DCS216 Operating Systems

Lecture 22
Memory (5)

May 27th, 2024

Instructor: Xiaoxi Zhang

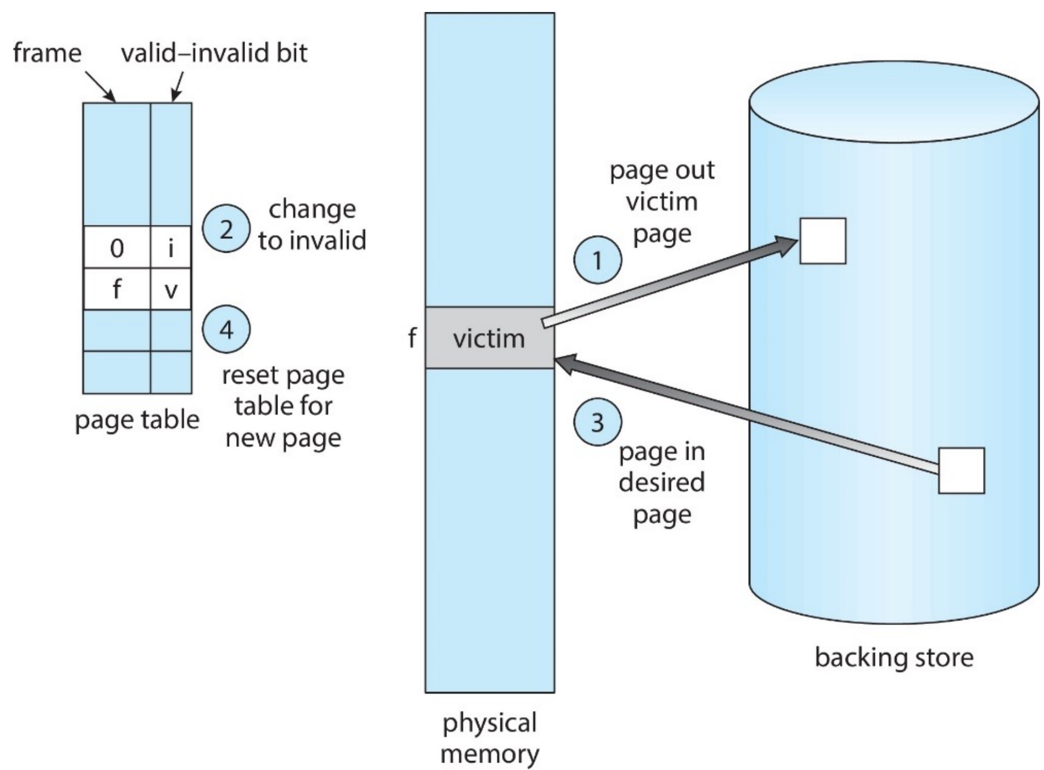
Sun Yat-sen University

Content

- Background
- Demand Paging
- Copy-on-Write (**COW**)
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- OS Examples
  - Linux
  - Windows
  - Solaris

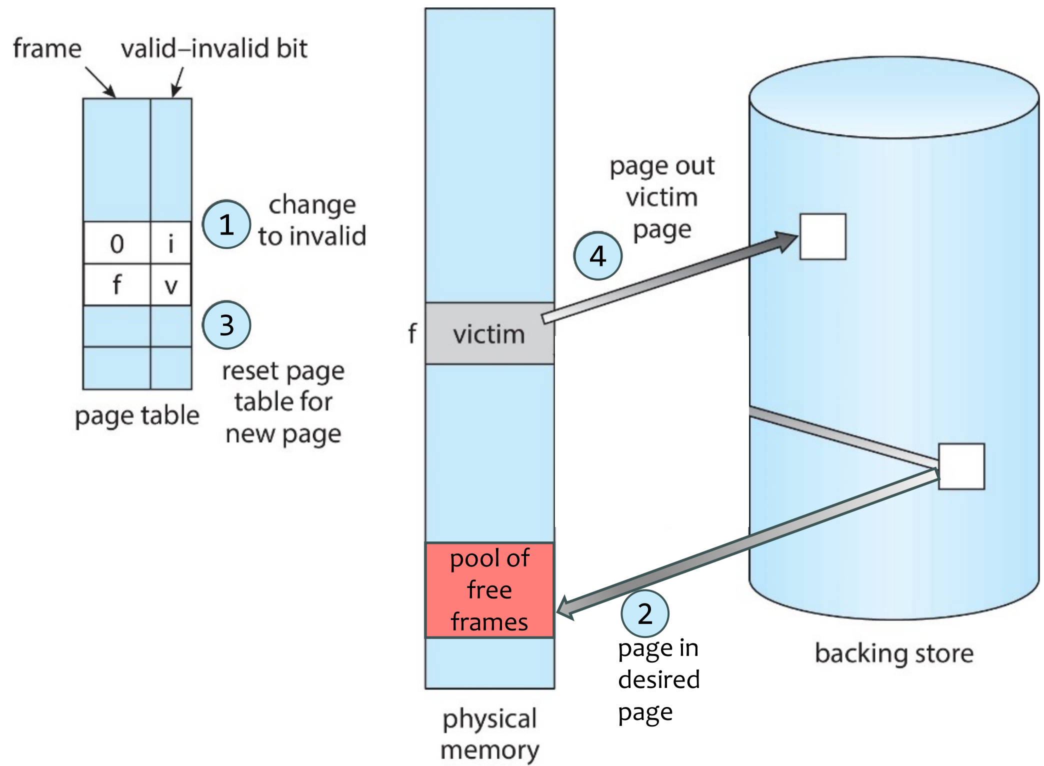
Page Fault

- Recall that when Page Fault occurs and there is no free frame, a **Page Replacement Algorithm** (e.g., LRU) normally needs to perform two **I/O** operations: page-out victim page and page-in desired page.
  - With "**Modified Bit**" set in PTE, page-out is mandatory.
  - How can we minimize the time for page-out and page-in?



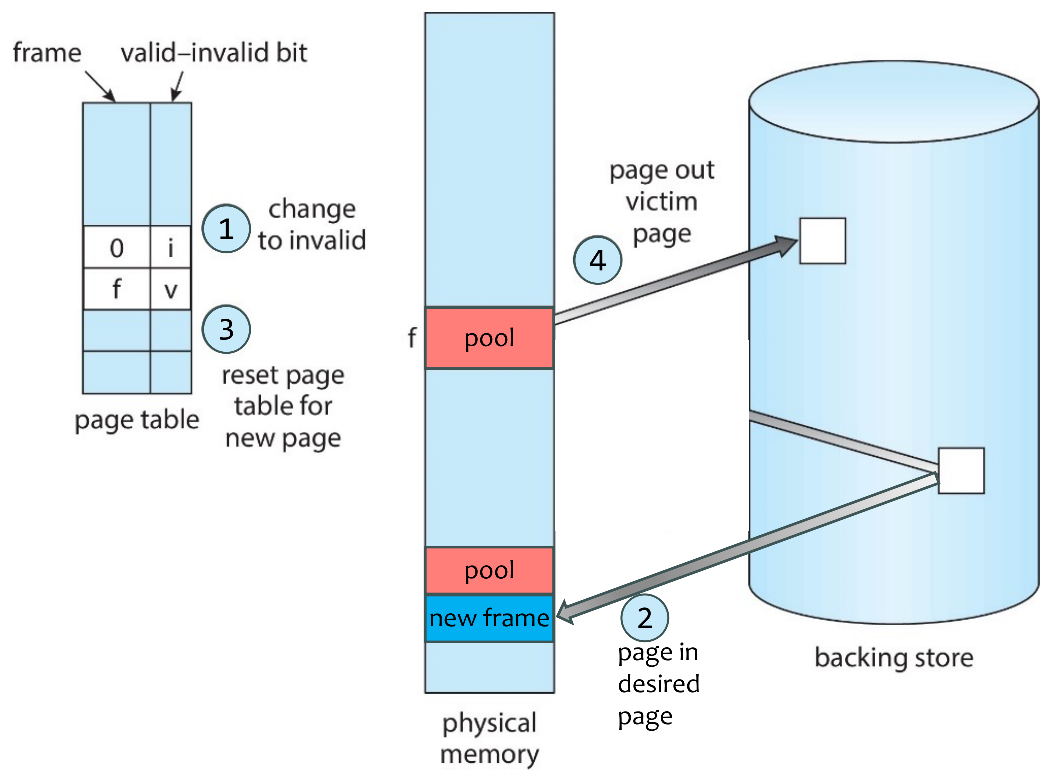
Page-Buffering Algorithms

- How can we minimize the time for page-out and page-in?
  - We maintain a pool of free frames.
  - When a Page Fault occurs, a victim frame is chosen. The desired page is loaded into **a frame** from the pool of free frames **before** the victim page is paged out.



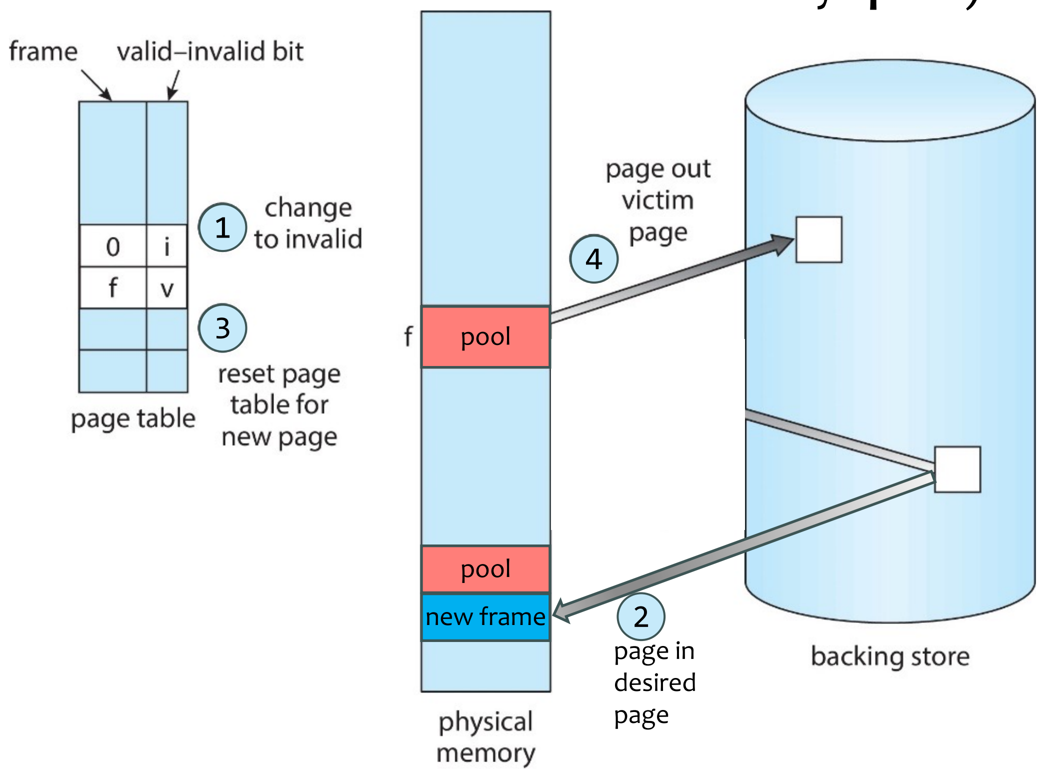
Page-Buffering Algorithms

- How can we minimize the time for page-out and page-in?
  - Once the victim page is paged out (and written to disk), the **victim frame** will be **added** to the pool of free frames.
  - This allows the process to **restart** as soon as possible, without **waiting** for the victim page to be written out.



Page-Buffering Algorithms

- How can we minimize the time for page-out and page-in?
  - The downside?
    - We have to allocate an **extra** pool of free frames that **should not** be actively **used**, but rather as a buffer for victim frames.
    - Another example of **space-time** trade-off, (i.e., optimize paging **time** at the cost of extra unused memory **space**).



Demand Paging

- **Two major problems to implement Demand Paging:**
  - Frame Allocation Algorithm
    - How many frames to allocate to each process?
    - Which frames to replace?
  - Page Replacement Algorithm
    - Which frame(s) to choose as the victim frame(s)?
    - **Goal**: To achieve the lowest Page Fault Rate
  - Evaluation (by simulation)
    - Running it on a particular string of memory references (reference string) and computing the number of page faults on that string
      - String is just page numbers, not full addresses
      - Repeated access to the same page does not cause a page fault
      - Results depend on the number of frames available
    - Our example reference string of referred page numbers is

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```

Demand Paging

- **Two major problems to implement Demand Paging:**
  - Frame Allocation Algorithm
    - How many frames to allocate to each process?
    - Which frames to replace?
  - Page Replacement Algorithm
    - Which frame(s) to choose as the victim frame(s)?
    - **Goal:** To achieve the lowest Page Fault Rate
  - Evaluation (by simulation)
    - Running it on a particular string of memory references (reference string) and computing the number of page faults on that string
      - String is just page numbers, not full addresses
      - Repeated access to the same page does not cause a page fault
      - Results depend on the number of frames available
    - Our example reference string of referred page numbers is

`7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1`

Allocation of Frames

- How do we allocate to various processes the fixed amount of free memory that is available?
- For example, consider a system with **512KB** of physical memory and the Page Size is **4KB**
  - Number of frames: 128
  - Suppose the OS occupies 140KB $\Rightarrow$ 35 Frames
  - Frames remaining for user processes = 128 – 35 = 93
    - These 93 frames would be available in the free frames list.
- Pure Demand Paging:
  - When a user process started execution, the first 93 **Page Faults** would get all free frames from the list.
  - When the free frames are exhausted, a **Page Replacement Algorithm** would be used to **select** one of the 93 in-memory pages to be **replaced** with the 94^{th} page, and so on.
  - When the process terminates, the 93 frames would once again be placed on the free-frames list.

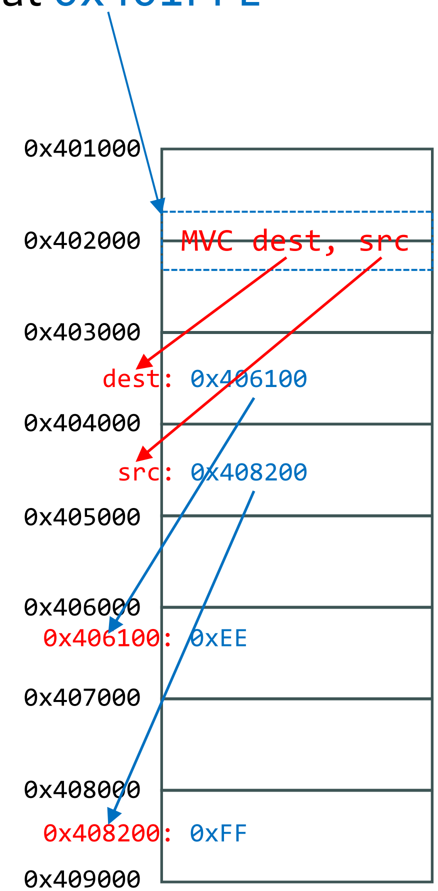
Allocation of Frames

- Constraints for Allocation of Frames:
  - **Maximum:** We cannot allocate more than the total number of available frames (unless there is page sharing)
  - **Minimum:** A minimum number of frames should be allocated to each process.
- There should be enough frames to hold all the different pages that any single instruction can **reference**.
  - **Minimum** number of frames per process is **defined** by the **architecture**.
  - Suppose in a one-level **indirect** addressing, a `load` instruction (on page 15) refers to an address on page 3, which is an indirect reference to page 22.
    - A minimum of 3 frames are needed.
  - In IBM 370, the **SS MOVE** (**S**torage-to-**S**torage **MOVE**) instruction:
    - `MVC dest, src`

Allocation of Frames

- In IBM 370, the **SS MOVE** (**S**torage-to-**S**torage **MOVE**) instruction:
  - `MVC dest, src`
  - Instruction itself is 6 bytes, which might span **2** pages
    - E.g., the `` `MVC dest, src` `` instruction is located at `0x401FFE`
  - **2** pages to handle `dest` (`0x403100`)
    - 1 page to reference the addr of label `dest`
    - 1 page to reference the addr `dest` points to
  - **2** pages to handle `src` (`0x404500`)
    - 1 page to reference the addr of label `src`
    - 1 page to reference the addr `src` points to
  - In total: a minimum of **6 frames** needed

    in the worst case.



