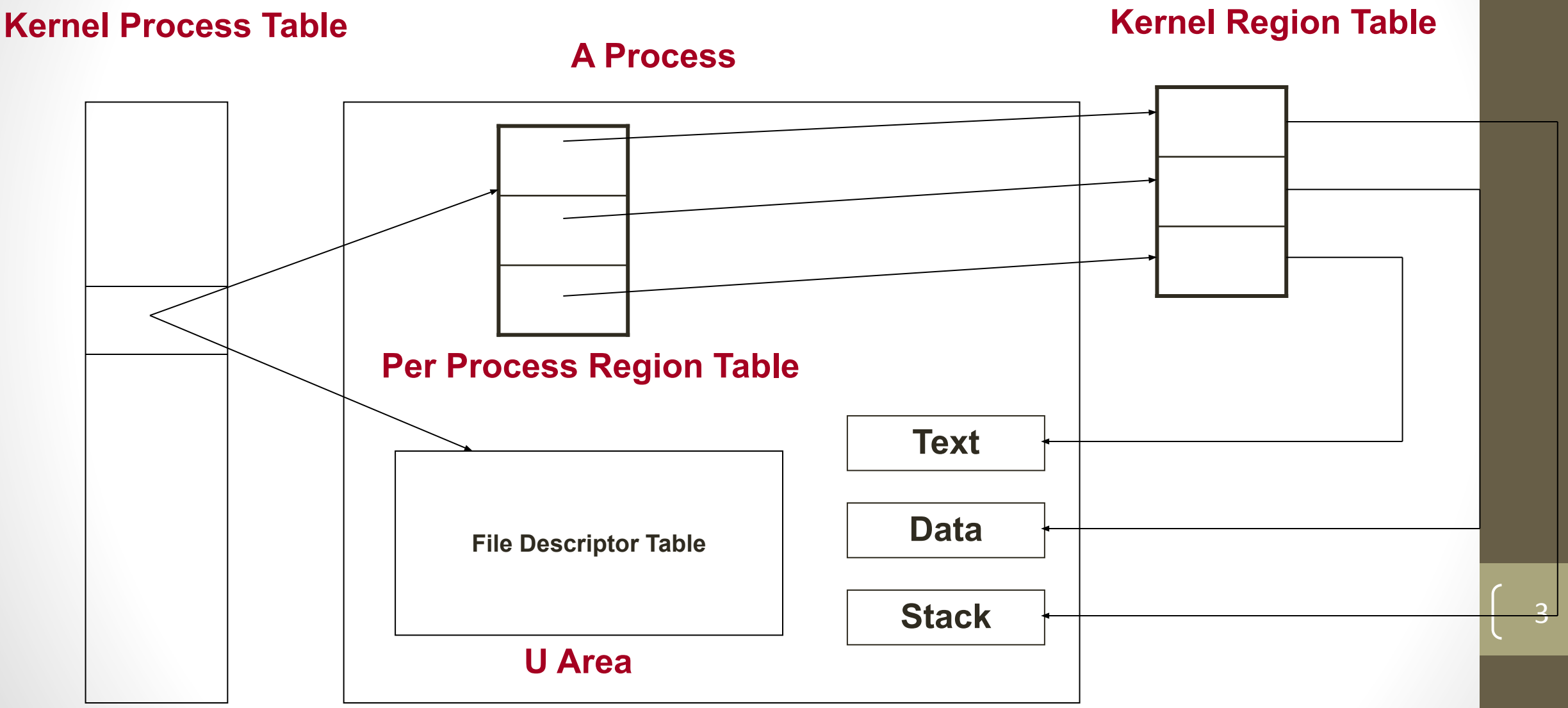
# Memory Management Policies

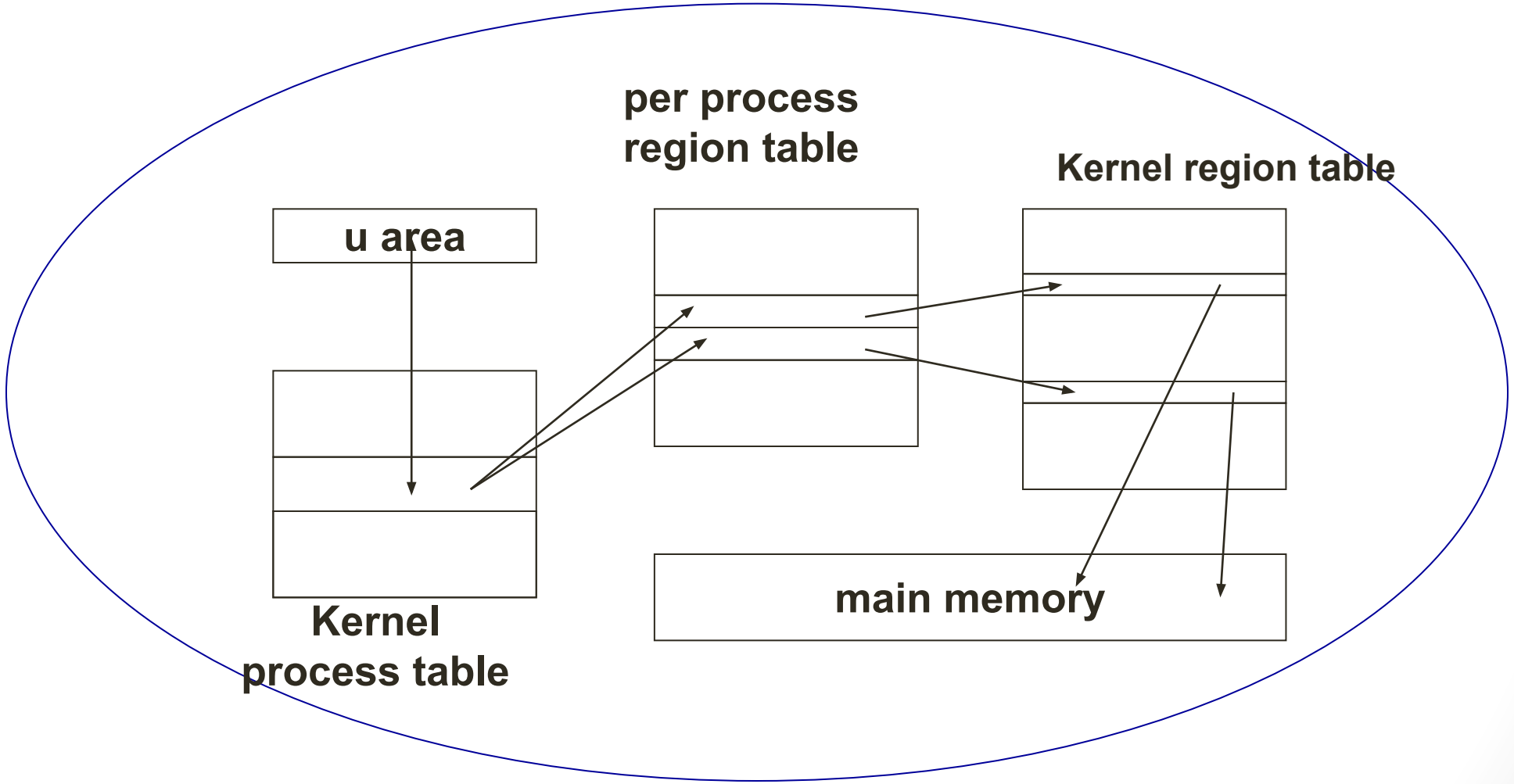
# Memory

- Primary memory is a precious resource that frequently cannot contain all active processes in the system
- At least part of a process must be contained in primary memory to run; the CPU cannot execute a process that exists entirely in secondary memory.
- The memory management system decides which processes should reside (at least partially) in main memory
- It monitors the amount of available primary memory and may periodically write processes to a secondary device called the swap device to provide more space in primary memory
- At a later time, the kernel reads the data from swap device back to main memory

# Data Structures for Process



# Data Structure for Process (contd.)



# UNIX Memory Management Policies

- **Swapping**
  - Easy to implement
  - Less system overhead
- **Demand Paging**
  - Greater flexibility

# 1. Swapping

- The swap device is a block device in a configurable section of a disk
- Kernel allocates space on swap device in contiguous blocks without fragmentation
- It maintains free space of the swap device in an in-core table, called map, that uses first-fit allocation of contiguous blocks of resource.
- The kernel treats each unit of the swap map as group of disk blocks
- As kernel allocates and frees resources, it updates the map accordingly

