view – virtual address space

• The logical view: each process has a huge contiguous virtual address space
  – 32-bit system: $2^{32}$
  – 64-bit system: $2^{64}$ (so far, only $2^{48}$ is used)

• This largely simplifies the compiler, which assumes a uniform huge address space, regardless of the allocation in physical memory
Anatomy of the virtual address space of a process

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)
- Random stack offset
- RLIMIT_STACK (e.g., 8MB)
- Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
Paging Example

Internal fragmentation?
- Yes, but it only occurs for the last page when the requested size is not a multiple of pages. E.g., a process that requests 3.1 pages of space will get 4 pages.

External fragmentation?
- No, any “holes”, i.e., page frames, left by the exited process can be reused happily.
Understanding the output of `pmap`

Please try “pmap –XX [pid]” after class

The output of `pmap` provides detailed information about the memory mapping of a process. Here are some key columns in the output:

- **Address**: The starting address of the memory mapping.
- **Perm**: Permissions associated with the mapping (r - read, w - write).
- **Offset**: The offset within the file where the mapping begins.
- **Device**: The device number for the memory mapping.
- **Inode**: The inode number of the file.
- **Sec**: The number of sections in the memory mapping.
- **Rsv**: Reserved memory.
- **File**: The file name associated with the memory mapping.
- **Shared**: Shared memory.
- **Private**: Private memory.
- **Dirty**: Dirty memory.
- **Anonymous**: Anonymous memory.
- **Anonludgepages**: Anon ludge pages.
- **Swap**: Swap memory.
- **Pages**: Pages in the mapping.
- **Size**: Size of the mapping.
- **Locked**: Locked memory.
- **Pagesize**: Page size.
- **Vmages**: Virtual memory sizes.
- **Mapping**: Mapping flags.

The source of the mapping — either a file, an anonymous mapping, or a special mapping such as `[heap]`, `[stack]`, `[vmap]`, `[vsyscall]` — is indicated at the bottom of the output.

The memory layout includes:

- **Device major\minor number**: The device number for the mapping.
- **File inode number**: The inode number of the file.
- **Resident Set Size**: The current resident set size in KiB.
- **Proportional Share Size**: The proportional share size of the mapping.
- **Amount of memory which is shared**: The amount of memory shared.
- **Amount of memory which is private**: The amount of memory private to the process.
- **Amount of memory which is swapped**: The amount of memory swapped out.
- **Amount of anonymous memory**: The amount of anonymous memory.
- **Swapped**: Swapped memory.
- **Pagesize**: Page size. Page sizes in (KiB) used in this mapping.

For more information, please refer to the manual page of `pmap` for detailed documentation.
Questions

• Where do global and static variables reside during execution?
  – It depends
  – **BSS** if you declare them w/o init values
    • System zeros page content automatically, so you can expect they are all zeros
  – **Data** otherwise

• Why cannot I declare a big array (>8M) in my function?
  – Stack limit
  – You should use malloc to allocate from the heap

• Why do I encounter segmentation fault exceptions?
  – Many possibilities
  – Refer to some unallocated holes
  – Refer to some heap buffers that have been freed
  – Access protected areas, such as kernel space; write to read-only
  – How to debug: Use “bt” (backtrace) of gdb to debug
Summary

• Fixed partitions
• Dynamic partitions
• Buddy system
  – Split-based allocation
  – Coalescing-buddy-based deallocation
  – Freelists-based implementation

• Segmentation
• Paging

• Something you can use in your future design and coding
  – Free list
Writing Assignment

• What is internal fragmentation? What is external fragmentation?
• What is the worst case of internal fragmentation for a memory allocation in the Buddy System?
• If the swapping mechanism is not used, do the schemes of segmentation/paging still have advantages over contiguous memory allocation?
• What will happen if the RAM is less than the total size of the working sets of the processes?