Allocation of Frames

- In x86, however, **direct** memory-to-memory movement is not allowed.
  - `` `mov (%esi, %ebx, 4), (%edx)` `` is illegal.
- The most complex instruction (in terms of # of memory accesses)
  - E.g., `` `mov (%esi, %ebx, 4), %edx` `` *(located at `0x401FFE`)*.
  - requires a minimum of **4** **frames**.
    - **2** pages for instruction (that **spans** two pages)
    - **2** pages for indirect memory reference of one of the operands.

Allocation of Frames

- **Allocation Algorithms**
  - Frame Allocation Algorithms would help us in determining how many frames should be allocated to different processes in a multiprogramming environment.
  - 3 most common allocation algorithms:
    - **Equal** Allocation
    - **Proportional** Allocation
    - **Priority** Allocation

Allocation of Frames

- **Equal Allocation**
  - The frames are equally distributed among the processes.
  - If we have m frames and n processes
    - allocate m/n frames to each process. (ignoring the OS for the moment)
  - For example:
    - Number of frames = 93
    - Number of processes = 5
    - Each process gets $\lfloor 93/5 \rfloor = 18$ frames
      - the remaining 3 frames can be kept as **buffer pool of free frames.**
  - The Problem: **memory requirement of processes are not the same.**
  - For example, in a system with 64 frames with page size of 1KB. For only 2 processes: **P1(10KB)** and **P2(127KB)**, if we adopt equal allocation, then each process gets allocated 32 frames.
    - For **P1(10KB)**, 22 frames were wasted.
    - For **P2(127KB)**: underallocated.

Allocation of Frames

Proportional Allocation

- The frames are distributed among processes according to their sizes.
- Let:
  - m be the number of frames
  - s_i be the size of process p_i
  - $S = \sum s_i$
- then Number of frames allocated to p_i:
  - $a_i = \frac{s_i}{S} \times m$
- For example, in a system with 64 frames with page size of 1KB. For only 2 processes: **P1(10KB**) and **P2(127KB**), if we adopt proportional allocation, then
  - Frames allocated to p_1: $a_1 = \frac{s_1}{S} \times m = \frac{10}{137} \times 64 \approx 5$
  - Frames allocated to p_2: $a_2 = \frac{s_2}{S} \times m = \frac{127}{137} \times 64 \approx 59$

Allocation of Frames

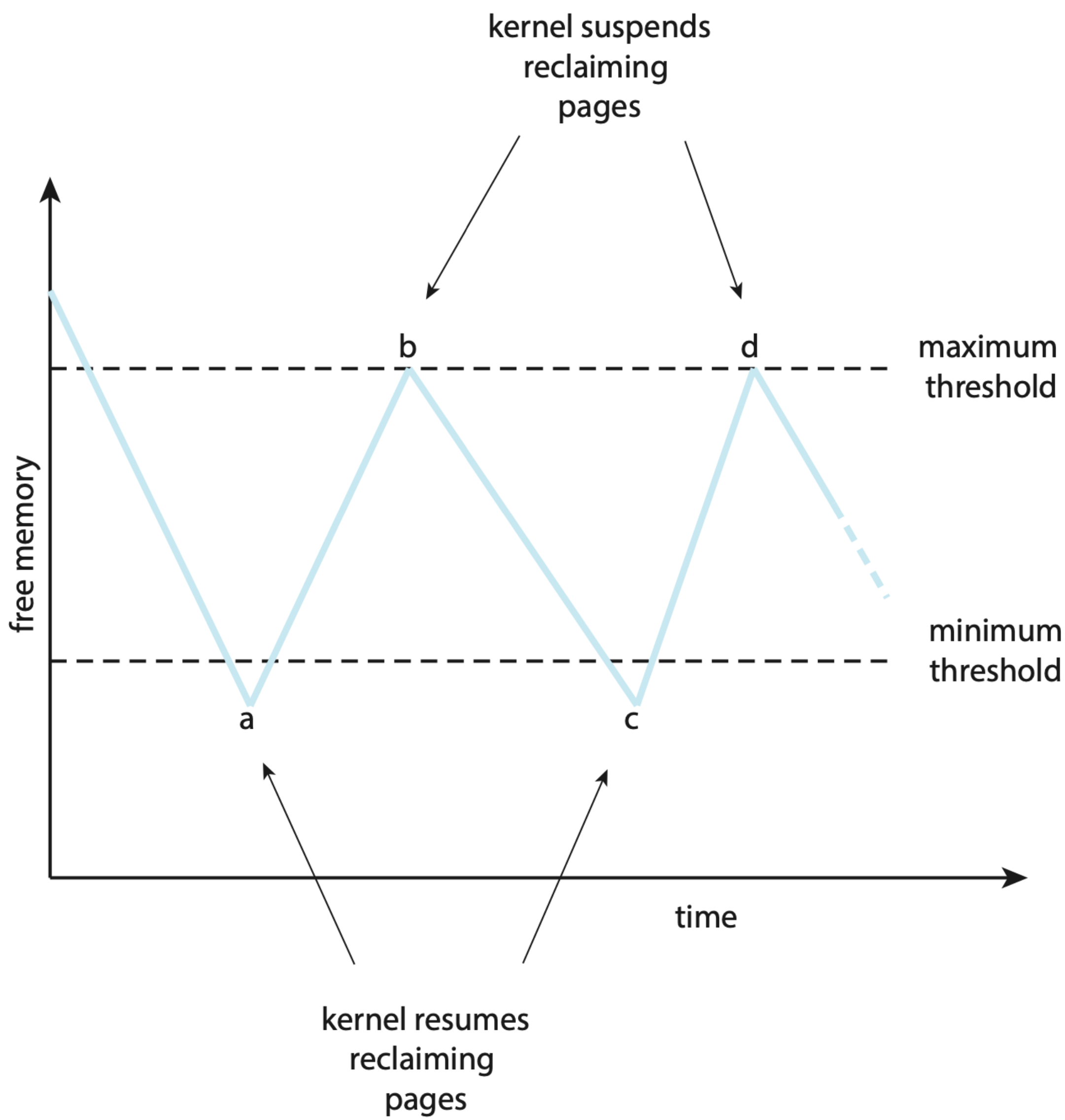
Priority Allocation

- The frames are distributed among processes according to their **priorities.**
  - Processes with higher priorities are allocated more frames so as to **speed up** their execution.
  - $a_i = \frac{prio_i}{\sum prio_i} \times m$
- A combination of size **and** priority can be used rather than just the process size **or** priority.

Allocation of Frames

- **Global Allocation vs. Local Allocation**
  - Page Replacement Algorithms can also be classified into two broad categories based on the way frames are allocated to different processes:
    - **Global** Allocation
      - Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
    - **Local** Allocation
      - Local replacement requires that each process select from **only** its own set of allocated frames.

Global Allocation



Thrashing

- Consider a process that doesn't have enough frames for execution.

It will quickly cause a Page Fault

Replace a page with the new desired page

The page that was replaced was an actively used page $\Rightarrow$ Page Fault

Replace a page with the new desired page

The page that was replaced was an actively used page $\Rightarrow$ Page Fault

This cycle continues leading to a high paging activity called **Thrashing**.

If a process is spending more time in paging rather than in executing, then we say that process is **Thrashing (系统抖动)**.

- **Severe Performance Issue due to Thrashing**
  - The OS monitors CPU utilization. If CPU utilization is too low, we generally increase the degree of multiprogramming by introducing a new process into the system.
  - As processes are busy swapping pages in and out, they queue up for the paging device and the ready queue (processes) becomes empty.
  - Since processes are now waiting for the paging device, the CPU utilization decreases.
  - The CPU scheduler observes the decreased utilization and therefore attempts to increase it by introducing new processes in the system.
  - These new processes tries to take frames from the older processes, hence increasing the number of page faults.
  - The CPU utilization drops further $\Rightarrow$ the cycle repeats
  - Here we see that Thrashing occurs and system throughput decreases dramatically: **No real work is being done**.

■ **Severe Performance Issue due to Thrashing**

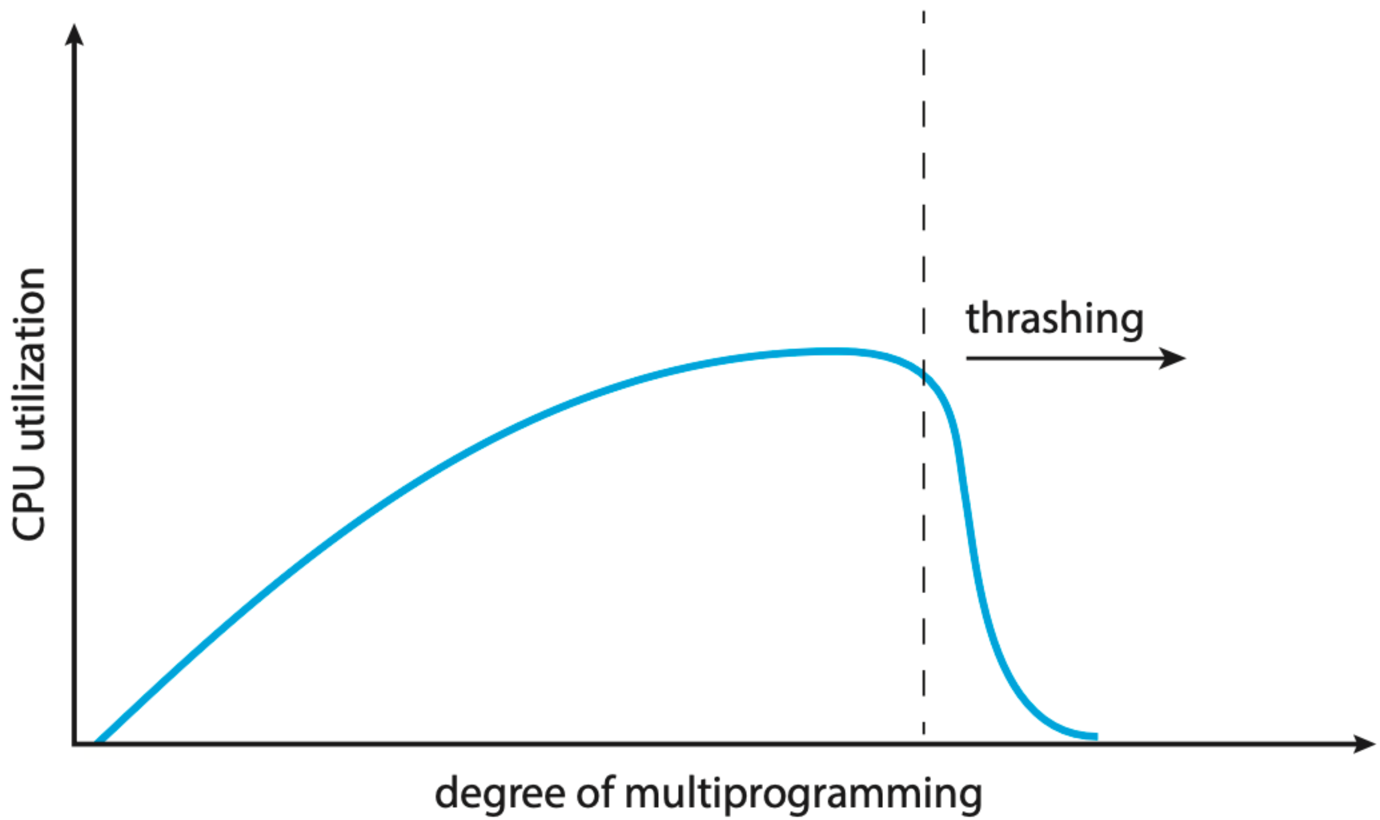


Figure 10.20 Thrashing.

- **Thrashing**
  - If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
    - low CPU utilization
    - OS spends most of its time swapping in and out of disk
  - **Thrashing**: a process is busy swapping pages in and out with little or no actual progress.
  - **Question:**
    - How do we detect **Thrashing**?
    - What is the best response to **Thrashing**?

Working Set Model

- How to prevent **Thrashing**?
  - We must provide a process with as many frames as it needs.
  - But how do we know how many frames it "needs"?
- We make use of the **Working Set Strategy** where it checks how many frames a process is actually using.
  - This approach defines the **locality model** of process execution.
  - As a process executes, it moves from **locality** to **locality**.
    - A **locality** is a set of pages that are actively used together.
  - A running program is generally composed of several different **localities**, which may overlap.

Thrashing

- **Working Set Model**
  - **Locality (局部性)** of a process changes over time.
  - At time (a), the **locality** is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}.
  - At time (b), the **locality** changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}.
  - Notice the overlap, as some pages are (e.g., **{18, 19, 20}**) are part of both localities.