# Swapping

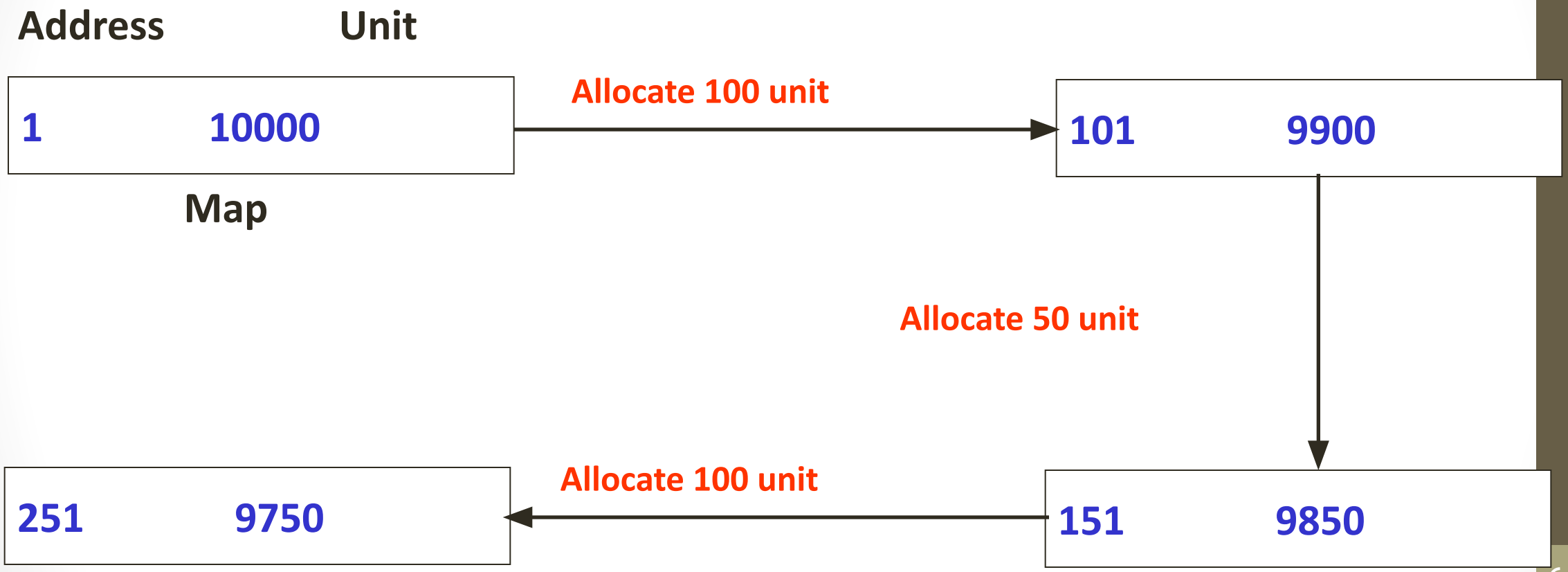
- A map is an array where each entry consists of an address of an allocatable resource and the number of resource units available there; the kernel interprets the address and units according to the type of map.
- Initially, a map contains one entry that indicates the address and the total number of resources.

| Address | Units |
|---|---|
| 1 | 10000 |

**Figure 9.1.** Initial Swap Map.

- As the kernel allocates and frees resources, it updates the map so that it continues to contain accurate information about free resources.

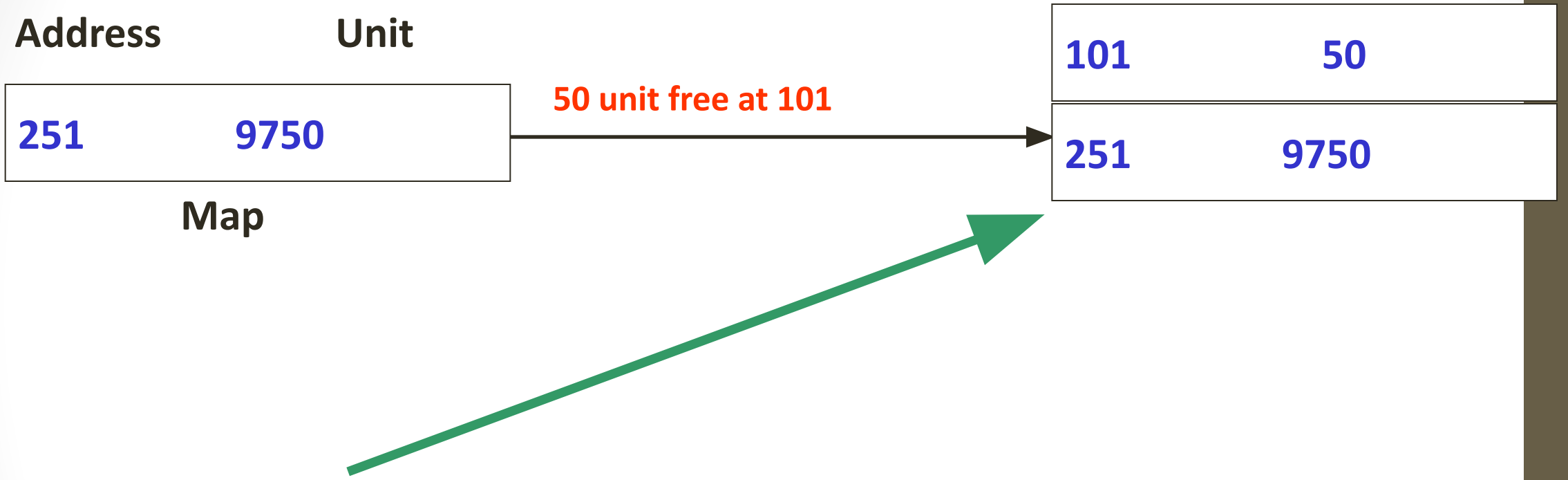
# Allocating Swap Space



# Algorithm to allocate swap space

```
algorithm malloc        /* algorithm to allocate map space */
input:   (1) map address        /* indicates which map to use */
         (2) requested number of units
output: address, if successful
         0, otherwise
{
        for (every map entry)
        {
                if (current map entry can fit requested units)
                {
                        if (requested units == number of units in entry)
                                delete entry from map;
                        else
                                adjust start address of entry;
                        return (original address of entry);
                }
        }
        return(0);
}
```

# Freeing Swap Space

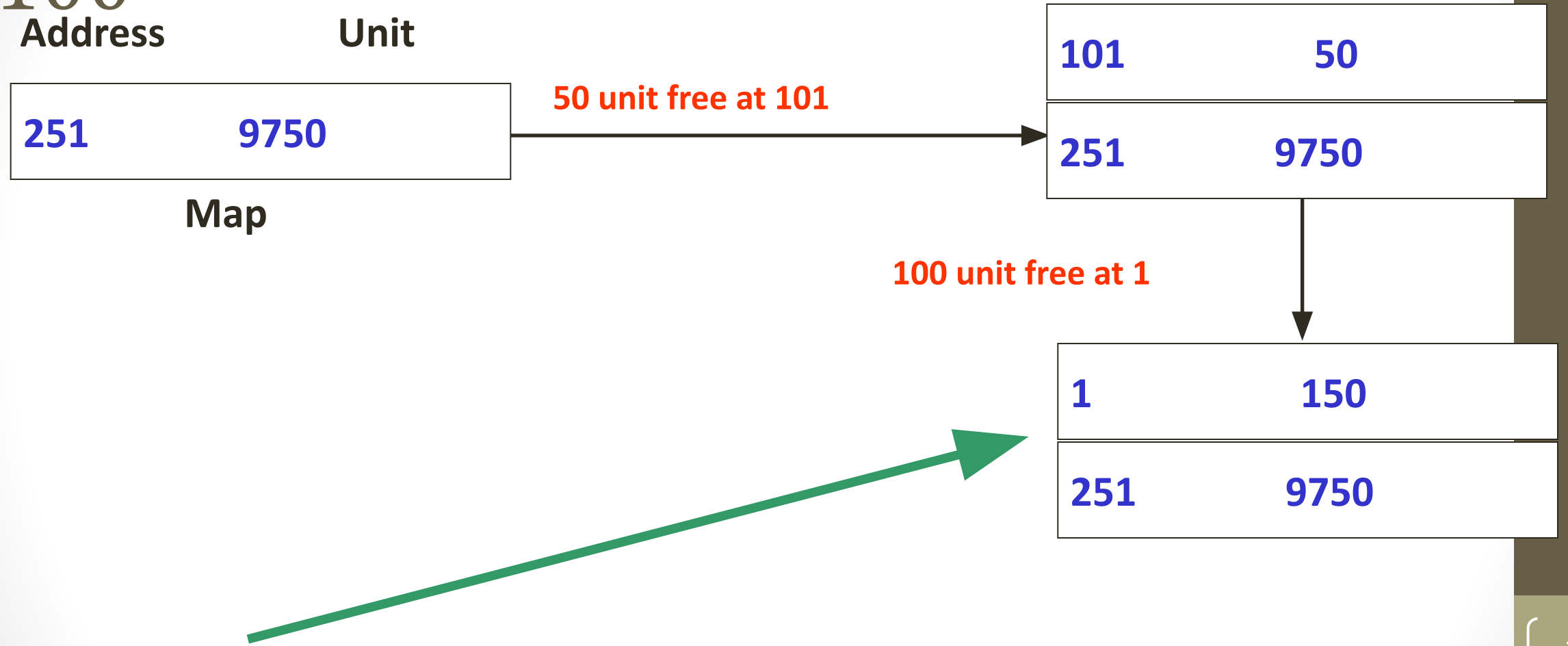



**Case 1:** Free resources fill a hole,

but not contiguous to any resources in the map

# Freeing Swap Space 1

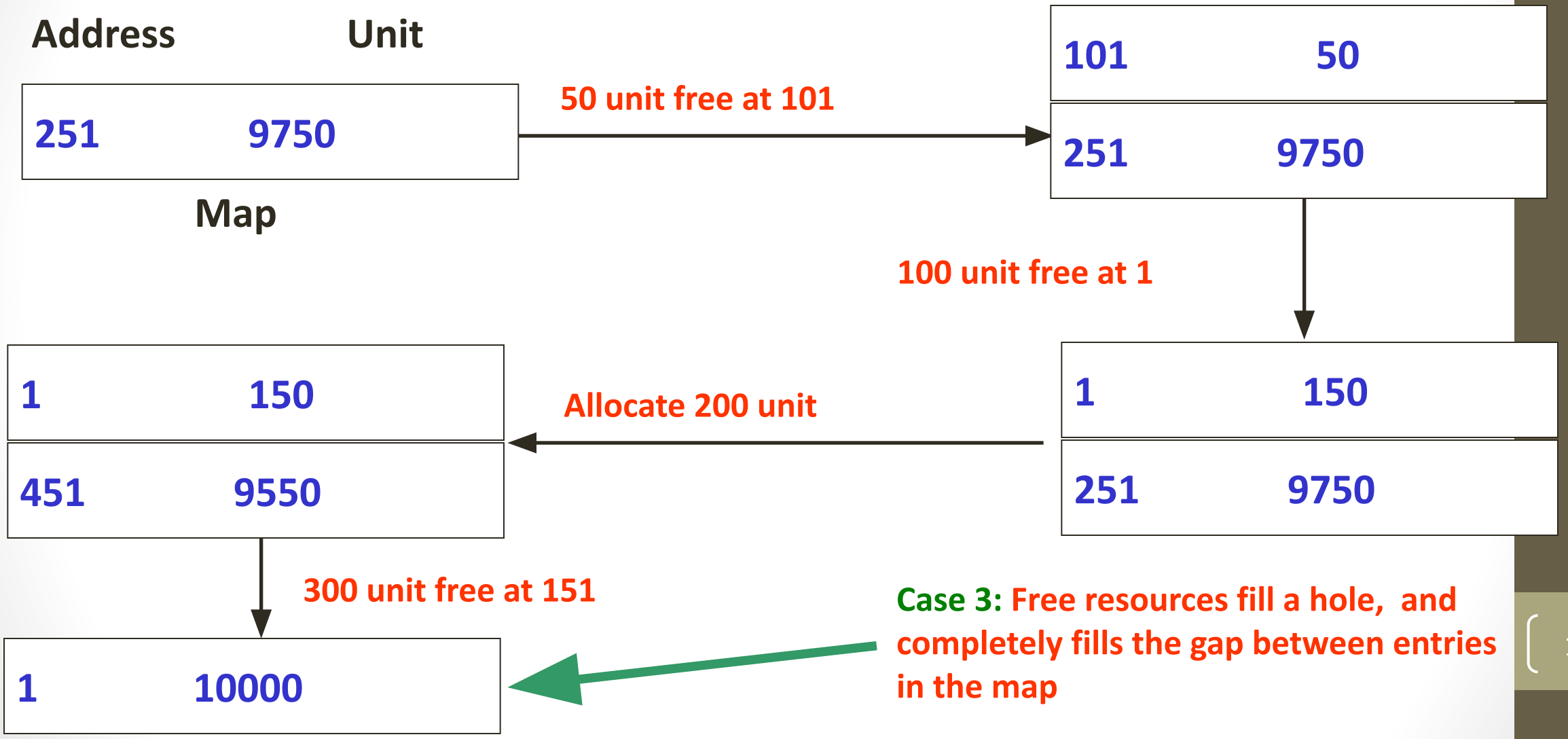
100




**Case 2:** Free resources fill a hole,

and immediately precedes an entry in the map

# Freeing Swap Space



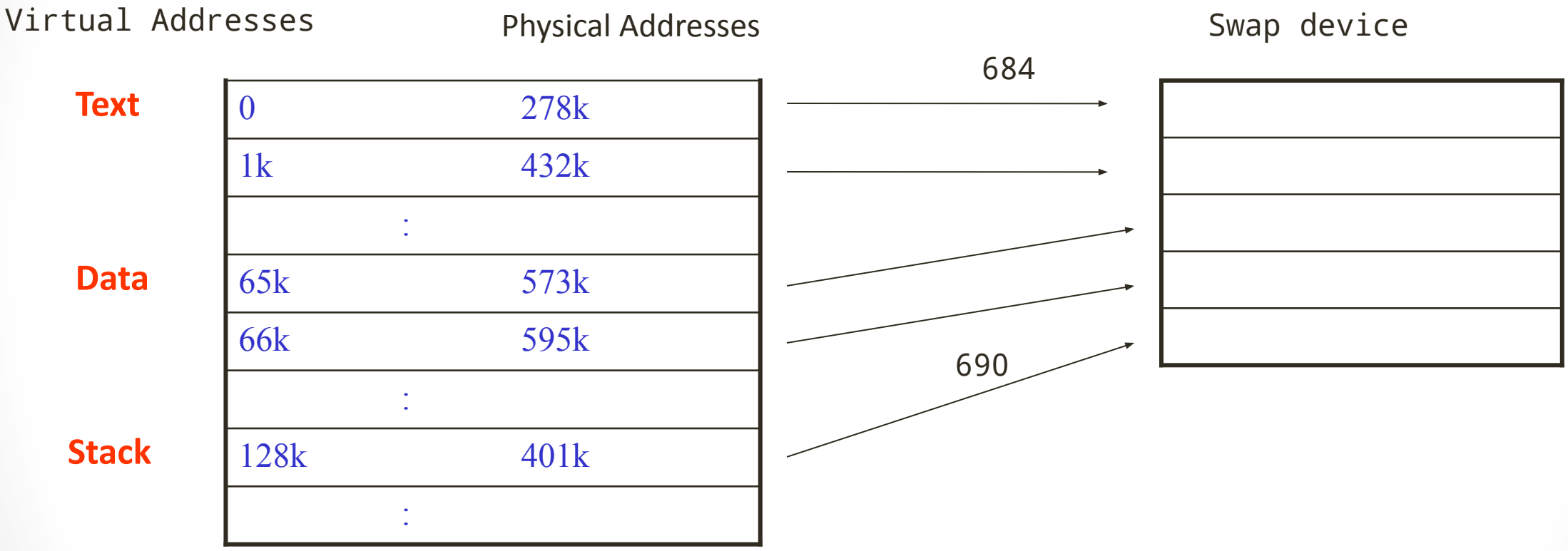
# Swapping Process Out

- Memory □ Swap device
- Kernel swaps process out when it needs memory, in the below cases
  1. When fork() called for allocating space to child process
  2. When brk() called for increasing the size of process
  3. When process become larger by growth of its stack
  4. Previously swapped out process want to swap in but not enough memory

# Swapping Process Out

- When the kernel decides that a process is eligible for swapping from main memory, it decrements the reference count of each region in the process and swaps the region out if its reference count drops to 0.
- The kernel allocates space on a  swap device and locks the process in memory (for cases 1 -3) , preventing the swapper from swapping it out, while the current swap operation is in progress.
- The kernel saves the swap address of the region in the region table entry.
- The kernel must gather the page addresses of data at primary memory to be swapped out
- Kernel copies the physical memory assigned to a process to the allocated space on the swap device
- The mapping between physical memory and swap device is kept in page table entry