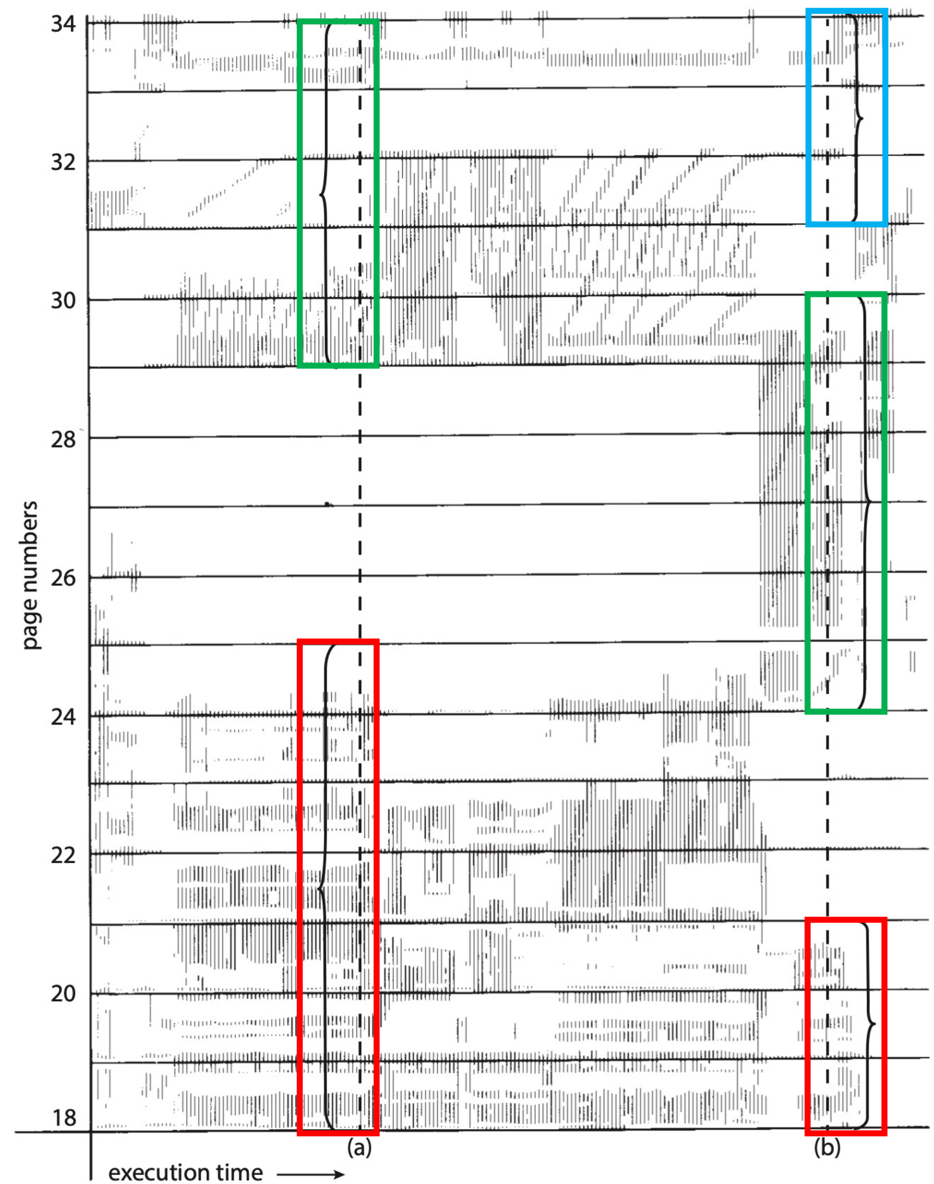
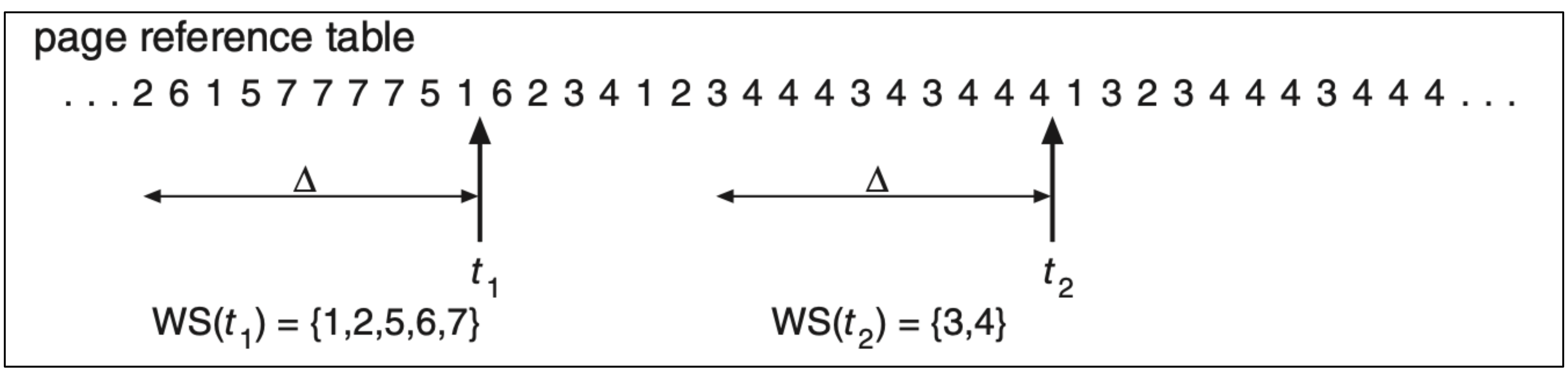


Figure 10.21 Locality in a memory-reference pattern.

Working Set Model

- The **Working Set Model** is based on the assumption of **locality**.
  - We use a parameter Δ to define the working set window.
  - The set of pages in the most recent Δ page references is the **working set**.
  - If a page is in active use, it will be in the **working set**.
  - If it is no longer being used, it will drop from the **working set** Δ **time units after its last reference**.
  - Thus, the **working set** is an approximation of the program's **locality**.
- For example: $\Delta = 10$.
  - The working set at time t_1 is {1, 2, 5, 6, 7}, with size of 5 pages.
  - The working set at time t_2 is {3, 4}, with size of 2 pages.



Working Set Model

- The accuracy of the **working set** depends on the selection of Δ
  - If Δ is **too large** $\Rightarrow$ it may overlap serveral localities.
  - If Δ is **too small** $\Rightarrow$ it may not cover the entire localities.
  - If Δ is **infinite** $\Rightarrow$ the working set is the set of pages touched during the process execution.
- The most important property of the **working set** is its **size**.
  - The **working set size** of a process is denoted by WSS_i
  - so the total **demand** for frames in a system is

$$D = \sum WSS_i$$

  - If the total demand is greater than the total number of available frames, i.e., $D > m$, then **Thrashing** will occur.
    - In this case, the OS will select a process to suspend. In other words, the process's pages are swapped out and its frames are reallocated to other processes.

Working Set Model

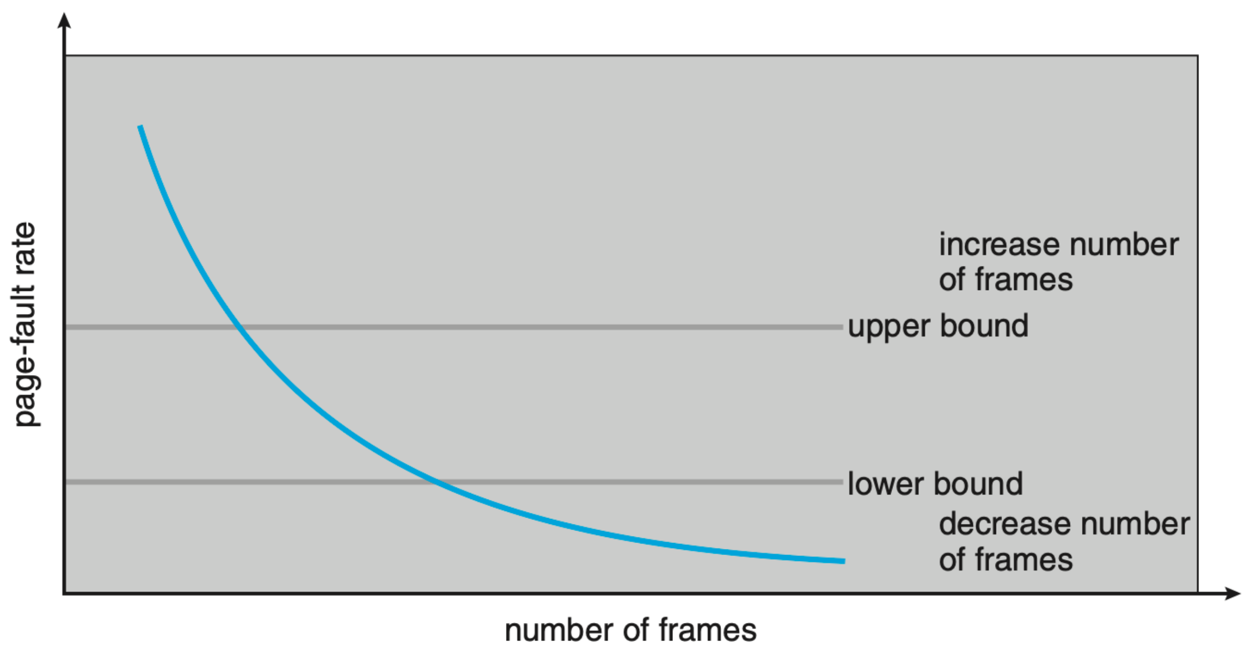
- The most important property of the **working set** is its **size**.
  - The **working set size** of a process is denoted by WSS_i
  - so the total **demand** for frames in a system is

$$D = \sum WSS_i$$

  - If the total demand is greater than the total number of available frames, i.e., $\boxed{D > m\text{, then Thrashing will occur.}}$
    - In this case, the OS will select a process to suspend. In other words, the process's pages are swapped out and its frames are reallocated to other processes.
- The **working set strategy prevents thrashing** while keeping the degree of multiprogramming as **high** as possible
  - Thus, optimizing CPU utilization.

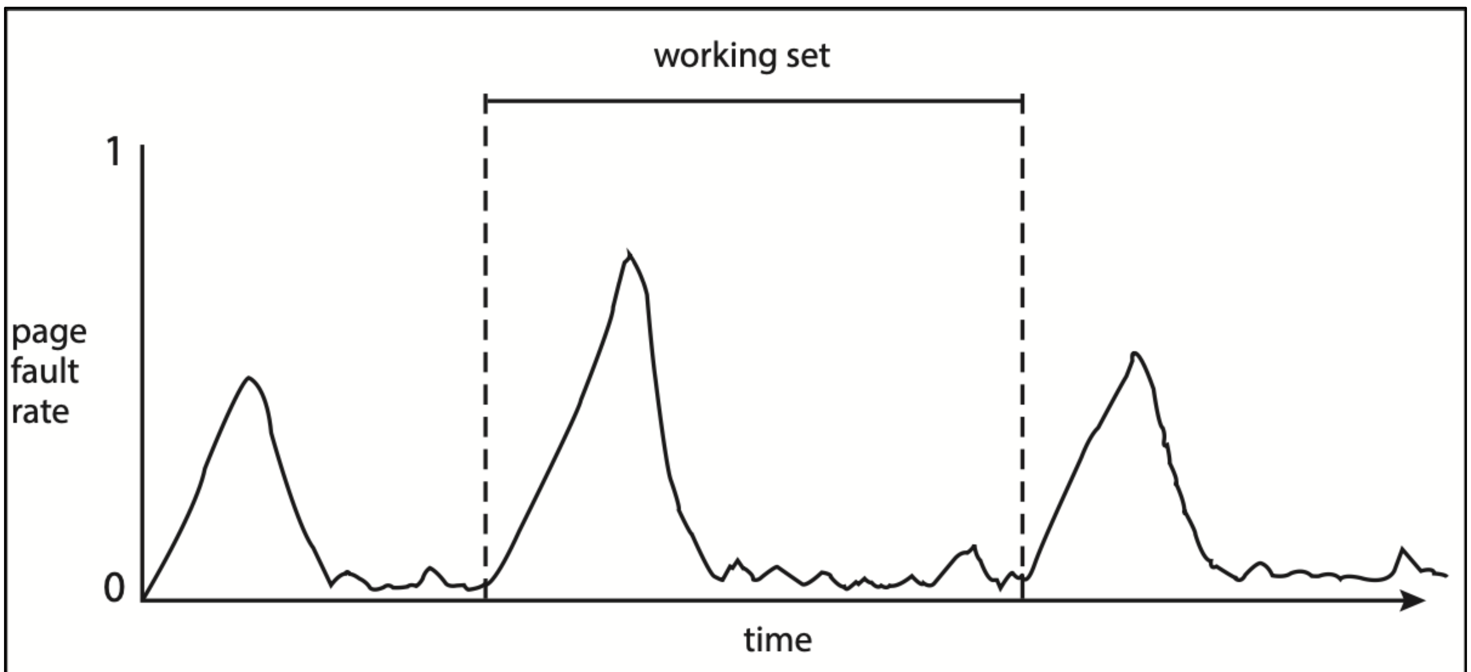
Working Set Model

- The working set model is successful, but it seems too clumsy.
- A strategy that uses the **Page-Fault Frequency** (**PFF, 缺页率**) takes a more **direct** approach: **Control** the page-fault rate.
  - If it is too high, we know the process needs more frames and we allocate the process another frame.
  - If it is too low, then the process may have too many frames and we remove a frame from the process.



Working Set Model

- There is a direct relationship between **working set** of a process and its **page-fault rate**.
  - **Working set** changes over time.
  - **Page-fault rate** peaks and valleys over time.
  - A **peak** in the page-fault rate occurs when we begin demand-paging a new locality; Once the working set of this new locality is in memory, the page-fault-rate **falls**.



- **Allocating Kernel Memory**
  - When a user-mode process requests additional memory (e.g., via `malloc`), pages are **allocated** from the list of free frames maintained by the **kernel**.
  - However, memory allocation inside the kernel is different. When the kernel requests additional memory (e.g., via `kmalloc`), pages are often allocated from a **different** free-memory pool.
    - The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory **conservatively** and attempt to **minimize waste** due to fragmentation.
    - Pages allocated to user-mode processes do not necessarily have to be **contiguous physical memory**. However, certain hardware devices interact directly with physical memory (*without the benefit of a virtual memory interface, since VM is part of the kernel*), and consequently may require memory residing in physically **contiguous pages**.

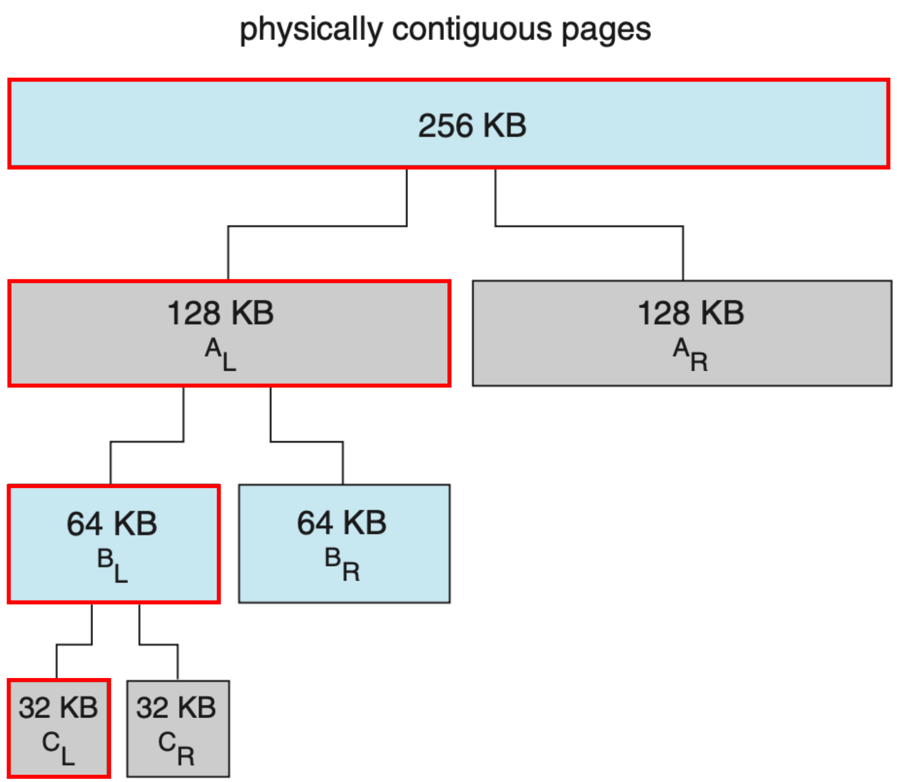
Allocating Kernel Memory

- **Buddy System**
  - Buddy system allocates memory from a fixed-size segment consisting of **physical contiguous pages**.
  - Memory is allocated from this segment using a **power-of-2 allocator**.
    - Satisfies requests in units sizes as **a power of 2** (e.g., **4KB, 8KB, 16KB…**)
    - Request *not appropriately sized* **rounded up** to next highest power of 2
    - When smaller allocation needed than is available, current chunk split into two **buddies** of next-lower power of 2
  - Assume a 256KB chunk (A) available, and the kernel requests **21KB**:
    - Split into A_L and A_R (each 128KB in size)
      - A_L further split into B_L and B_R (each 64KB in size)
        - B_L further split into C_L and C_R (each 32KB in size)
          - C_L is allocated to satisfy the **21KB** request.

Allocating Kernel Memory

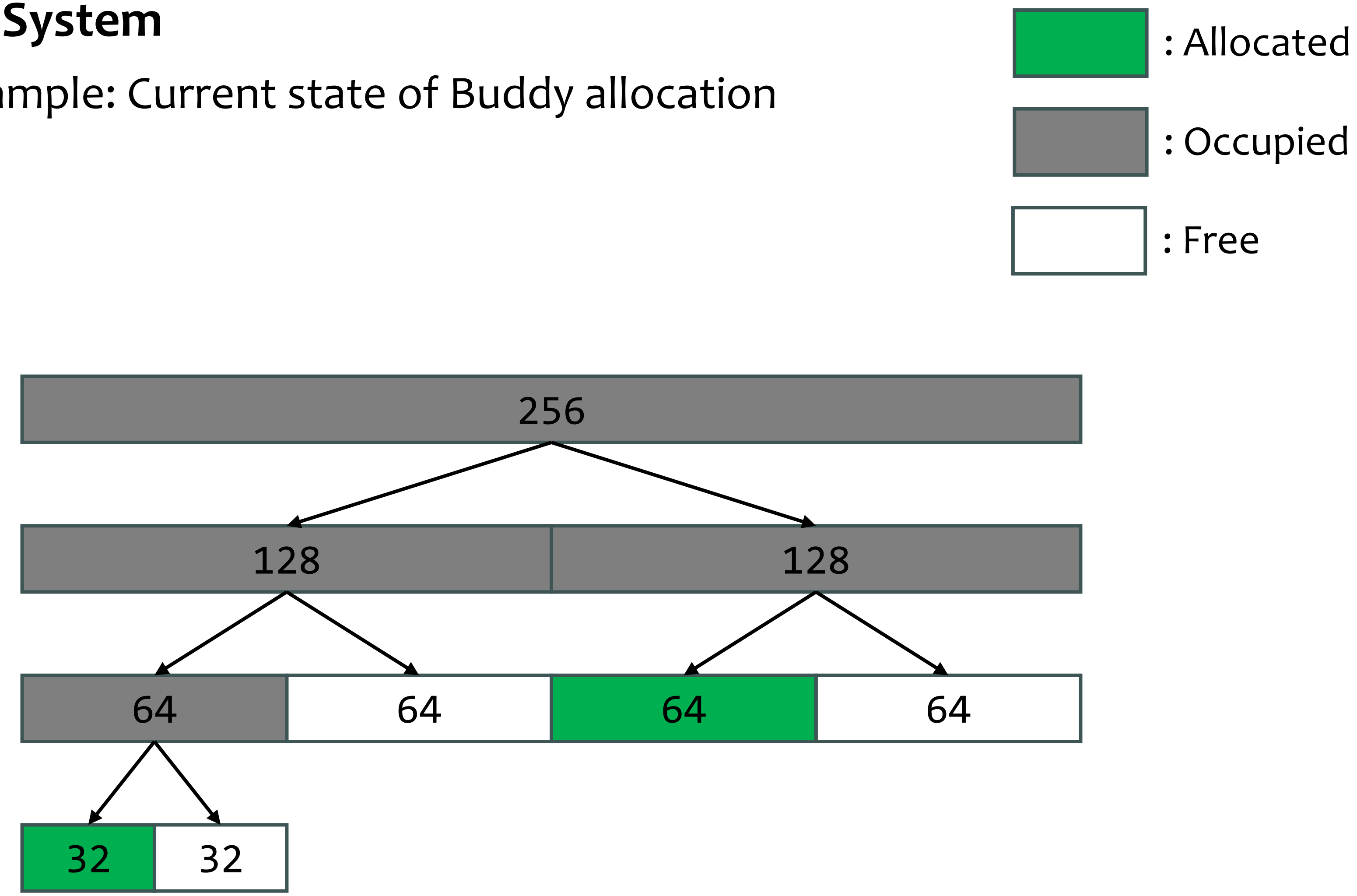
Buddy System

- Assume a 256KB chunk (A) available, and the kernel requests **21KB**:
  - Split into A_L and A_R (each 128KB in size)
    - A_L further split into B_L and B_R (each 64KB in size)
      - B_L further split into C_L and C_R (each 32KB in size)
        - C_L is allocated to satisfy the **21KB** request.



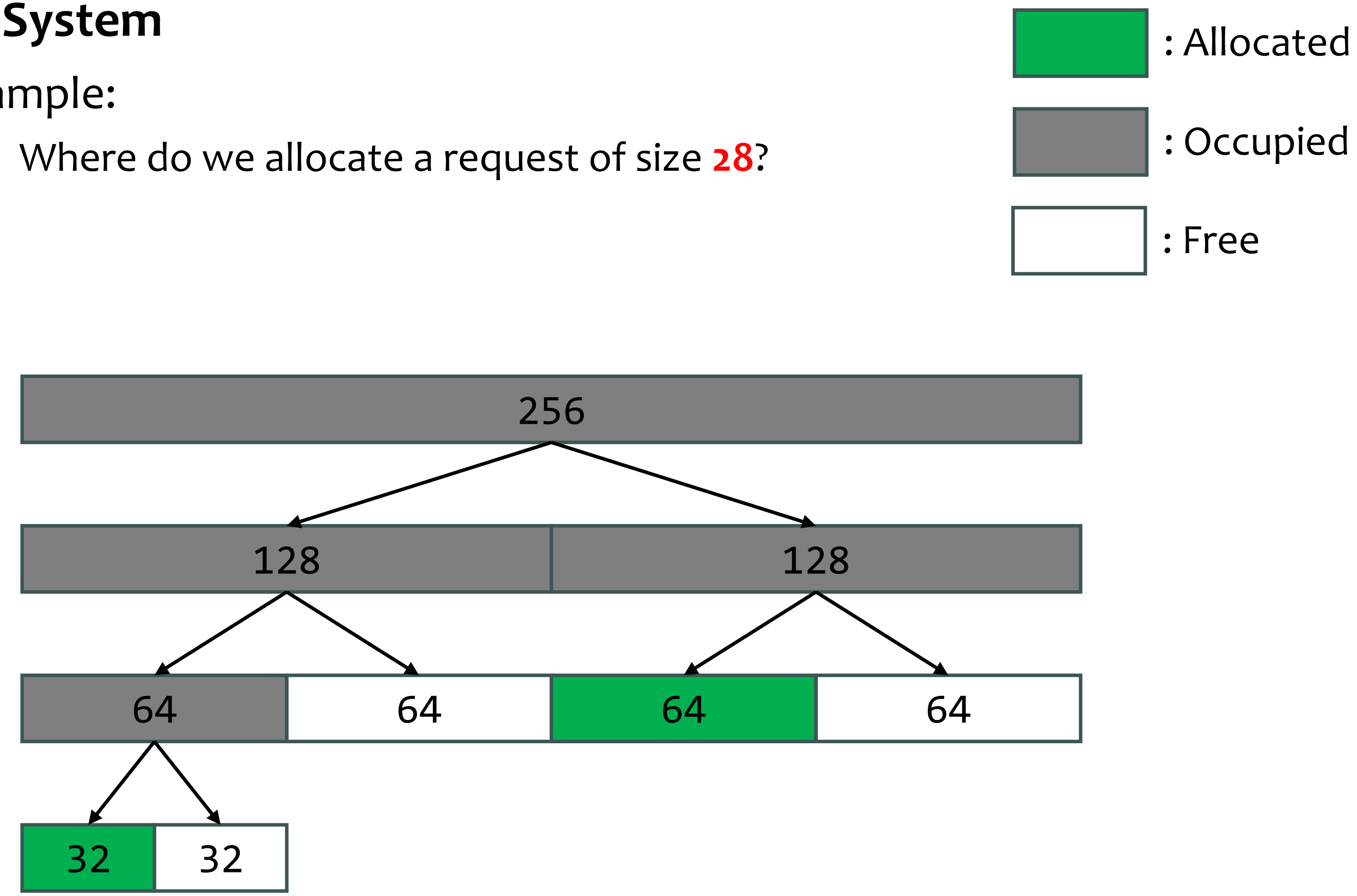
Allocating Kernel Memory

- **Buddy System**
  - Example: Current state of Buddy allocation

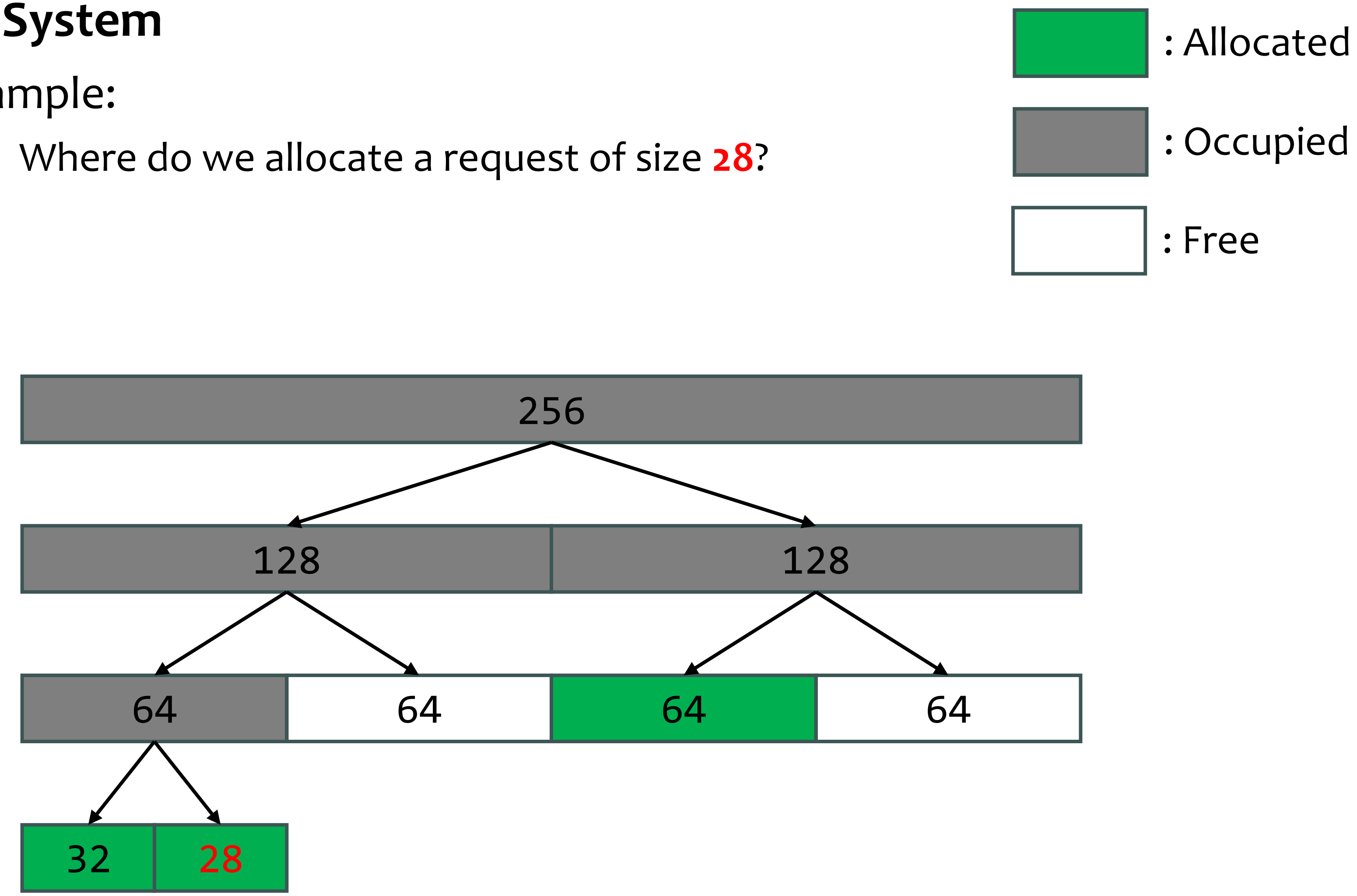


Allocating Kernel Memory

- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?

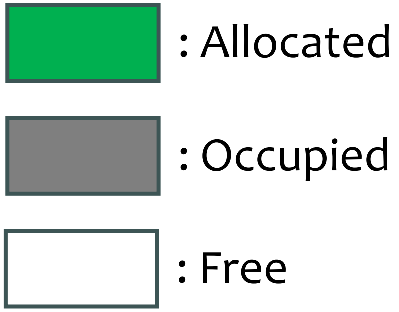


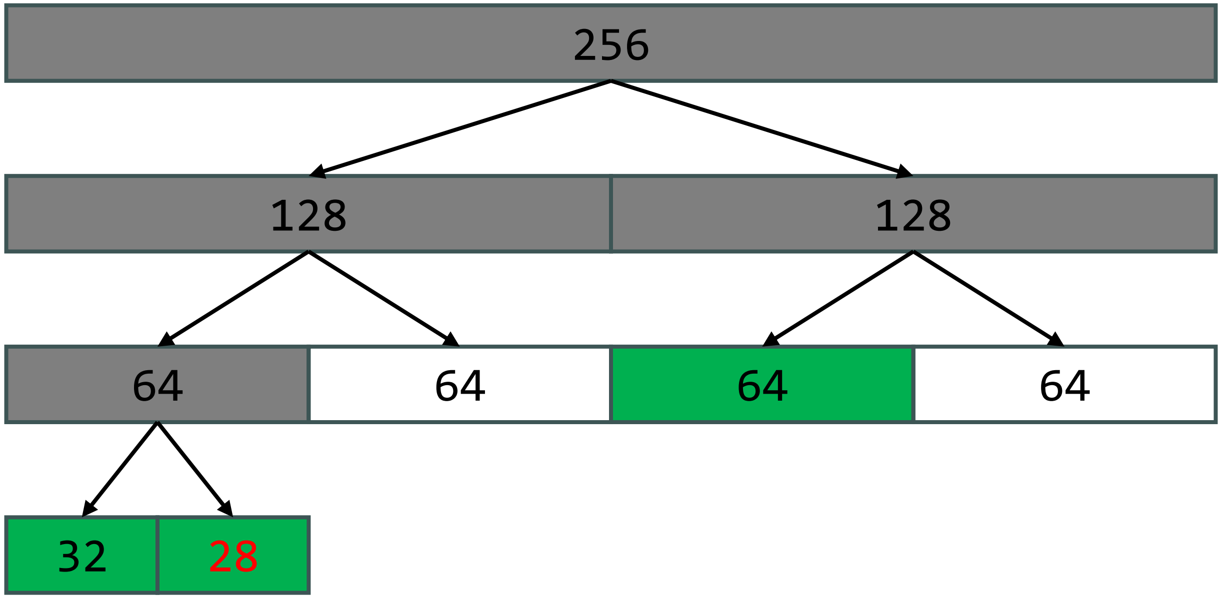
- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?



Allocating Kernel Memory

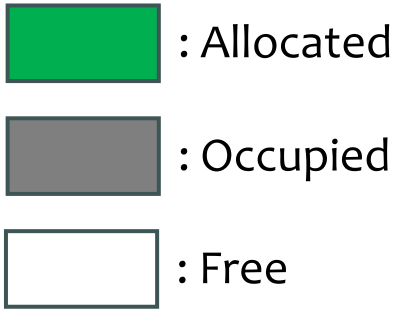
- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?
    - Where do we allocate a request of size 36?