# Swapping Process Out

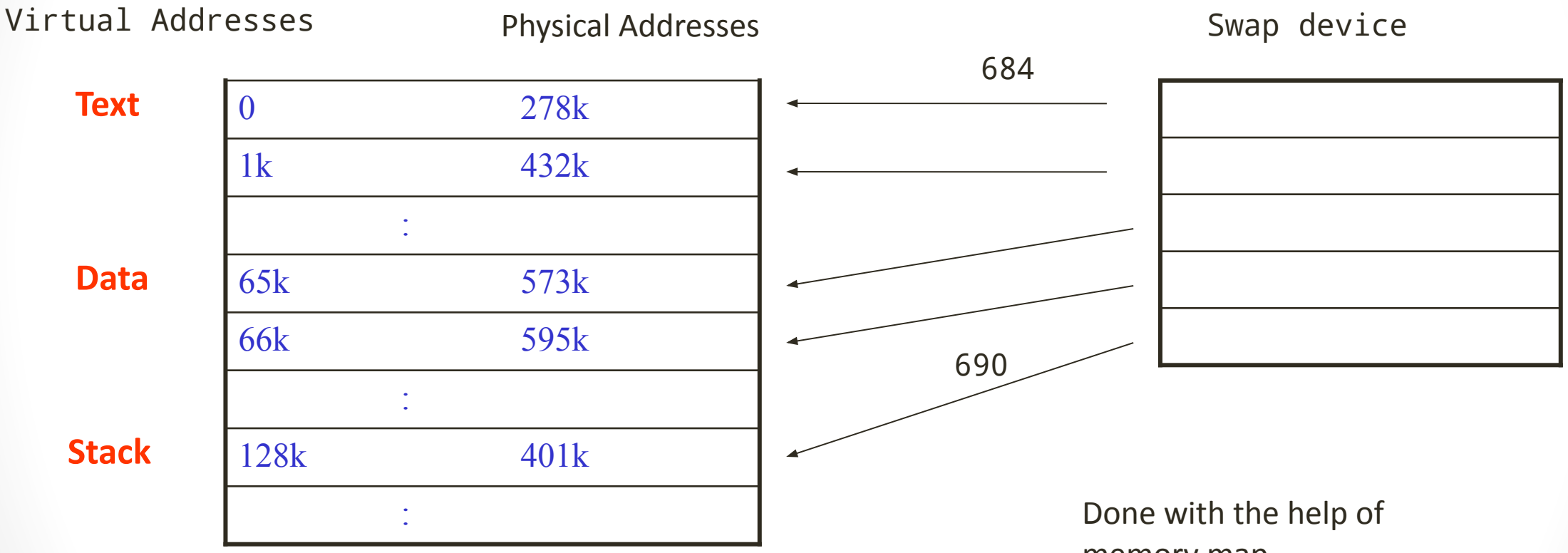



**Mapping process onto the swap device**

# Swapping Process In

- Process 0, the swapper, is the only process that swaps processes into memory from swap devices.
- At the conclusion of system initialization, the swapper goes into an infinite loop, where its only task is to do process swapping.
- It attempts to swap processes in from the swap device, and it swaps processes out if it needs space in main memory.
- The swapper sleeps if there is no work for it to do (for example, if there are no processes to swap in) or if it is unable to do any work (there are no processes eligible to swap out) ; the kernel periodically wakes it up.
- When the swapper wakes up to swap processes in, it examines all processes that are in the state "ready to run but swapped out" and selects one that has been swapped out the longest.
- If there is enough free memory available, the swapper swaps the process in, reversing the operation done for swapping out: It allocates physical memory, reads the process from the swap device, and frees the swap space.

# Swapping Process In




**Swapping a process into memory**

# Algorithm : swapping in and out process

```
algorithm swapper       /* swap in swapped out processes,
                         * swap out other processes to make room */
input:  none
output: none
{
    loop:
        for (all swapped out processes that are ready to run)
                pick process swapped out longest;
        if (no such process)
        {
                sleep (event must swap in);
                goto loop;
        }
        if (enough room in main memory for process)
        {
                swap process in;
                goto loop;
        }
```

# Algorithm : swapping in and out process

```
/* loop2: here in revised algorithm (see page 285) */
    for (all processes loaded in main memory, not zombie and not locked in memory)
    {
        if (there is a sleeping process)
            choose process such that priority + residence time
                is numerically highest;
        else /* no sleeping processes */
            choose process such that residence time + nice
                is numerically highest;
    }
    if (chosen process not sleeping or residency requirements not
                satisfied)
        sleep (event must swap process in);
    else
        swap out process;
    goto loop;      /* goto loop2 in revised algorithm */
}
```

# Fork Swap

- The description of the fork system call assumed that the parent process found enough memory to create the child context.
- Otherwise, the kernel swaps the process out without freeing the memory occupied by the in-core (parent) copy.
- When the swap is complete, the child process exists on the swap device; the parent places the child in the "ready-to-run" state and returns to user mode.
- Since the child is in the "ready-to-run" state. the swapper will eventually swap it into memory, where the kernel will schedule it; the child will complete its part of the fork system call and return to user mode.

# Expansion Swap

- If a process requires more physical memory than is currently allocated to it, either as a result of user stack growth or invocation of the brk system call and if it needs more memory than is currently available, the kernel does an expansion swap of the process.
- It reserves enough space on the swap device to contain the memory space of the process, including the newly requested space.
- Then, it adjusts the address translation mapping of the process to account for the new virtual memory but does not assign physical memory (since none was available).
- Finally, it swaps the process out in a normal swapping operation, zeroing out the newly allocated space on the swap device.
- When the kernel later swaps the process into memory, it will allocate physical memory according to the new (augmented size) address translation map.
- When the process resumes execution, it will have enough memory.

Figure 9.10 depicts five processes and the time they spend in memory or on the swap device as they go through a sequence of swapping operations. For simplicity, assume that all processes are CPU intensive and that they do not make any system calls; hence, a context switch happens only as a result of clock interrupts at 1-second intervals. The swapper runs at highest scheduling priority, so it always runs briefly at 1-second intervals if it has work to do. Further, assume that the processes are the same size and the system can contain at most two processes simultaneously in main memory. Initially, processes A and B are in main memory and the other processes are swapped out. The swapper cannot swap any processes during the first 2 seconds, because none have been in memory or on the swap device for 2 seconds (the residency requirement), but at the 2-second mark, it swaps out processes A and B and swaps in processes C and D. It attempts to swap in process E, too, but fails because there is no more room in main memory. At the 3 second mark, process E is eligible for swapping because it has been on the swap device for 3 seconds, but the swapper cannot swap processes out of main memory because their residency time is under 2 seconds. At the 4-second mark, the swapper swaps out processes C and D and swaps in processes E and A.