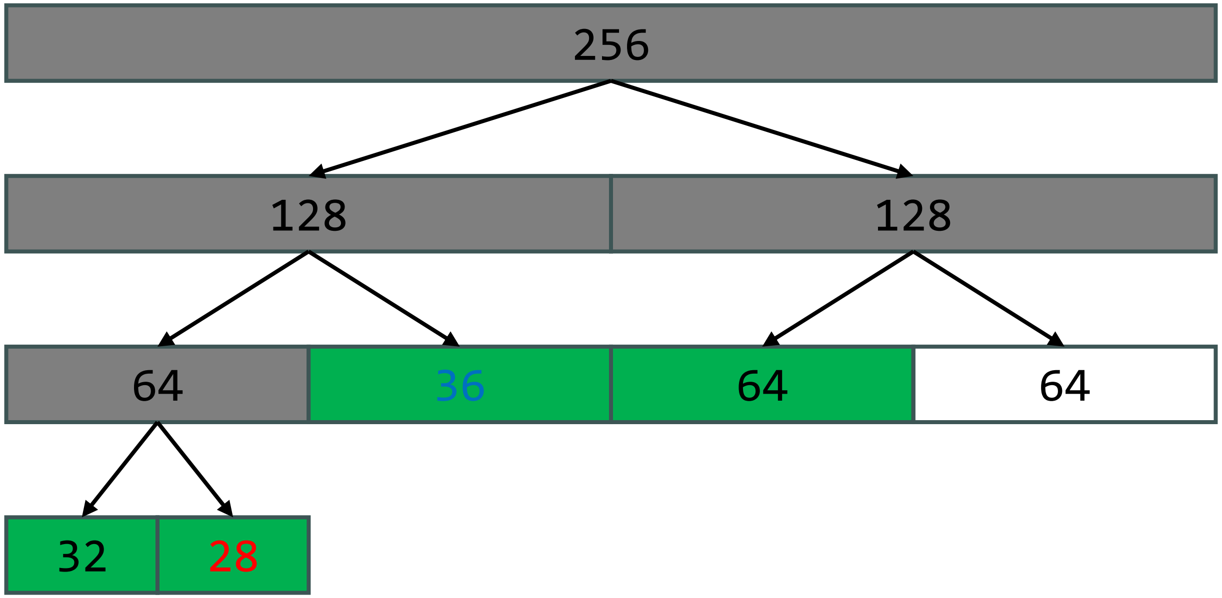




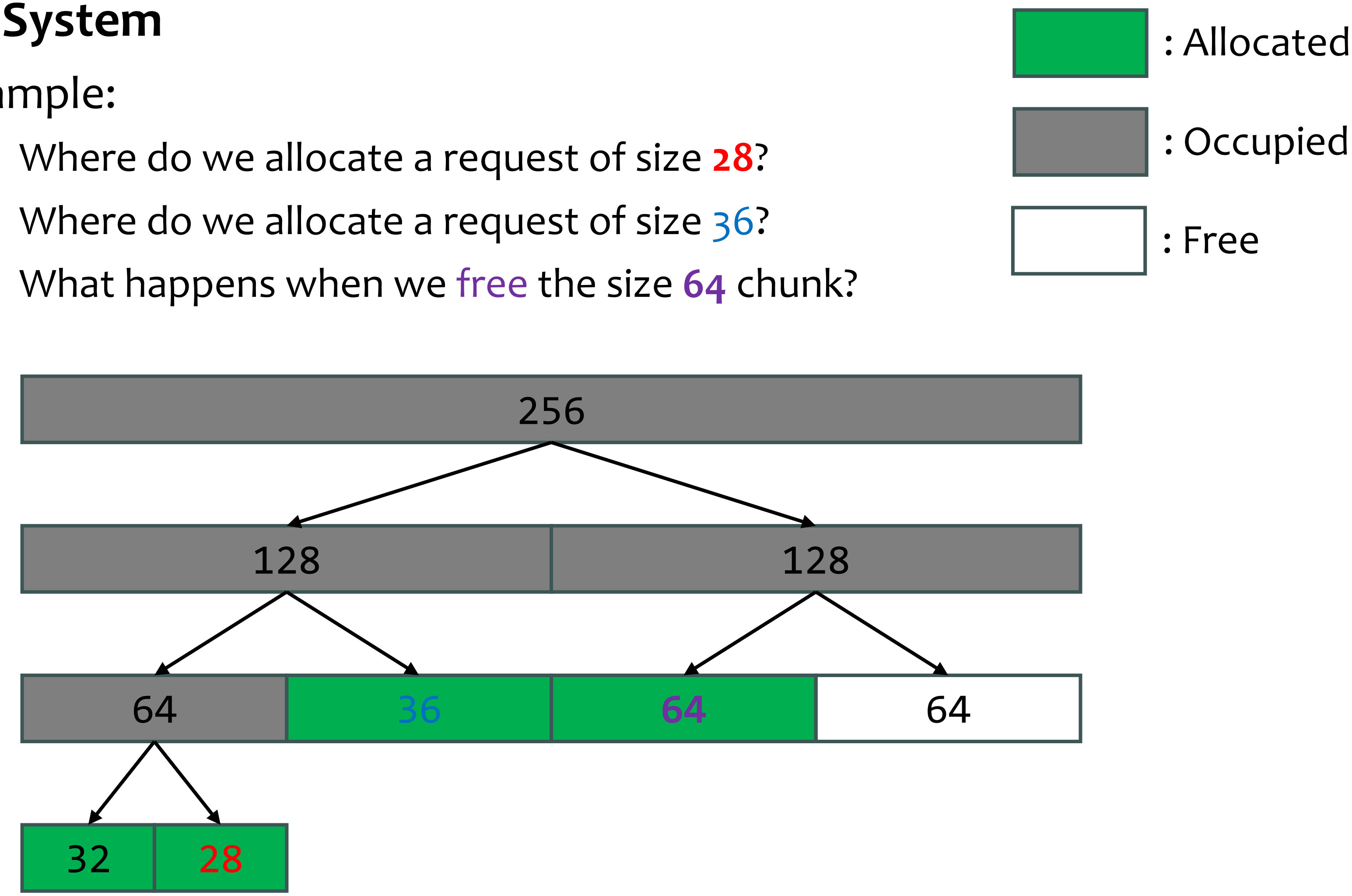
Allocating Kernel Memory

- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?
    - Where do we allocate a request of size 36?



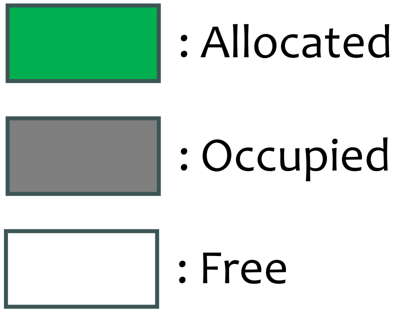


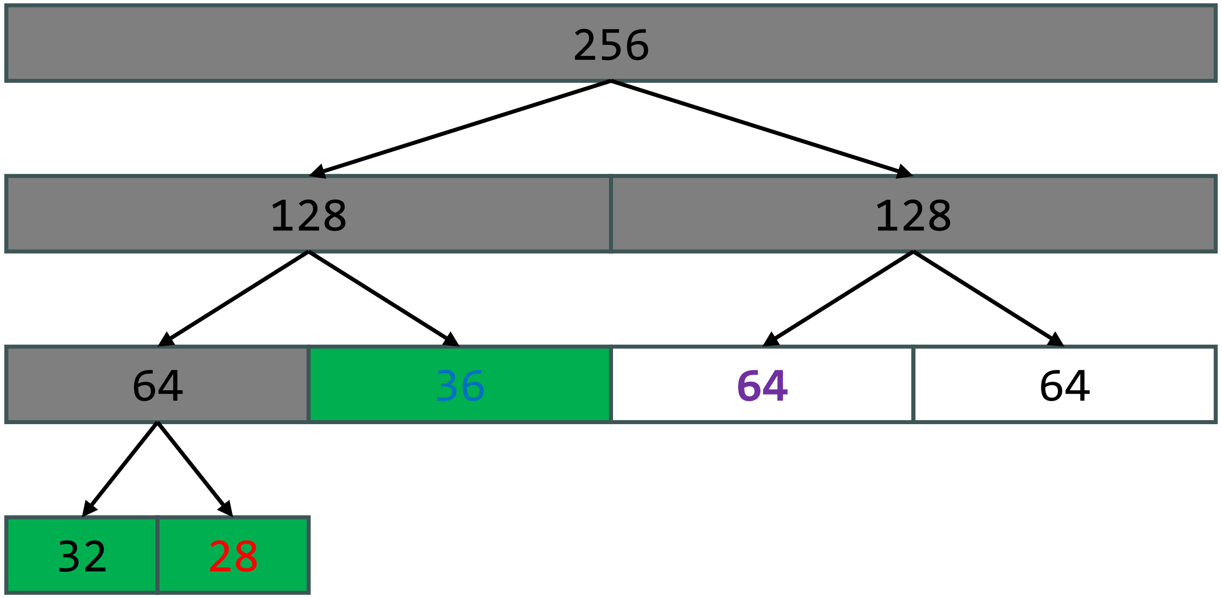
- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?
    - Where do we allocate a request of size 36?
    - What happens when we free the size **64** chunk?



Allocating Kernel Memory

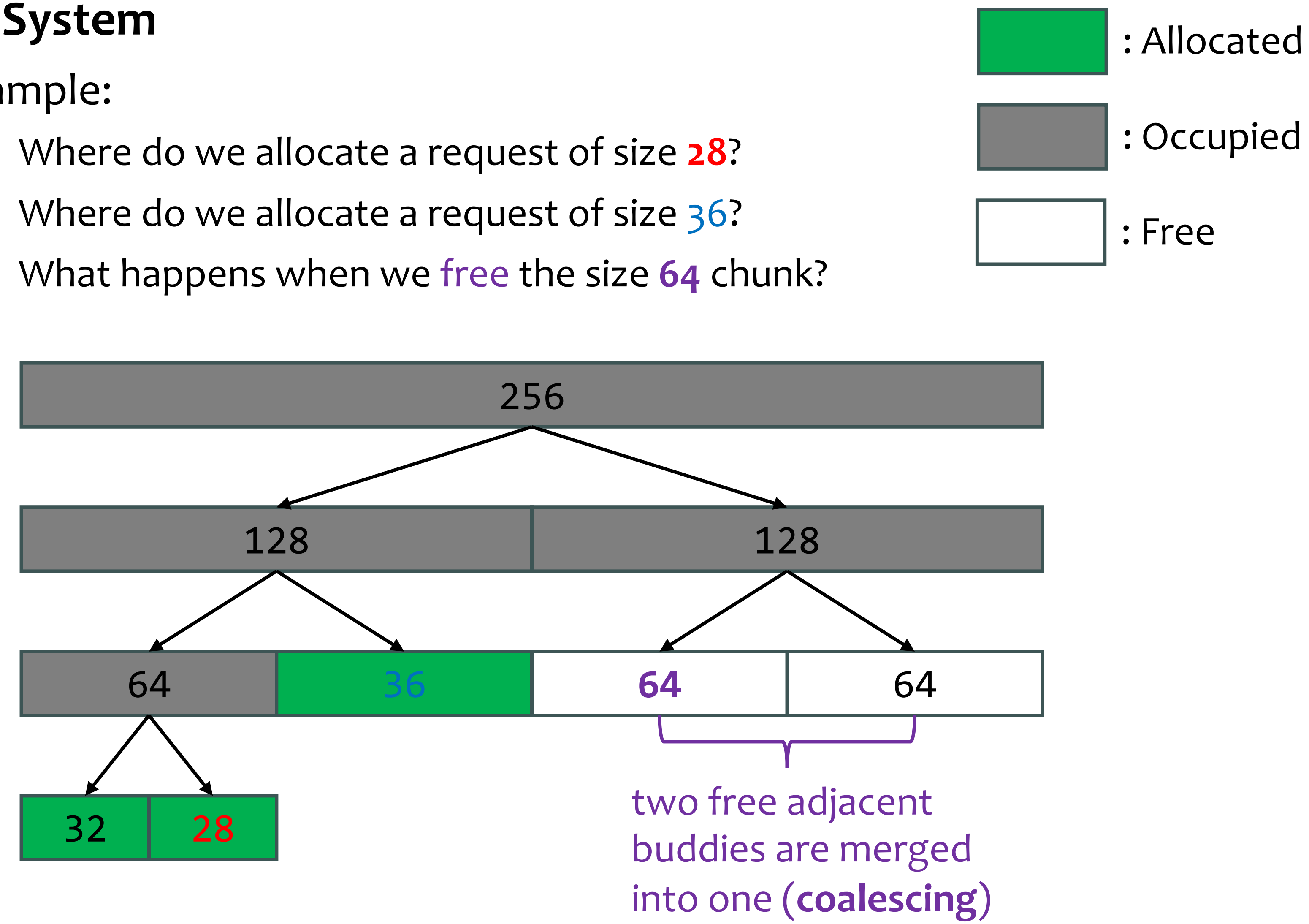
- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?
    - Where do we allocate a request of size 36?
    - What happens when we free the size **64** chunk?





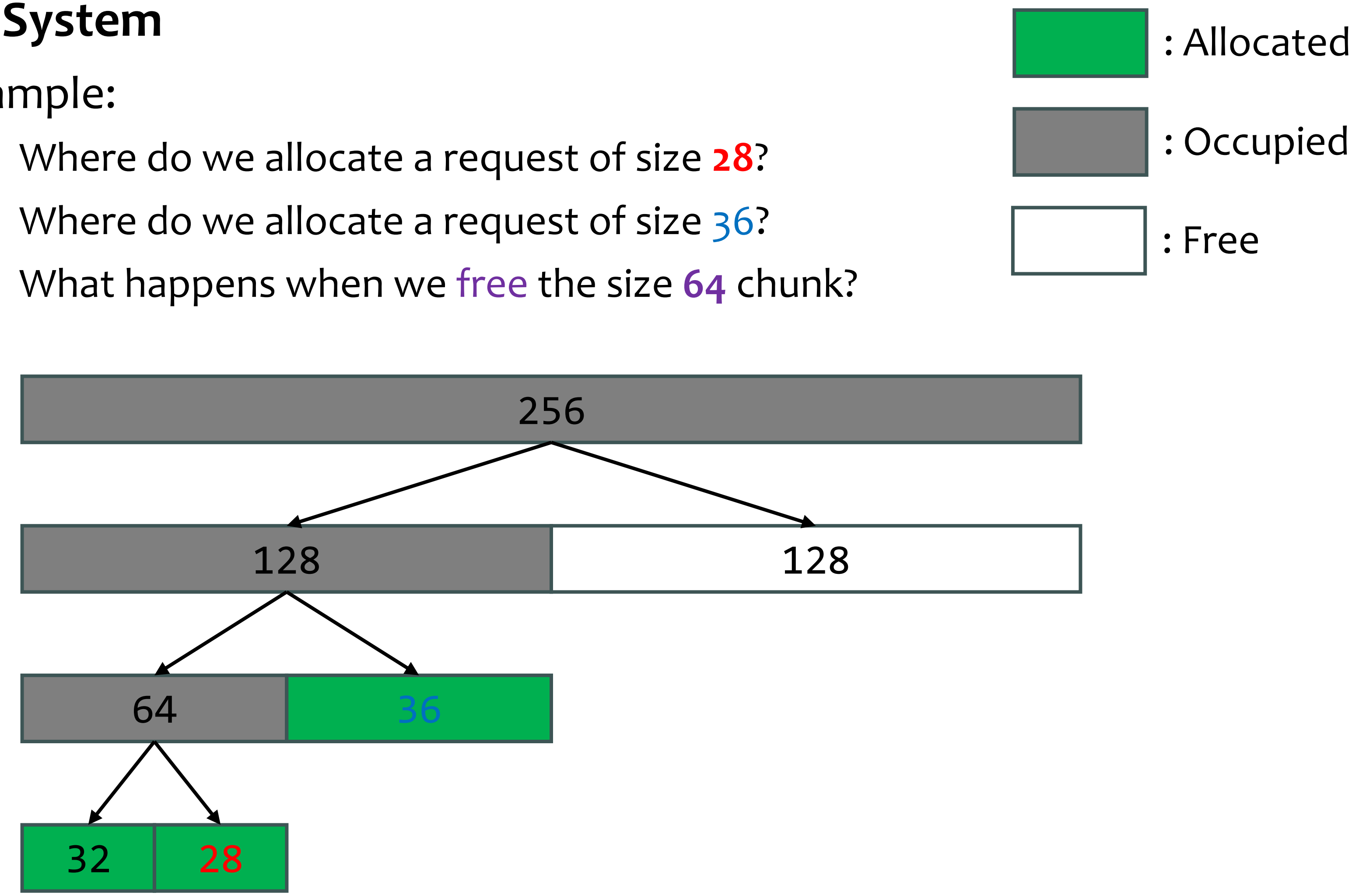
Buddy System

- Example:
  - Where do we allocate a request of size **28**?
  - Where do we allocate a request of size 36?
  - What happens when we free the size **64** chunk?



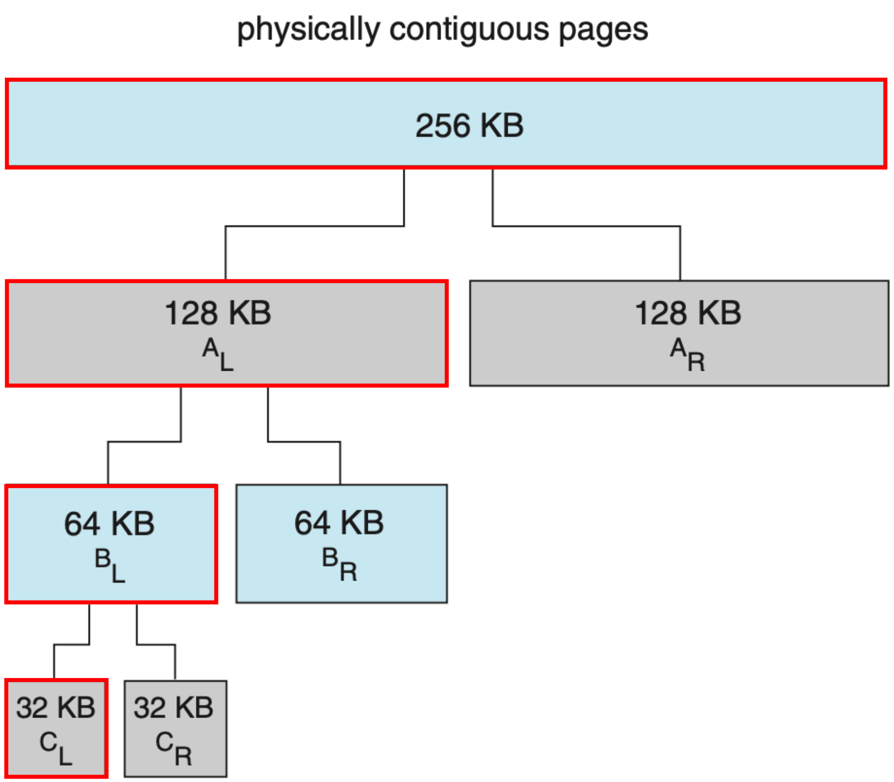
Allocating Kernel Memory

- **Buddy System**
  - Example:
    - Where do we allocate a request of size **28**?
    - Where do we allocate a request of size 36?
    - What happens when we free the size **64** chunk?



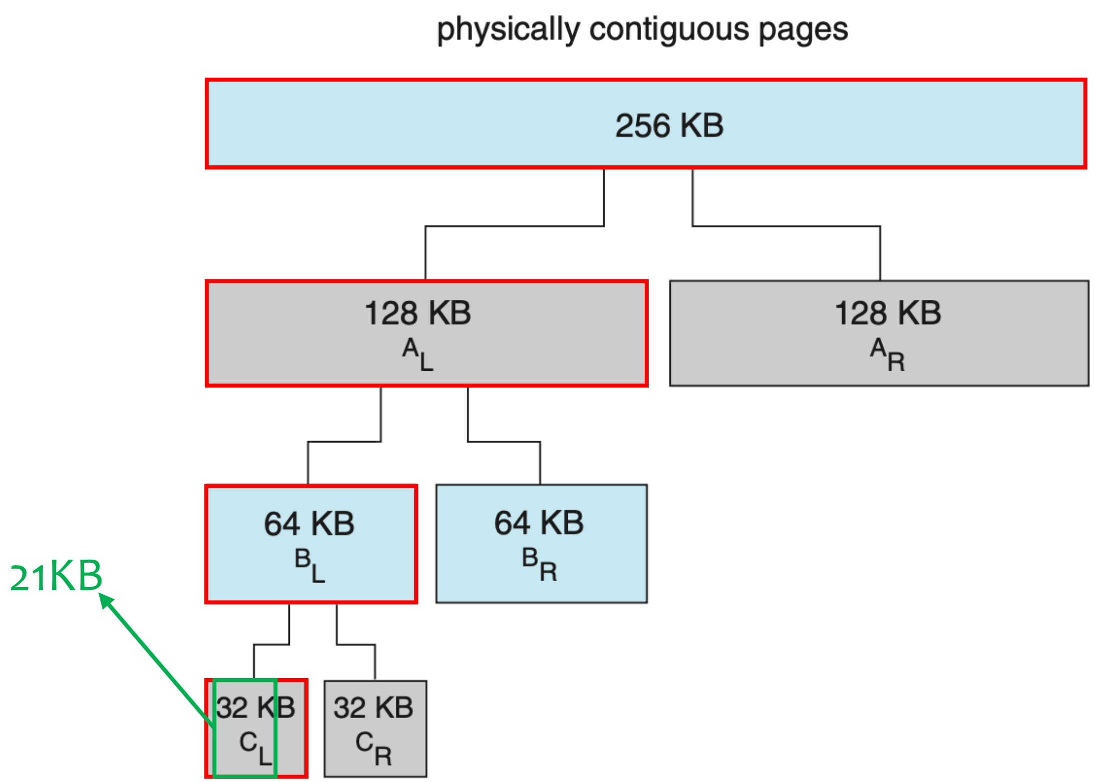
Buddy System

- **Advantage:** Adjacent buddies can be combined to form larger segments **quickly** using a technique known as **coalescing**.
  - For example, when the kernel releases C_L, the system can coalesce C_L and C_R into a 64KB segment B_L, which in turn can be coalesced with B_R to form a 128KB segment A_L, and so on...



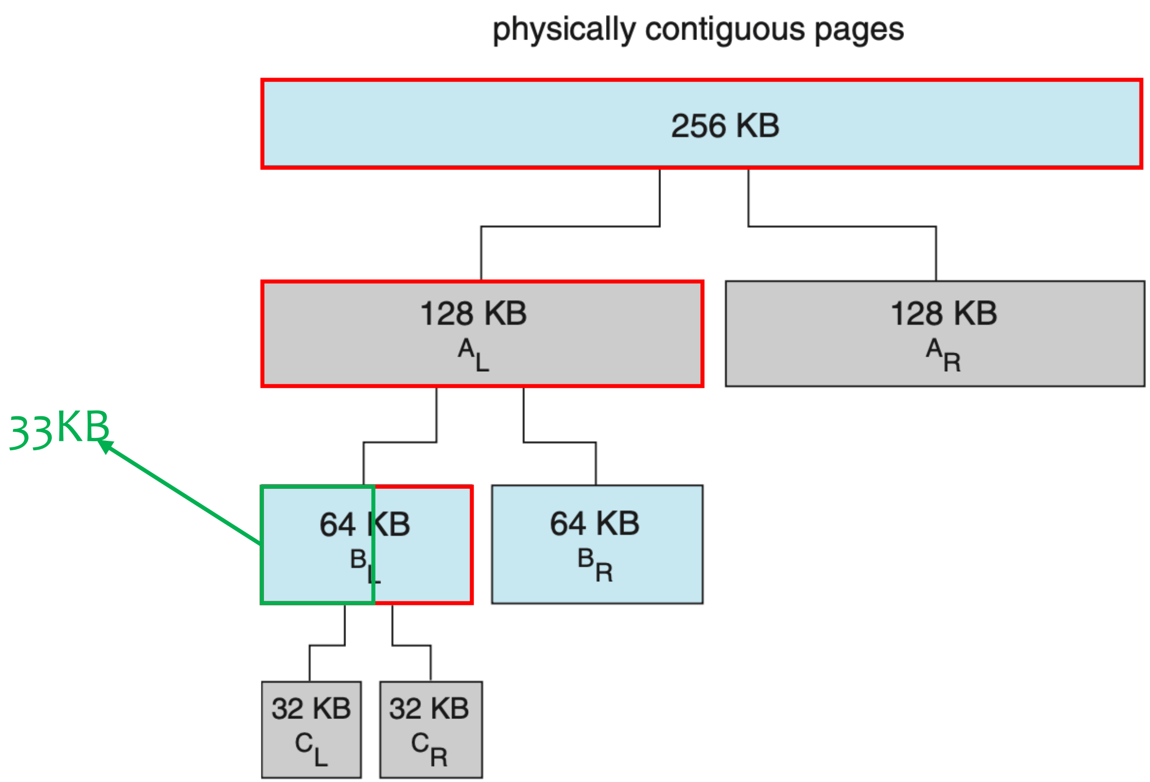
Allocating Kernel Memory

- **Buddy System**
  - **Disadvantage**: Rounding up to the next highest power of 2 is very likely to cause **fragmentation** within allocated segments.
    - For example, the **21KB** request is satisfied with a **32KB** segment, with **11KB** wasted.



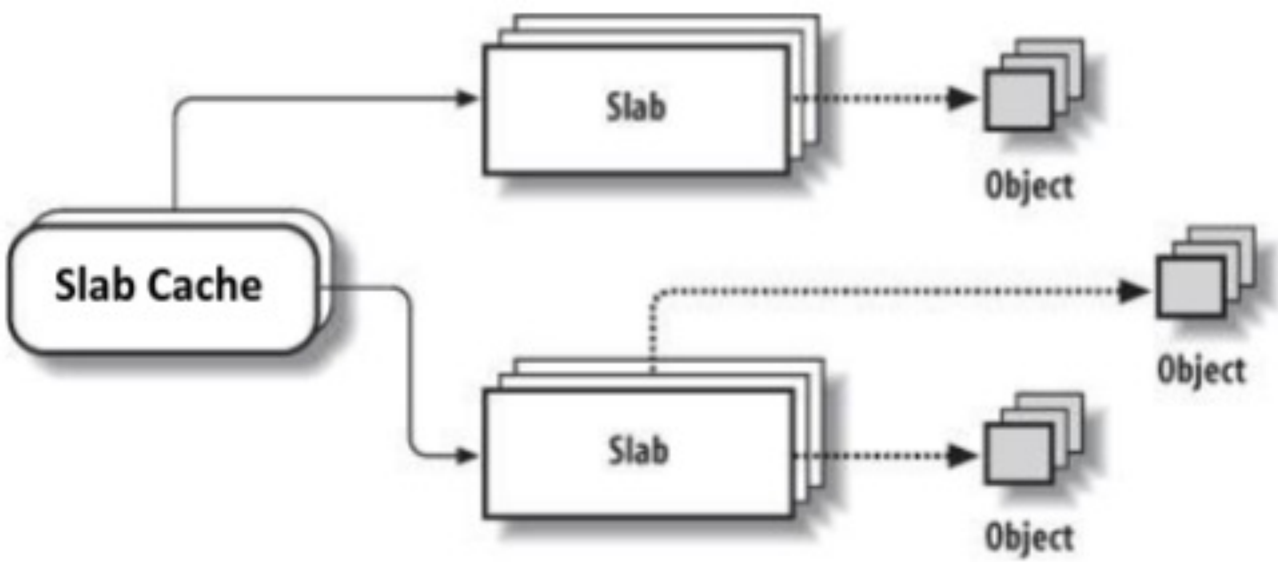
Allocating Kernel Memory

- **Buddy System**
  - **Disadvantage**: Rounding up to the next highest power of 2 is very likely to cause **fragmentation** within allocated segments.
    - For example, the **21KB** request is satisfied with a **32KB** segment, with **11KB** wasted.
    - Worse, consider a **33KB** request that is satisfied with a **64KB** segment.

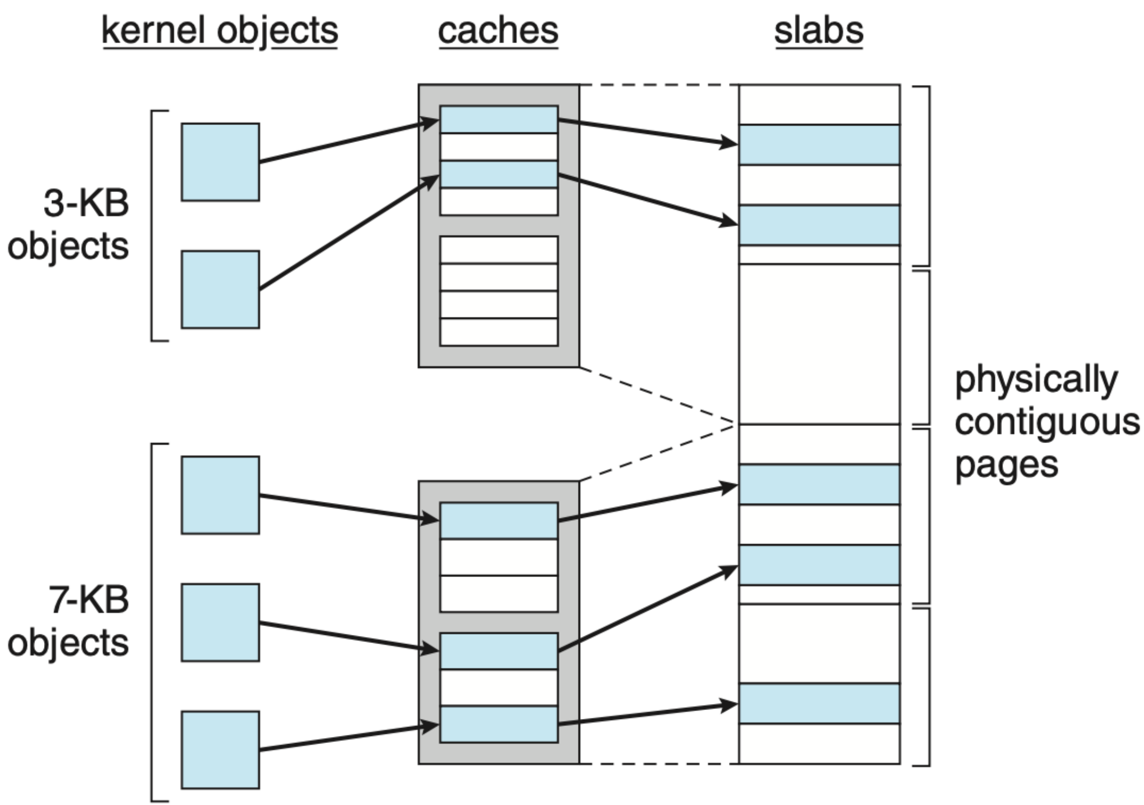


Slab Allocation

- **Slab allocation**: an alternative strategy for allocating kernel memory.
  - A **slab** is made up of one or more physically contiguous pages/frames.
  - A **cache** consists of one or more **slabs**.
  - There is a single cache for **each unique** kernel data structure