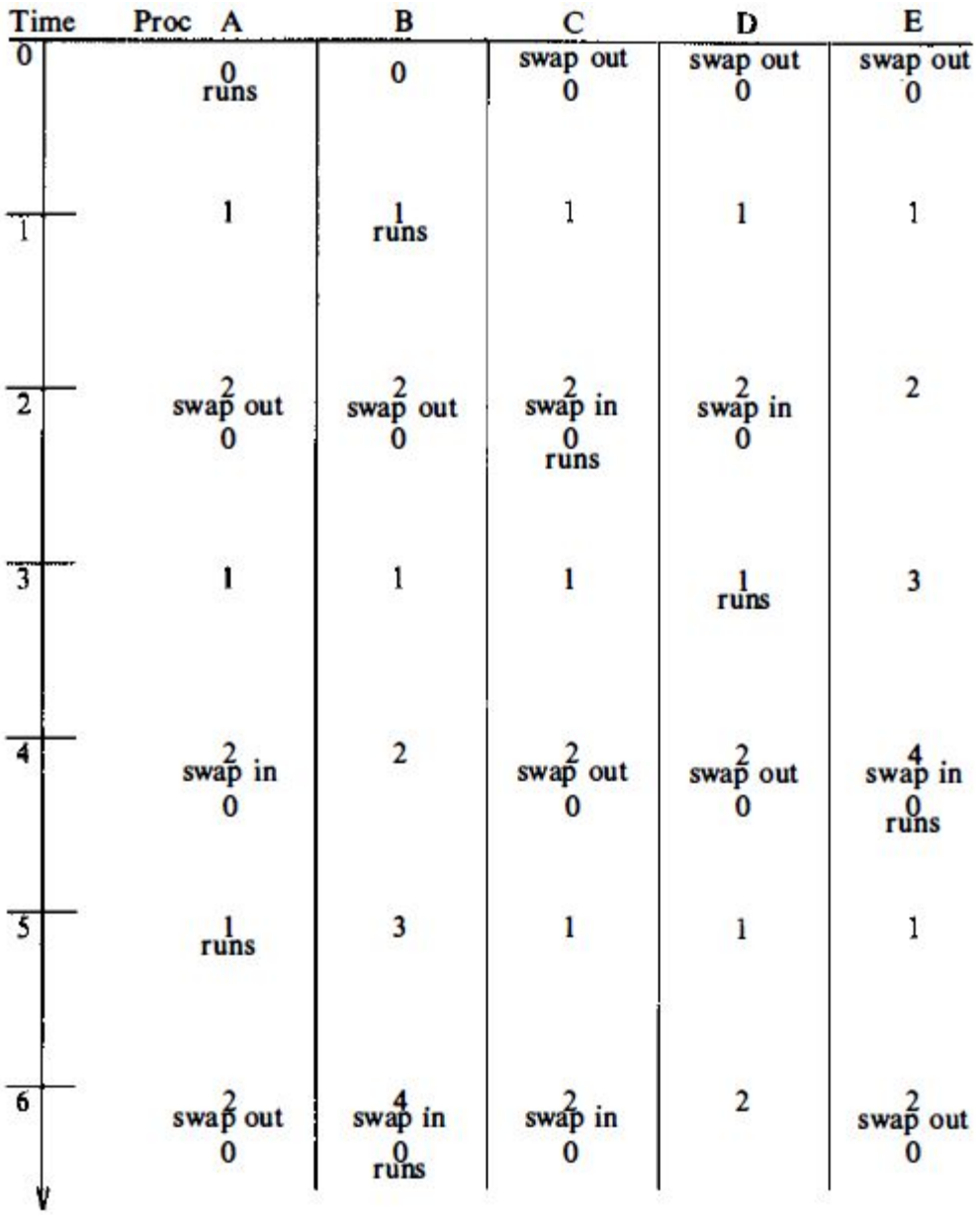



**Figure 9.10.** Sequence of Swapping Operations

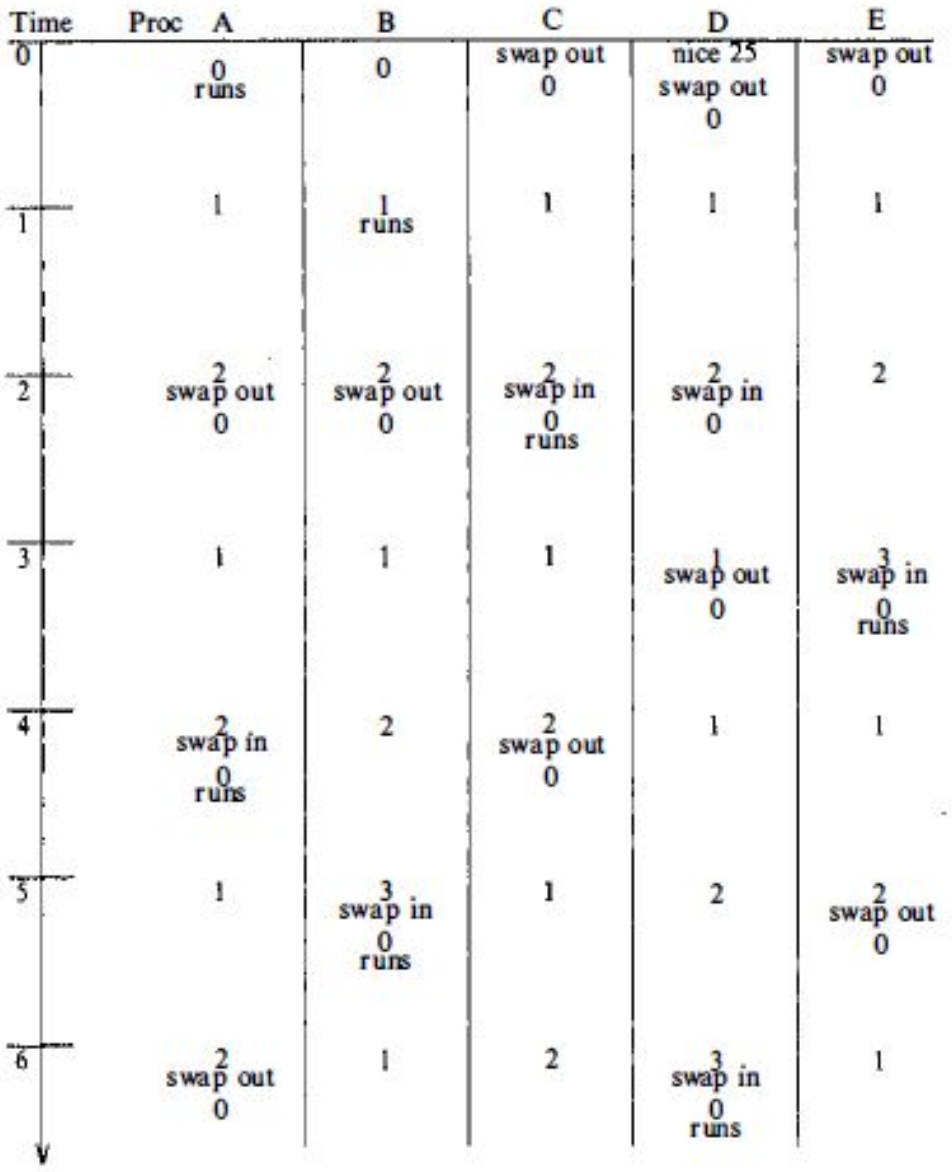



**Figure 9.11.** Thrashing due to Swapping

# 2. Demand Paging

- Machines whose memory architecture is based on pages and whose CPU has restartable instructions can support a kernel that implements a demand paging algorithm, swapping pages of memory between main memory and a swap device.
- Demand paging systems free processes from size limitations otherwise imposed by the amount of physical memory available on a machine.
- Since a process may not fit into physical memory, the kernel must load its relevant portions into memory dynamically and execute it even though other parts are not loaded.
- Processes tend to execute instructions in small portions of their text space, such as program loops and frequently called subroutines, and their data references tend to cluster in small subsets of the total data space of the process. This is known as the **principle of "locality."**

# Demand Paging

- **Working set of a process** is the set of pages that the process has referenced in its last n memory references; the number n is called the **window** of the working set.
- Because the working set is a fraction of the entire process, more processes may fit simultaneously into main memory than in a swapping system, potentially increasing system throughput because of reduced swapping traffic.
- When a process addresses a page that is not in its working set, it incurs a page fault; in handling the fault, the kernel updates the working set, reading in pages from a secondary device if necessary.

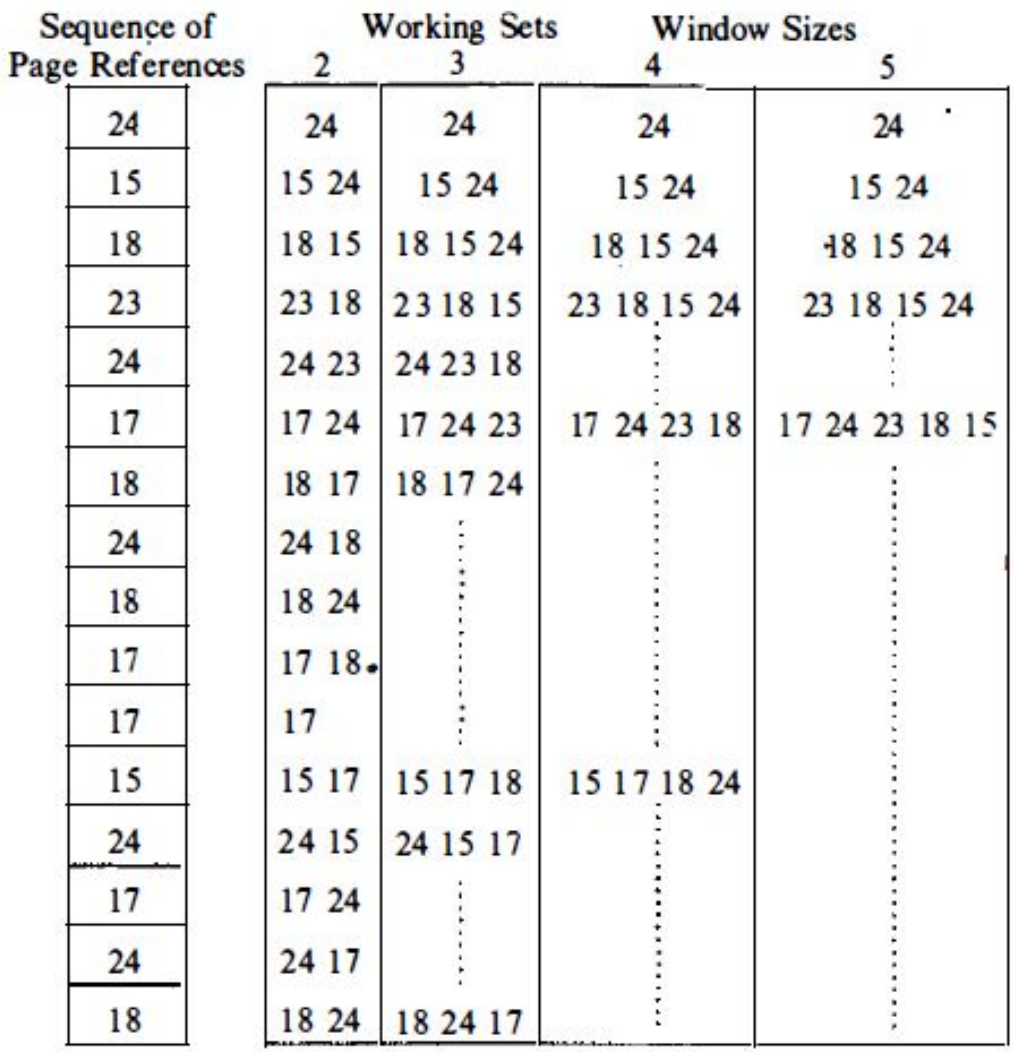



**Figure 9.12.** Working Set of a Process

# Demand Paging

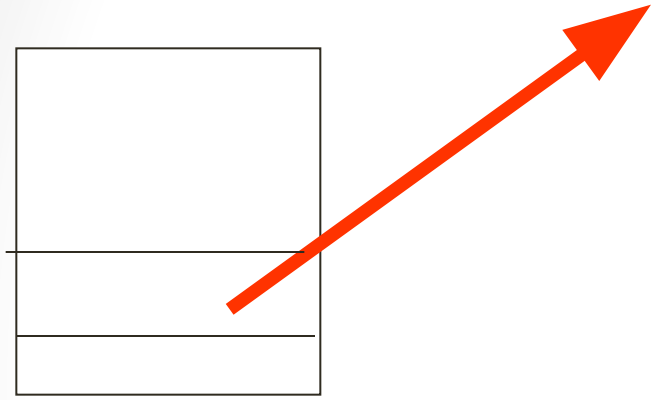
- Systems approximate a working set model by setting a **reference bit** whenever a process accesses a page and by sampling memory references periodically: If a page was **recently referenced, it is part of a working set; otherwise, it "ages"** in memory until it is eligible for swapping.
- When a process accesses a page that is not part of its working set, it incurs a **validity page fault** . The kernel suspends execution of the process until it reads the page into memory and makes it accessible to the process.
- When the page is loaded in memory, the process restarts the instruction it was executing when it incurred the fault.
- Thus, the implementation of a paging subsystem has two parts:
  - swapping rarely used pages to a swapping device and
  - handling page faults.