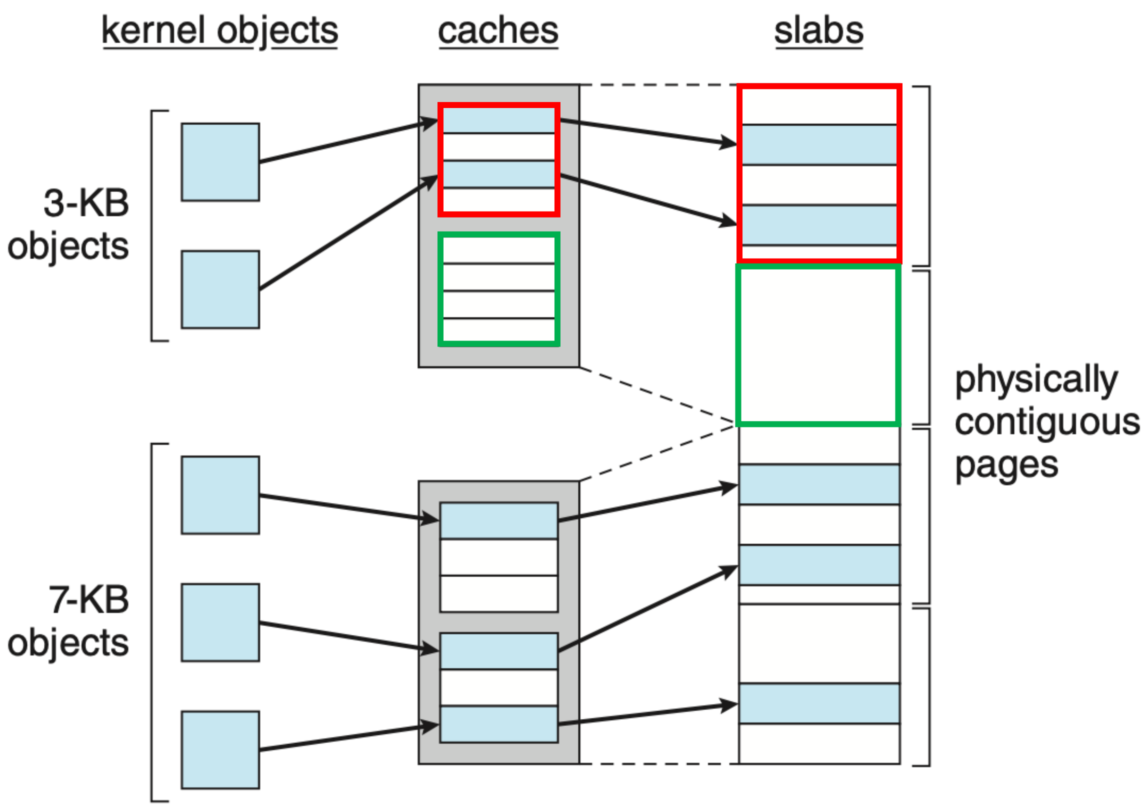


- **Slab Allocation**
  - Slab allocation: an alternative strategy for allocating kernel memory.
    - A slab is made up of one or more physically contiguous pages/frames.
    - A cache consists of one or more slabs.
    - There is a single cache for **each unique** kernel data structure
      - For example, all **3KB** objects belong to the same cache.



Slab Allocation

- **Slab allocation**: an alternative strategy for allocating kernel memory.
  - A **slab** is made up of one or more physically contiguous pages/frames.
  - A **cache** consists of one or more **slabs**.
  - There is a single cache for **each unique** kernel data structure
    - For example, all **3KB** objects belong to the same cache.



Slab Allocation

- The slab allocation algorithm uses **caches** to store kernel objects.
- When a **cache** is created, a number of objects (which are initially marked as free) are allocated to the **cache**.
  - The number of objects in the cache depends on the size of the associated slab.
    - For example, a 12KB slab (made up of 3 contiguous 4KB frames) could store six 2KB objects.
  - Initially, all objects in the cache are marked as **free**.
  - When a new object for a kernel data structure is needed, the allocated can assign any free object from the cache to satisfy the request.
    - The object assigned from the cache is marked as **used**.
- In Linux, a slab may be in one of **three** possible states:
  - **Full**. All objects in the slab are marked as **used**.
  - **Empty**. All objects in the slab are marked as **free**.
  - **Partial**. Mix of **used** and **free** objects.

Slab Allocation

- In Linux, a slab may be in one of **three** possible states:
  - **Full**. All objects in the slab are marked as **used**.
  - **Empty**. All objects in the slab are marked as **free**.
  - **Partial**. Mix of **used** and **free** objects.

- Upon request from the kernel:
  - The **SLAB** allocator first attempts to find a (**partial**) slab consisting free objects to satisfy the request.
  - If none exists, a free object is assigned from an **empty** slab
  - If no **empty** slabs are available, a new slab is allocated from contiguous physical frames and assigned to a cache
    - memory for the object is allocated from this new slab.

Allocating Kernel Memory

- **Slab Allocation**
  - The slab allocator provides two main benefits:
    - No memory is wasted due to fragmentation.
    - Memory requests can be satisfied quickly.

Allocating Kernel Memory

- **SLAB → SLOB → SLUB**
  - Linux originally used the buddy system. (< version 2.2)
  - Beginning with version 2.2, the Linux kernel adopted the **SLAB** allocator.
  - Recent distributions of Linux include the **SLOB** kernel allocators:
    - **SLOB** allocator is designed for systems with a limited amount of memory (e.g., embedded systems).
    - **SLOB** stands for **S**imple **L**ist **O**f **B**locks.
  - **SLOB** maintains three lists of objects:
    - **small** (for objects < **256** bytes)
    - **medium** (for objects < **1024** bytes)
    - **large** (for all other objects < **page size**)

Allocating Kernel Memory

- **SLAB → SLOB → SLUB**
  - Linux originally used the buddy system. (< version 2.2)
  - Beginning with version 2.2, the Linux kernel adopted the **SLAB** allocator.
  - Recent distributions of Linux include the **SLOB** kernel allocators:
    - **SLOB** allocator is designed for systems with a limited amount of memory (e.g., embedded systems).
    - **SLOB** stands for **S**imple **L**ist **O**f **B**locks.
  - **SLOB** maintains three lists of objects:
    - **small** (for objects < **256** bytes)
    - **medium** (for objects < **1024** bytes)
    - **large** (for all other objects < **page size**)
  - Memory requests are allocated from an object on the appropriate list using a **first-fit** policy.

Allocating Kernel Memory

- **SLAB → SLOB → SLUB**
  - Linux originally used the buddy system. (< version 2.2)
  - Beginning with version 2.2, the Linux kernel adopted the **SLAB** allocator.
  - Beginning with Version 2.6.24, the **SLUB** allocator replaced **SLAB** as the default allocator for the Linux kernel.
    - **SLUB** is basically a performance-optimized **SLAB**.
    - **SLUB** reduced much of the overhead required by the **SLAB** allocator.
    - For instance, whereas **SLAB** stores certain metadata with each slab, **SLUB** stores these data in the page structure the Linux kernel uses for each page.
    - Additionally, **SLUB** does not include the per-CPU queues that the **SLAB** allocator maintains for objects in each cache. For systems with a large number of processors, the amount of memory allocated to these queues is significant. Thus, **SLUB** provides better performance as the number of processors on a system increases.

Other Considerations

- **Other Considerations**
  - Prepaging
  - Page Size
  - TLB Reach
  - Inverted Page Table
  - Program Structure
  - I/O Interlock and Page Locking

- **Prepaging**
  - To reduce the large number of page faults that occurs at process startup
  - **Prepage** (预先调取页面) *all* or *some* of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used.
    - Is cost of $s \times \alpha$ save page faults **larger** or **less** than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
    - $\alpha \to 0 \Rightarrow$ prepaging is not worth it.

Other Considerations

Page Size

- Sometimes the OS designers have a choice in choosing the **Page Size**.
  - especially if running on custom-built CPU
- **Page size** selection must take into consideration:
  - Fragmentation
  - Page Table Size
  - Resolution
  - I/O overhead
  - Number of Page Faults
  - Locality
  - TLB Size and effectiveness
- Always a power of 2, usually in the range between 2^{12} and 2^{22} bytes.

Architecture	Huge Page Size
ARM64	4K, 2M, 1G
i386	4K, 4M
x86_64	4K, 8K, 64K, 256K, 1M, 4M, 16M, 256M
ppc64	4K, 16M

Other Considerations

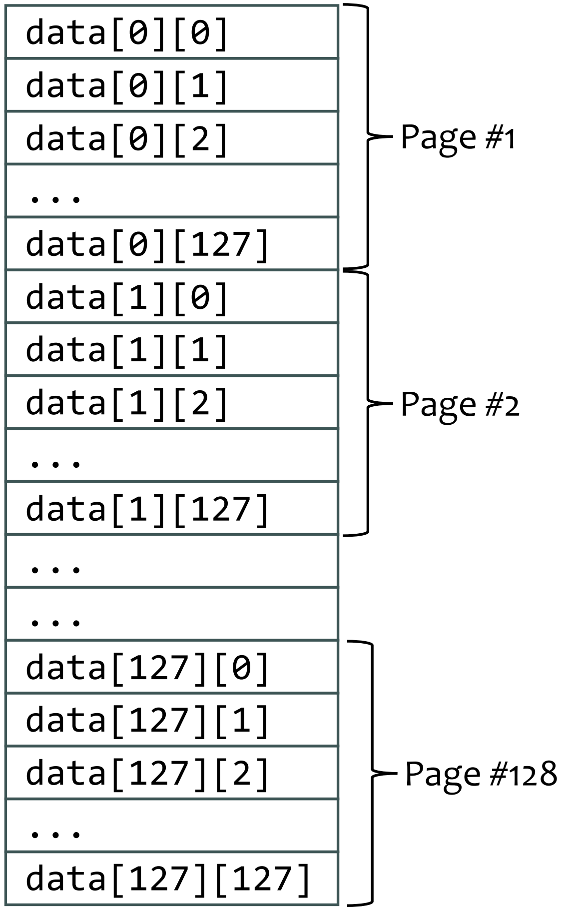
- **TLB Reach (TLB范围)**
  - **TLB Reach** - The amount of memory accessible from the TLB
  - **TLB Reach** = (**TLB Size**) × (**Page Size**)
  - Ideally, the working set of each process is stored in the TLB
    - Otherwise there is a high degree of page faults
  - Increase the Page Size
    - ⇒ This may lead to an increase in fragmentation as not all applications require a large page size
  - Provide Multiple Page Sizes
    - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```



Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **1**

data[0][0]	Page #1
data[0][1]	
data[0][2]	
...	
data[0][127]	
data[1][0]	Page #2
data[1][1]	
data[1][2]	
...	
data[1][127]	
...	
...	
data[127][0]	Page #128
data[127][1]	
data[127][2]	
...	
data[127][127]	

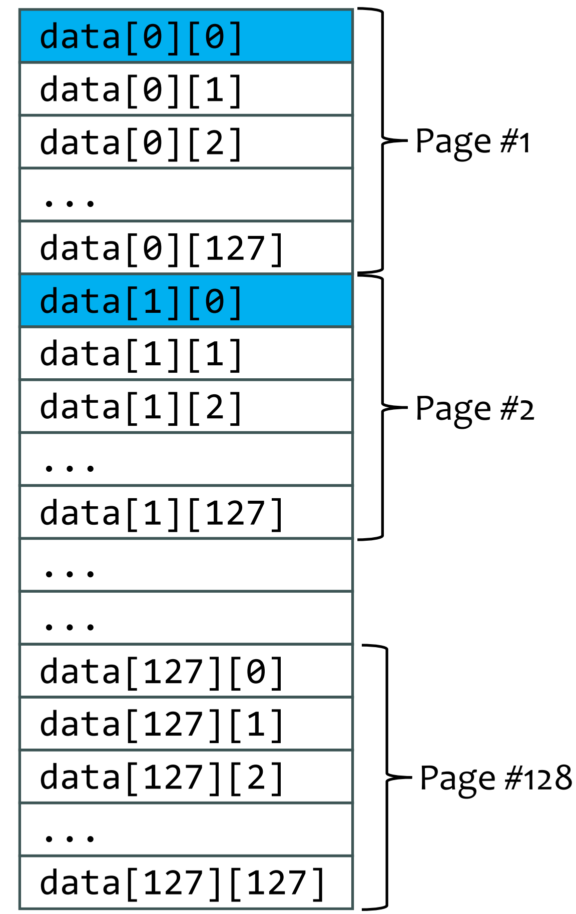
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **2**

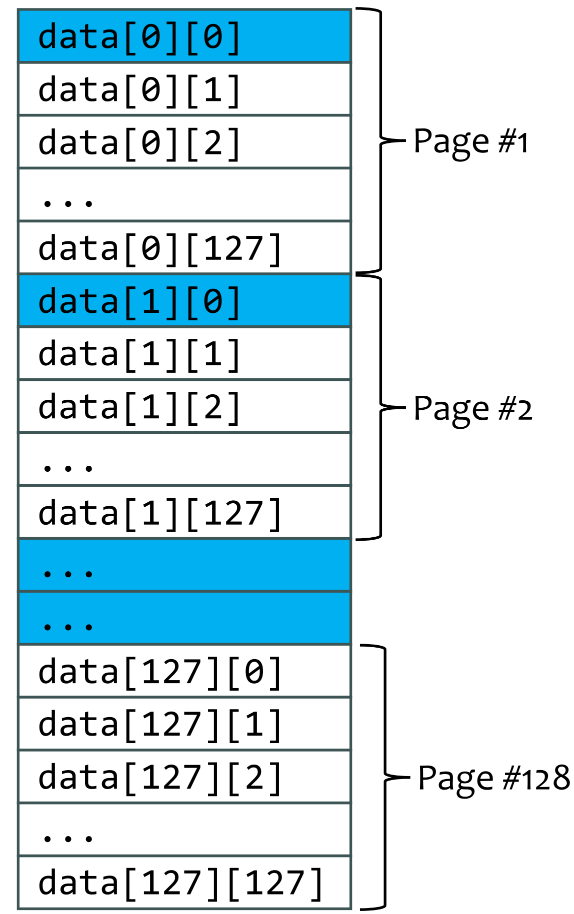


Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **127**



Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128** (FIFO evict Page #1)

data[0][0]	Page #1
data[0][1]	
data[0][2]	
...	
data[0][127]	
data[1][0]	Page #2
data[1][1]	
data[1][2]	
...	
data[1][127]	
...	
...	
data[127][0]	Page #128
data[127][1]	
data[127][2]	
...	
data[127][127]	

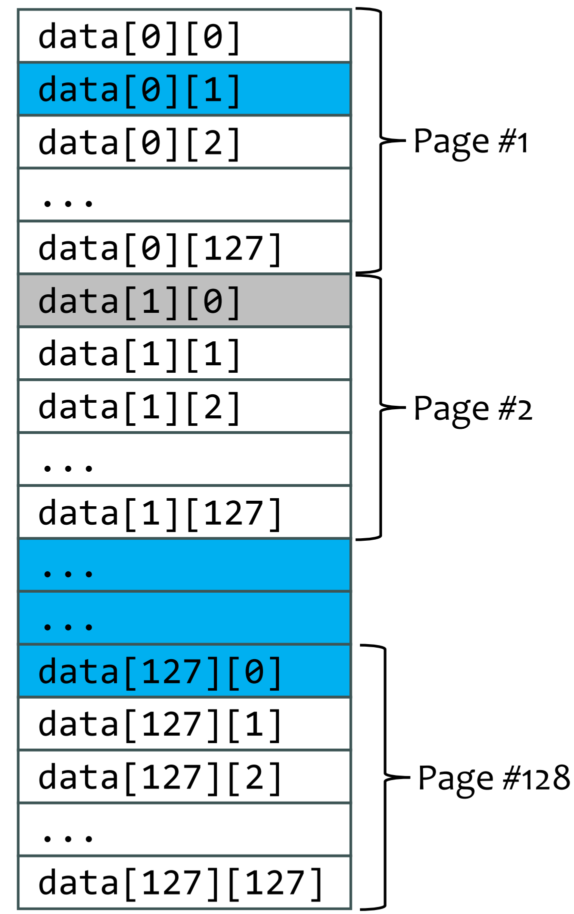
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **129**