# Data Structure for Demand Paging

- Page table entry
- Disk block descriptors
- Page frame data table
- Swap use table

# Page Table Entry and Disk Block Descriptor

**Region**



**Page Table**

| Page Table Entry | Disk Block Descriptor |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

**Page Table Entry**

| Page address | age | Cp/wrt | mod | ref | val | prot |
|---|---|---|---|---|---|---|

**Disk Block Descriptor**

| Swap device | Block num | Type |
|---|---|---|

# Page Table Entry

- Contains the physical address of page and the following bits:
  - Valid: whether the page content legal
  - Reference: whether the page is referenced  recently
  - Modify: whether the page content is modified
  - copy on write: kernel must create a new copy when a process modifies its content (required for fork)
  - Age: Age of the page
  - Protection: Read/ write permission

# Disk Block Descriptor

- Swap Device number as there may be several swap devices
- Block number that contains page

| Swap device | Block num | Type |
|---|---|---|

# Page frame data table

- The pfdata table describes each page of physical memory and is indexed by page number. The fields of an entry are
  - The page state, indicating that the page is on a swap device or executable file, that DMA is currently underway for the page (reading data from a swap device) , or that the page can be reassigned.
  - The number of processes that reference the page. The reference count equals the number of valid page table entries that reference the page.
  - The logical device (swap or file system) and block number that contains a copy of the page.
  - Pointers to other pfdata table entries on a list of free pages and on a hash queue of pages.

# Swap-use table

- The swap-use table contains an entry for every page on a swap device.
- The entry consists of a reference count of how many page table entries point to a page on a swap device.

# Fork in a Paging System

- The kernel duplicates every region of the parent process during the fork system call and attaches it to the child process.
- Kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries:
  - It simply increments the region reference count of shared regions.
- For private regions such as data and stack, however, it allocates a new region table entry and page table.
- Kernel then examines each parent page table entry:
  - If a page is valid, it increments the reference count in the pfdata table entry, indicating the number of processes that share the page via different regions (as opposed to the number that share the page by sharing the region) .
  - If the page exists on a swap device, it increments the swap-use table reference count for the page.

# Fork in a Paging System

- The page can now be referenced through both regions, which share the page until a process writes to it.
- The kernel then copies the page so that each region has a private version.
- To do this, the kernel turns on the "copy on write" bit for every page table entry in private regions of the parent and child processes during fork .
- If either process writes the page, it incurs a **protection fault**, and in handling the fault, the kernel makes a new copy of the page for the faulting process.

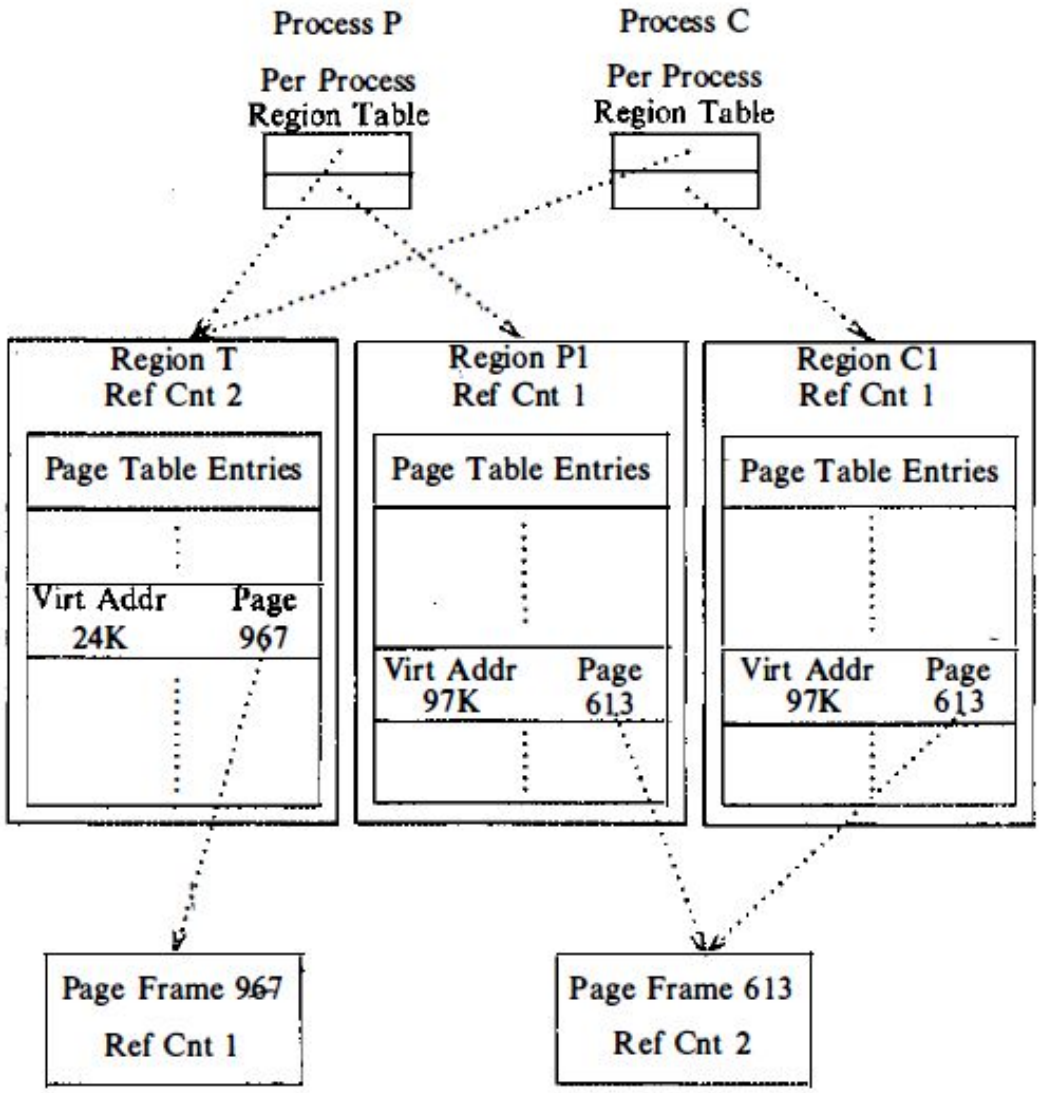



**Figure 9.15.** A Page in a Process that Forks

# Exec in a Paging System

- When a process invokes the exec system call, the kernel reads the executable file into memory from the file system.
- On a demand paged system, however, the executable file may be too large to fit in the available main memory.
- The kernel, therefore, does not preassign memory to the executable file but "faults" it in, assigning memory as needed.
- There are obvious inefficiencies in this scheme.
  - First, a process incurs a page fault when reading each page of the executable file;, even though it may never access the page.
  - Second, the page stealer may swap pages from memory before the exec is done, resulting in two extra swap operations per page if the process needs the page early.
- To make exec more efficient, the kernel can demand page directly from the executable file. Kernel finds all disk block numbers of the executable file during exec and attaches the list to the files inode.
- The validity fault handler later uses this information to load the page into main memory.

# The Page-Stealer Process

- The page stealer is a kernel process that swaps out memory pages that are no longer part of the working set of a process.
- The kernel creates the page stealer during system initialization and invokes it throughout the lifetime of the system when low on free pages.
- It examines every active, unlocked region, skipping locked regions in the expectation of examining them during its next pass through the region list, and increments the age field of all valid pages.
- **The kernel locks a region when a process faults on page in the region, so that the page stealer cannot steal the page being faulted in.**

# The Page-Stealer Process

- There are two paging states for a page in memory:
  - The page is aging and is not yet eligible for swapping, or
  - the page is eligible for swapping and is available for reassignment to other virtual pages.
- During each examination of the pages by the page stealer the age field is incremented by the page stealer.
- When the number of age exceeds a threshold value, the kernel puts the page into the second state, ready to be swapped.
- If a process references the aging page, the age field is again modified to 0, since it was referenced and becomes part of working set.
- The kernel wakes up the page stealer when the available free memory in the system is below a **low-water mark**, and the page stealer swaps out pages until the available free memory in the system exceeds a **high-water mark**.
- The use of high and low-water marks **reduces thrashing**, so the page stealer does not run as often.

# The Page-Stealer Process

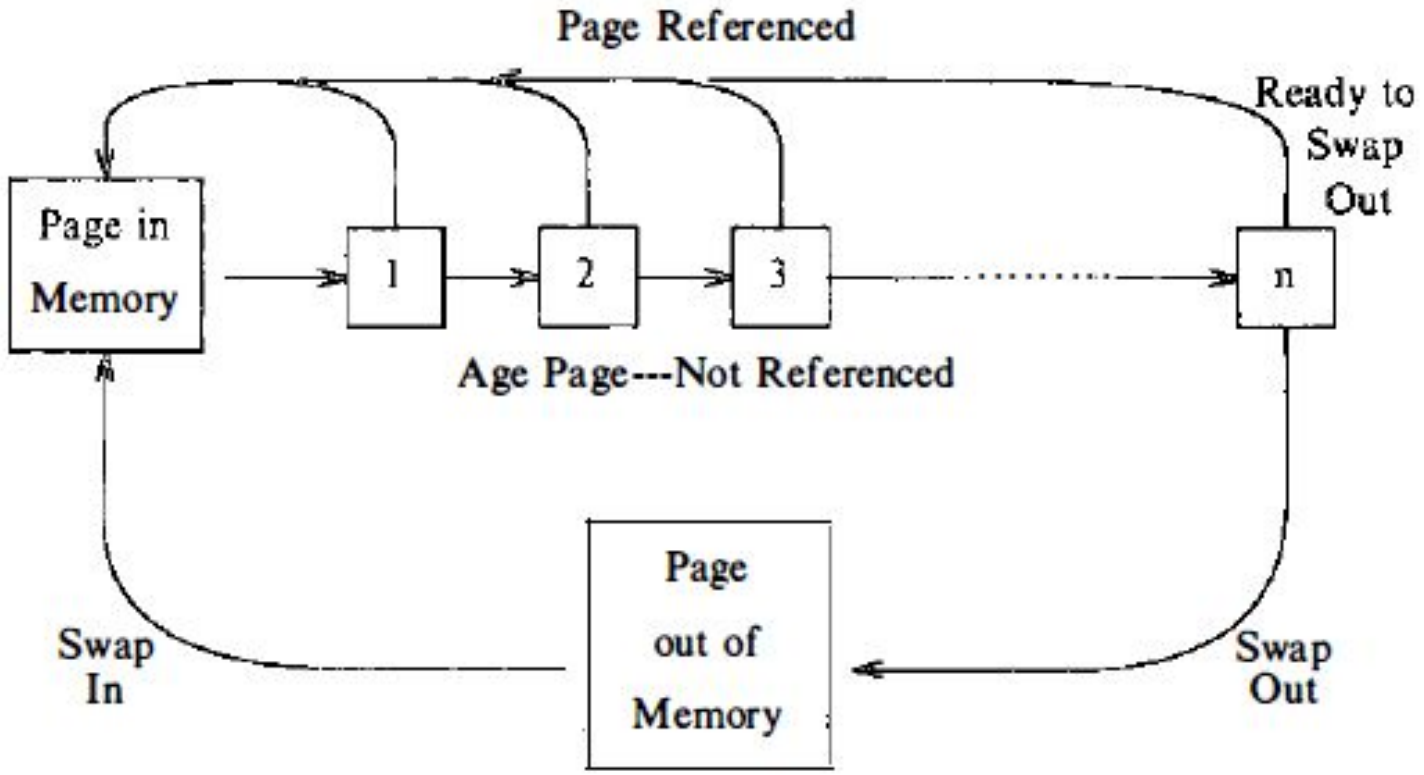



**Figure 9.18. State Diagram for Page Aging**

# The Page-Stealer Process

- When the page stealer decides to swap out a page, it considers whether a copy of the page is on a swap device. There are three possibilities.

1. If no copy of the page is on a swap device, the kernel "schedules" the page for swapping:
   - The page stealer places the page on a list of pages to be swapped out and continues; the swap is logically complete.
   - When the list of pages to be swapped reaches a limit, the kernel writes the pages to the- swap device.
2. If a copy of the page is already on a swap device and no process had modified its in-core contents (the page table entry modify bit is clear) , the kernel clears the page table entry valid bit, decrements the reference count in the pfdata table entry, and puts the entry on the free list for future allocation.
3. If a copy of the page is on a swap device but a process had modified its contents in memory, the kernel schedules the page for swapping, and frees the space it currently occupies on the swap device.

# Page Faults

- The system can incur two types of page faults:
  - validity faults and
  - protection faults.

# Validity Fault Handler

- If a process attempts to access a page whose valid bit is not set, it incurs a validity fault and the kernel invokes the validity fault handler.
- The hardware supplies the kernel with the virtual address that was accessed to cause the memory fault, and the kernel finds the page table entry and disk block descriptor for ·the page.
- The kernel locks the region containing the page table entry to prevent race conditions that would occur if the page stealer attempted to swap the page out.
- If the disk block descriptor has no record of the faulted page, the attempted memory reference is **invalid** and the kernel sends a "segmentation violation" signal to the offending process.
- If the memory reference was **legal**, the kernel allocates a page of memory to read in the page contents from the swap device or from the executable file.

# Validity Fault Handler

The page that caused the fault is in one of five states:

- On a swap device and not in memory,
- On the free page list in memory,
- In an executable file,
- Marked "demand zero,“
- Marked "demand fill."

# Validity Fault Handler

- If a page is on a swap device and not in memory (case 1 ) , it once resided in main memory but the page stealer had swapped it out.
- From the disk block descriptor, the kernel finds the swap device and block number where the page is stored and verifies that the page is not in the page cache.
- The kernel updates the page table entry so that it points to the page about to be read in, places the pfdata table entry on a hash list to speed later operation of the fault handler, and reads the page from the swap device.
- The faulting process sleeps until the I/O completes.

# Validity Fault Handler

- It is possible that the kernel had never reassigned the physical page after swapping it out, or that another process had faulted the virtual page into another physical page (case 2) .
- In either case, the fault handler finds the page in the page cache, keying off the block number in the disk block descriptor.
- It reassigns the page table entry to point to the page just found, increments its page reference count, and removes the page from the free list.

# Validity Fault Handler

- If a copy of the page does not exist on a swap device but is in the original executable file (case 3) , the kernel reads the page from the original file.
- The fault handler examines the disk block descriptor, finds the logical block number in the file that contains the page, and finds the inode associated with the region table entry.
- It uses the logical block number as an offset into the array of disk block numbers attached to the inode during exec .
- Knowing the disk block number, it reads the page into memory

# Validity Fault Handler

- If a process incurs a page fault for a page marked "demand fill" or "demand zero" (cases 4 and 5) , the kernel allocates a free page in memory and updates the appropriate page table entry.
- For "demand zero," it also clears the page to zero.
- Finally, it clears the "demand fill" or "demand zero" flags

# Validity Fault Handler

```
algorithm vfault        /* handler for validity faults */
input:   address where process faulted
output: none
{
        find region, page table entry, disk block descriptor
                corresponding to faulted address, lock region;
        if (address outside virtual address space)
        {
                send signal (SIGSEGV: segmentation violation) to process;
                goto out;
        }
        if (address now valid)        /* process may have slept above */
                goto out;
        if (page in cache)
        {
                remove page from cache;
                adjust page table entry;
                while (page contents not valid)  /* another proc faulted first */
                        sleep (event contents become valid);
        }
```

# Validity Fault Handler

```
        else        /* page not in cache */
        {
                assign new page to region;

                put new page in cache, update pfdata entry;
                if (page not previously loaded and page "demand zero")
                        clear assigned page to 0;
                else
                {
                        read virtual page from swap dev or exec file;
                        sleep (event I/O done);
                }
                awaken processes (event page contents valid);
        }
        set page valid bit;
        clear page modify bit, page age;
        recalculate process priority;
  out:  unlock region;
}
```

# Validity Fault Handler

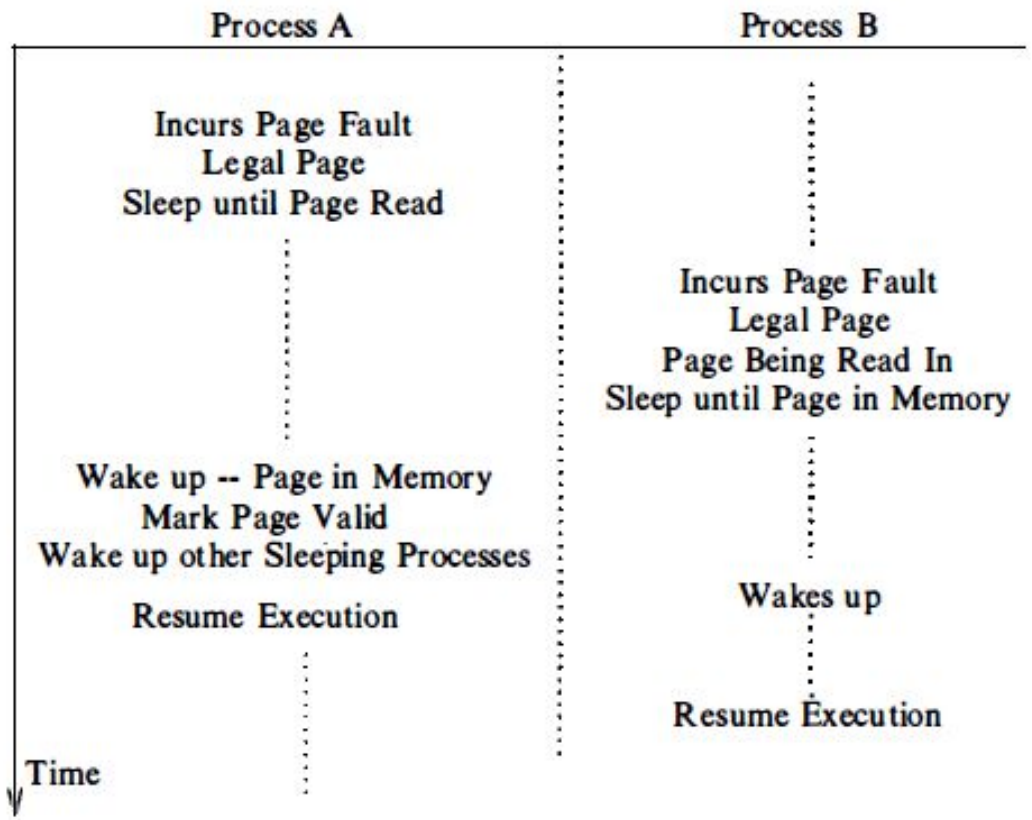



**Figure 9.24.** Double Fault on a Page

# Protection Fault Handler

- Protection fault can occur when
  - the process accessed a valid page but the permission bits associated with the page did not permit access
  - the process attempts to write a page whose copy on write bit was set during the fork system call.