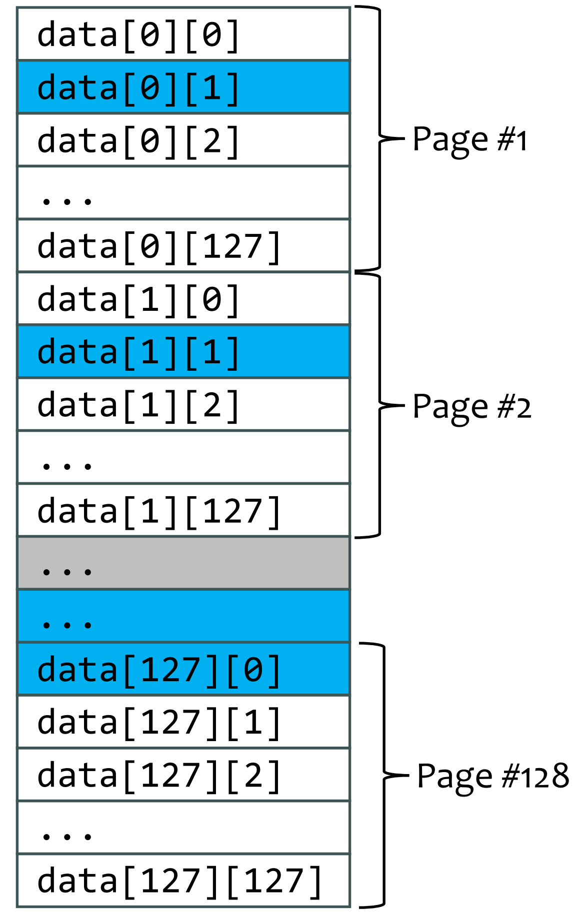
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **130**



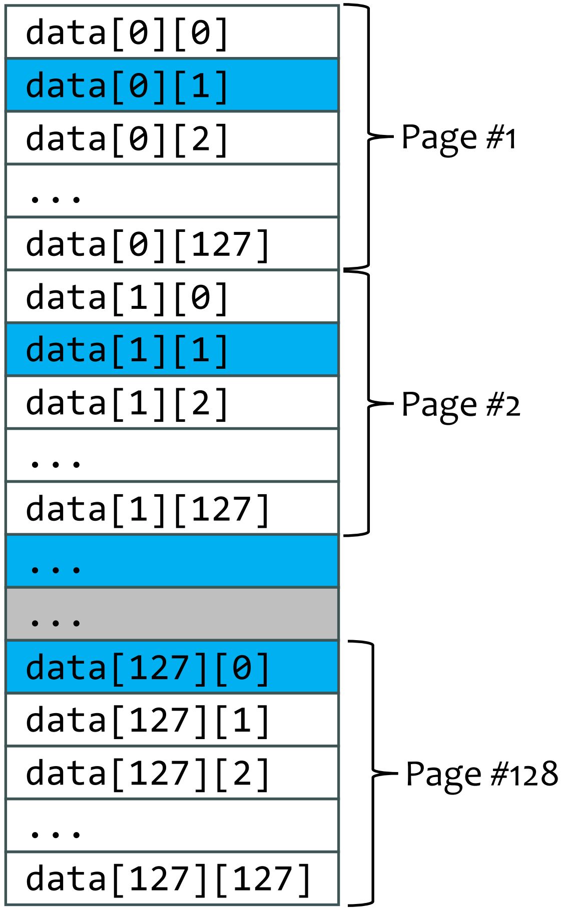
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **254**



Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **255**

data[0][0]	Page #1
data[0][1]	
data[0][2]	
...	
data[0][127]	
data[1][0]	Page #2
data[1][1]	
data[1][2]	
...	
data[1][127]	
...	
...	
data[127][0]	Page #128
data[127][1]	
data[127][2]	
...	
data[127][127]	

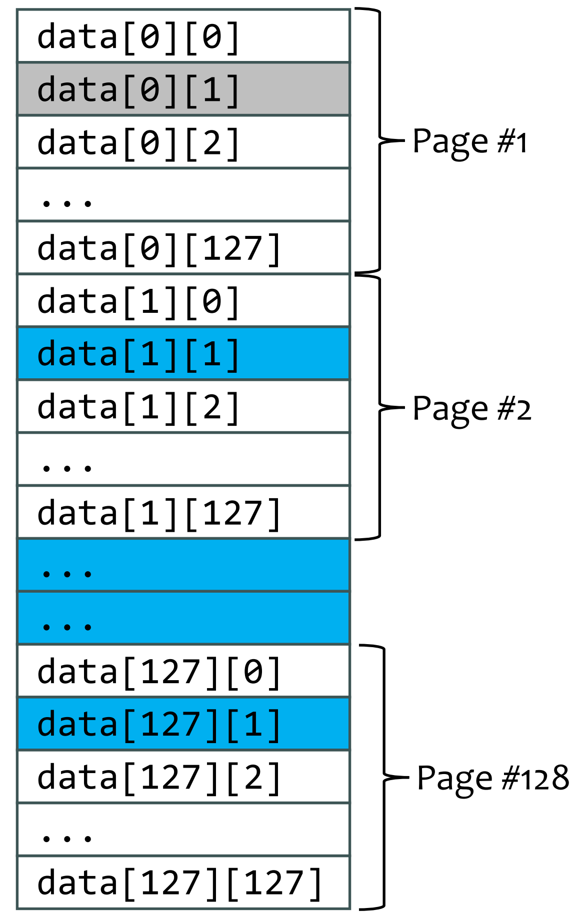
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **256**



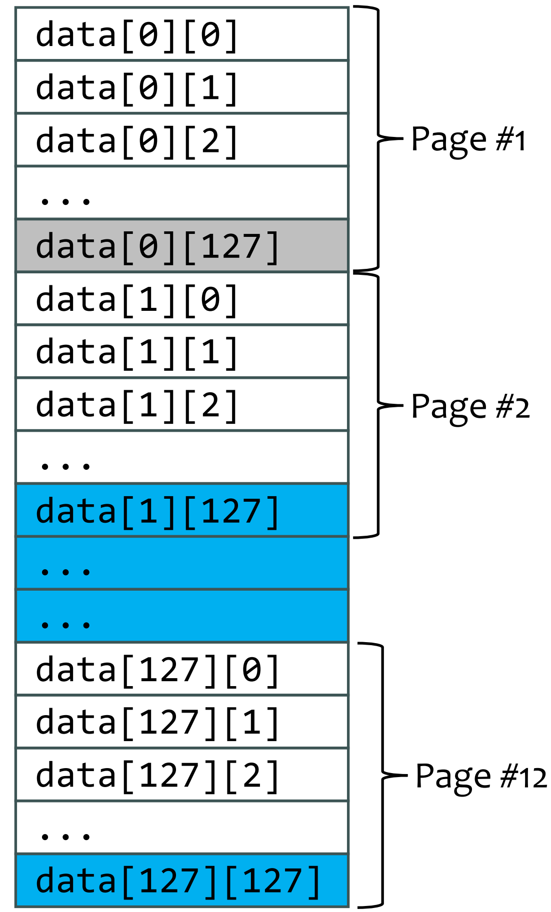
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults:
128 x 128 = 16384



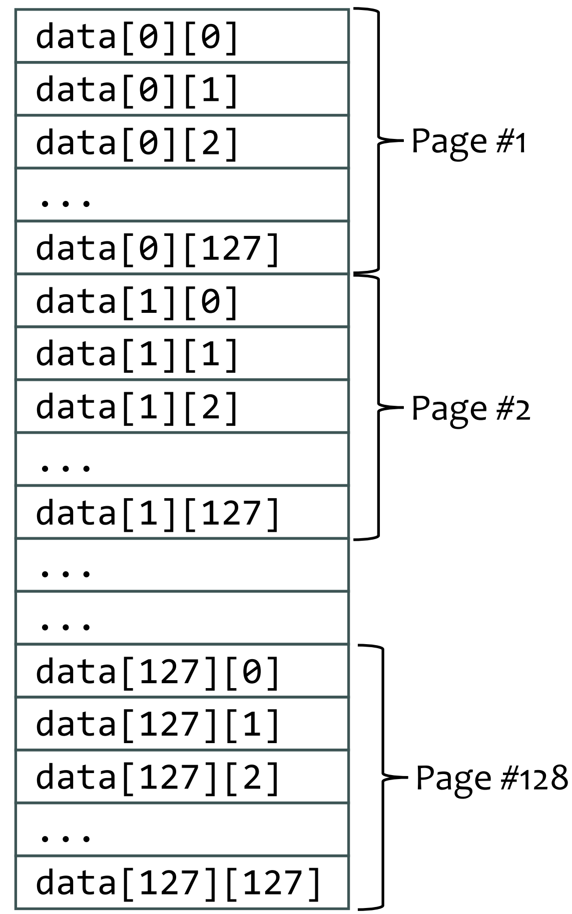
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128 x 128 = 16384**



- An optimized (cache-friendly) version:

```
int i, j;
int [128][128] data;

for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
        data[i][j] = 0;
    }
}
```

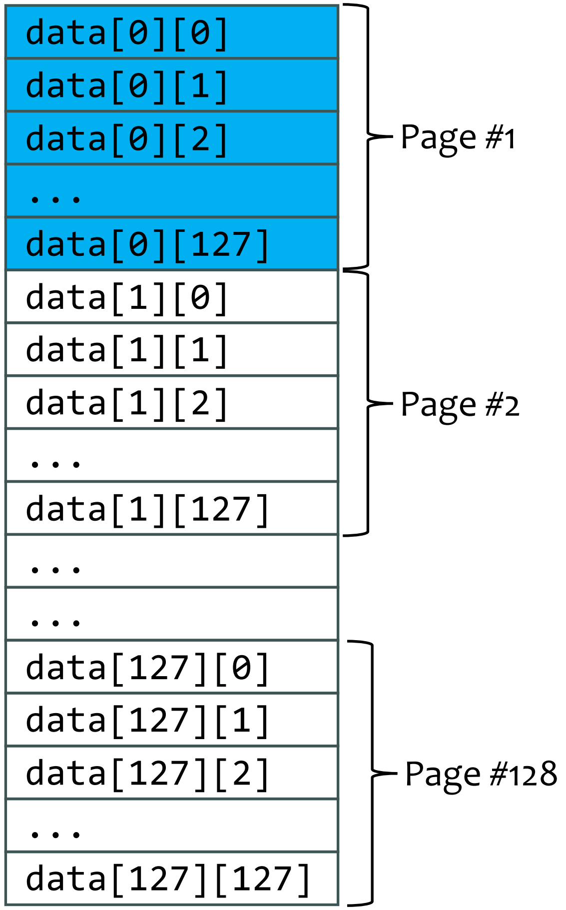
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128 x 128 = 16384**



- An optimized (cache-friendly) version:

```
int i, j;
int [128][128] data;

for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **1**

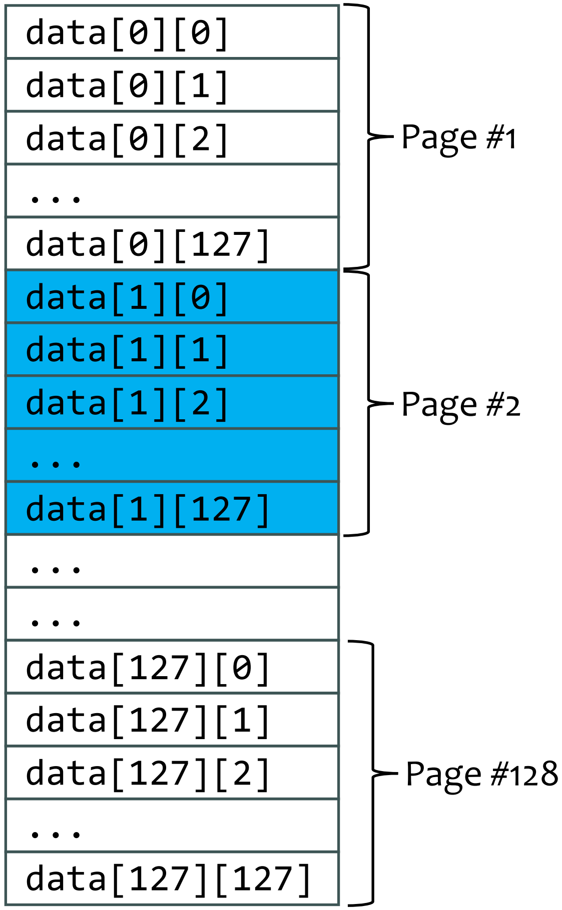
Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128 x 128 = 16384**



- An optimized (cache-friendly) version:

```
int i, j;
int [128][128] data;

for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **2**

Other Considerations

Program Structure

- Suppose a page size of 128 words (512 Bytes), # of frames: 127.
- Consider the following program snippet:

```
int i, j;
int [128][128] data;

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128 x 128 = 16384**

- An optimized (cache-friendly) version:

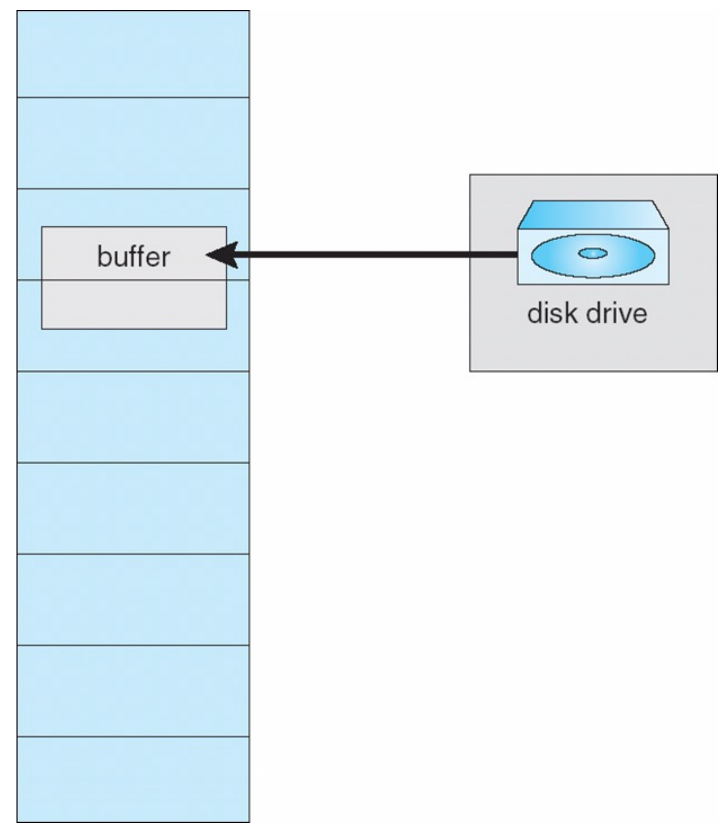
```
int i, j;
int [128][128] data;

for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
        data[i][j] = 0;
    }
}
```

# of Page Faults: **128**

data[0][0]	Page #1
data[0][1]	
data[0][2]	
...	
data[0][127]	
data[1][0]	Page #2
data[1][1]	
data[1][2]	
...	
data[1][127]	
...	
...	
data[127][0]	Page #128
data[127][1]	
data[127][2]	
...	
data[127][127]	

Other Considerations

- **I/O Interlock and Page Locking**
  - Pages must sometimes be locked into memory.
  - Consider I/O – Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.
  - Pinning of pages to lock into memory.



Thank you!