# Protection Fault Handler

- The hardware supplies the protection fault handler with the virtual address where the fault occurred, and the fault handler finds the appropriate region and page table entry.
- It locks the region so that the page stealer cannot steal the page while the protection fault handler operates on it.
- If the fault handler determines that the fault was caused because the copy on write bit was set, and if the page is shared with other processes, the kernel allocates a new page and copies the contents of the old page to it; the other processes retain their references to the old page.
- After copying the page and updating the page table entry with the new page number, the kernel decrements the reference count of the old pfdata table entry.

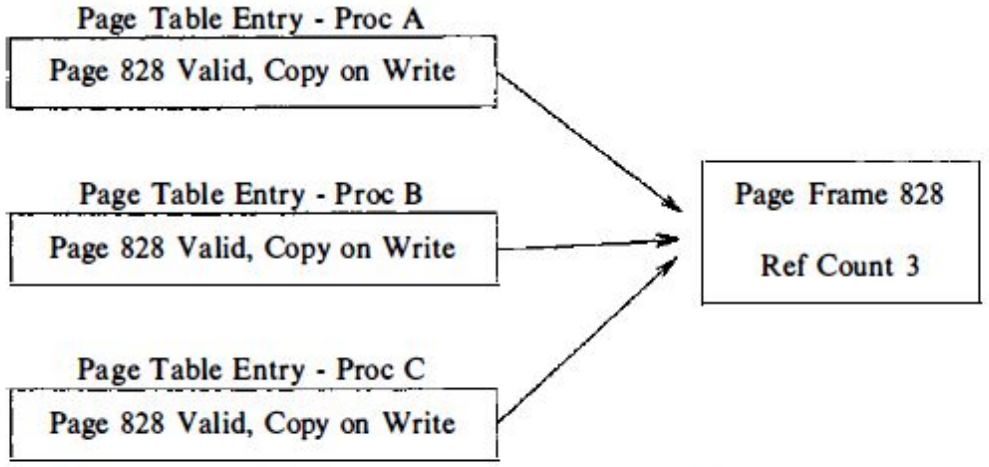
# Protection Fault Handler

```
algorithm pfault                /* protection fault handler */
input:      address where process faulted
output:      none
{
      find region, page table entry, disk block descriptor,
                  page frame for address, lock region;
      if (page not valid in memory)
            goto out;
      if (copy on write bit not set)
            goto out;                /* real program error - signal */
      if (page frame reference count > 1)
      {
          · allocate a new physical page;
            copy contents of old page to new page;
            decrement old page frame reference count;
            update page table entry to point to new physical page;
      }
```

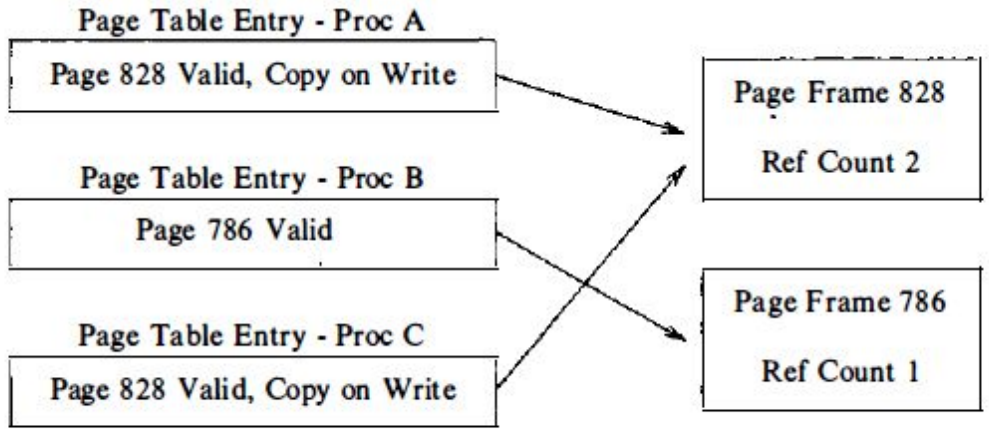
# Protection Fault Handler

```
        else        /* "steal" page, since nobody else is using it */
        {
                if (copy of page exists on swap device)
                        free space on swap device, break page association;
                if (page is on page hash queue)
                        remove from hash queue;
        }
        set modify bit, clear copy on write bit in page table entry;
        recalculate process priority;
        check for signals;
  out:  unlock region;
}
```

# Protection Fault Handler




(a) Before Proc B Incurs Protection Fault




(b) After Protection Fault Handler Runs for Proc B

**Figure 9.26.** Protection Fault with Copy on Write Set

# Protection Fault Handler

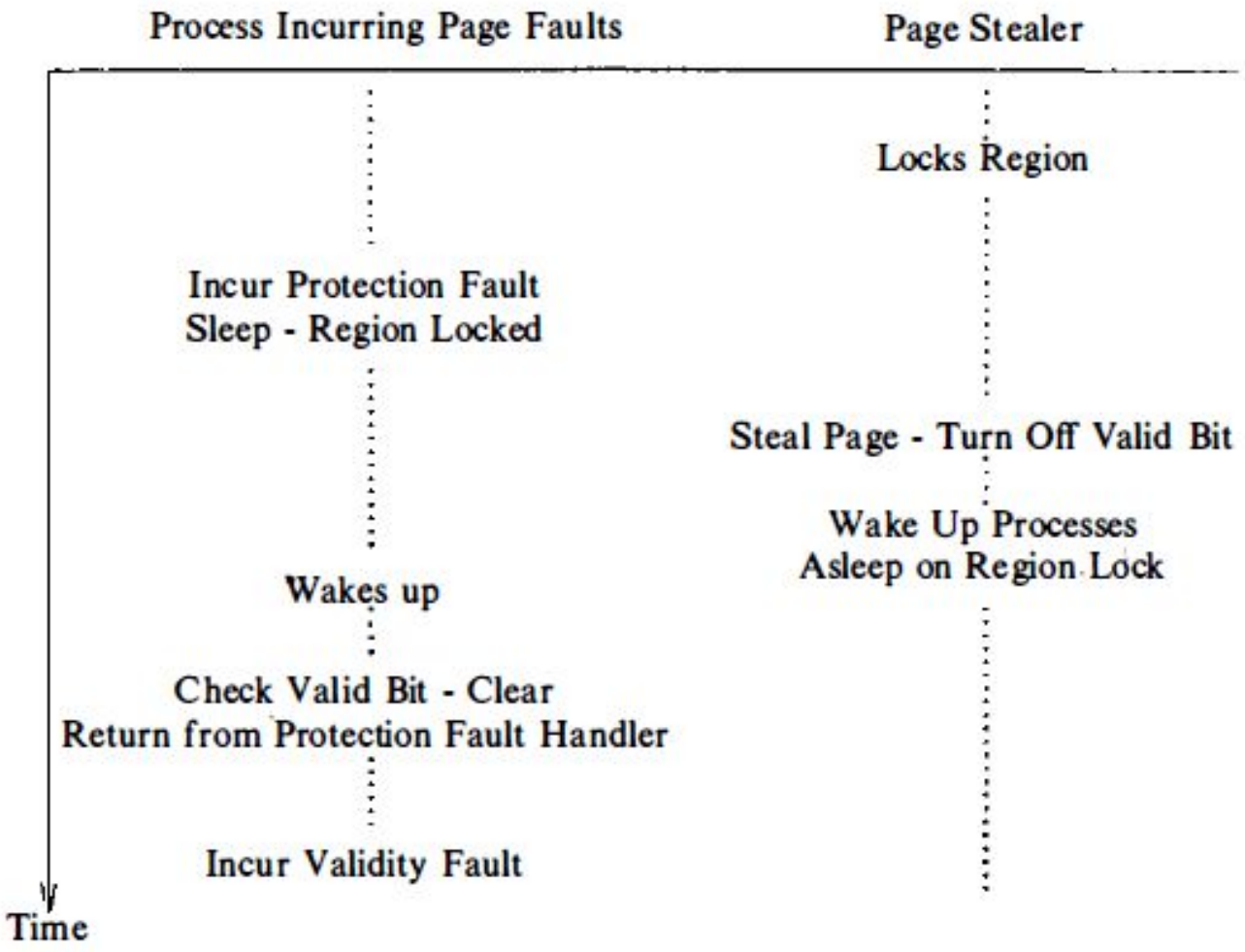



**Figure 9.27.** Interaction of Protection Fault and Validity Fault

# 3. A HYBRID SYSTEM WITH SWAPPING AND DEMAND PAGING

- The System V kernel runs swapping and demand paging algorithms to avoid thrashing problems.
- When the kernel cannot allocate pages for a process, it wakes up the swapper and puts the calling process into a state that is the equivalent of "ready to run but swapped".
- The swapper swaps out entire processes until available memory exceeds the high-water mark.
- For each process swapped out, it makes one "ready-to-run but swapped" process ready to run.
- It does not swap those processes in via the normal swapping algorithm but lets them fault in pages as needed.
- Later iterations of the swapper will allow other processes to be faulted in if there is sufficient memory in the system.
- This method slows down the system fault rate and reduces thrashing