Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

- **MMU : Hardware features for MM**
- **OS: Sets up MMU for a process, then schedules process**
- **Compiler : Generates object code for a particular OS + MMU architecture**
- **MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS**

More on Linking and Loading

- **Static Linking: All object code combined at link time and a big object code file is created**
- **Static Loading: All the code is loaded in memory at the time of exec()**
- **Problem: Big executable files, need to load functions even if they do not execute**
- **Solution: Dynamic Linking and Dynamic Loading**

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - **function code to function calls**
    - **references to global variables with “extern” declarations**
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

0000000000001189 <main>:

11d4: callq 1080 <printf@plt>

Disassembly of section .plt.got:

0000000000001080 <printf@plt>:

1080: endbr64

1084: bnd jmpq *0x2f3d(%rip) # 3fc8 <printf@GLIBC_2.2.5>

108b: nopl 0x0(%rax,%rax,1)

Dynamic Loading

- **Loader**
  - **Loads the program in memory**
  - **Part of exec() code**
  - **Needs to understand the format of the executable file (e.g. the ELF format)**
- **Dynamic Loading**
  - **Load a part from the ELF file only if needed during execution**
  - **Delayed loading**
  - **Needs a more sophisticated memory management by operating system – to be seen during this series of lectures**

Dynamic Linking, Loading

- **Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking**
  - **Hence called ‘link-loader’**
  - **Static or dynamic loading is still a choice**
- **Question: which of the MMU options will alllow for which type of linking, loading ?**

Continous memory management

What is Continous memory management?

- **Entire process is hosted as one continous chunk in RAM**
- **Memory is typically divided into two partitions**
  - **One for OS and other for processes**
  - **OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !**

Hardware support needed: base + limit (or relocation + limit)

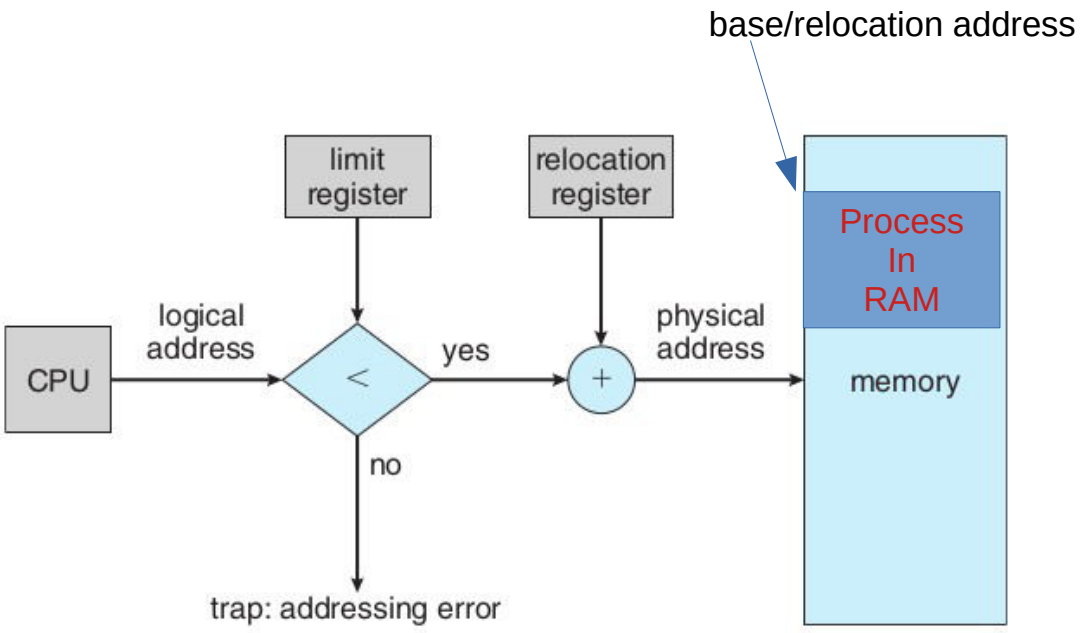


Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

Variable partition scheme

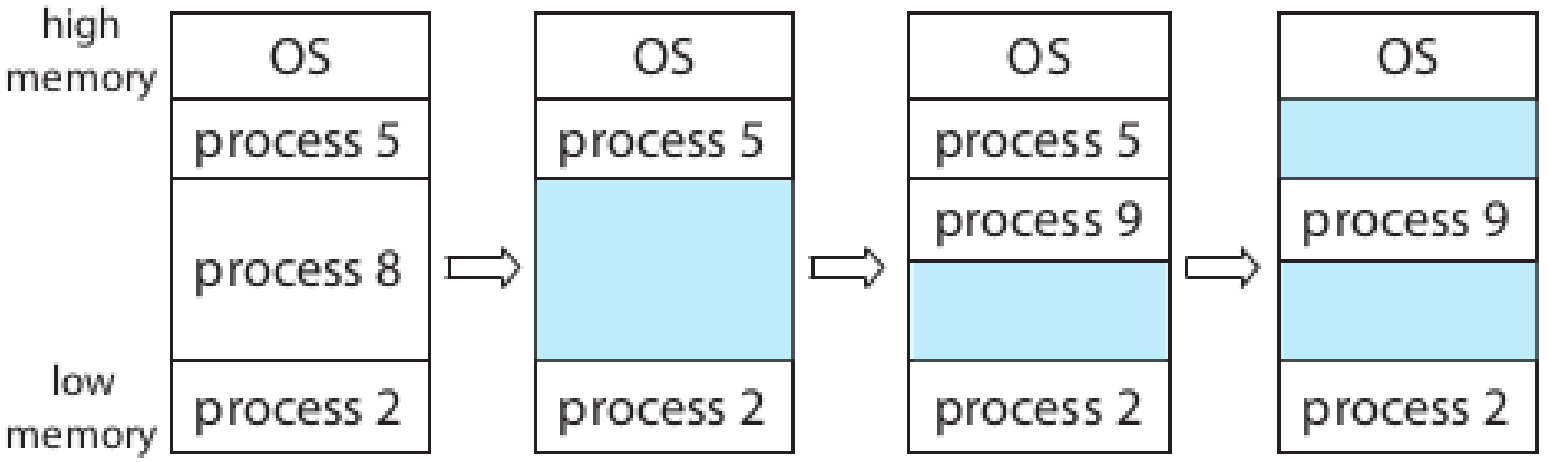


Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- **Suppose there are 3 free memory regions of sizes 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 15k**
- **Which region to allocate to it ?**

Strategies for finding a free chunk

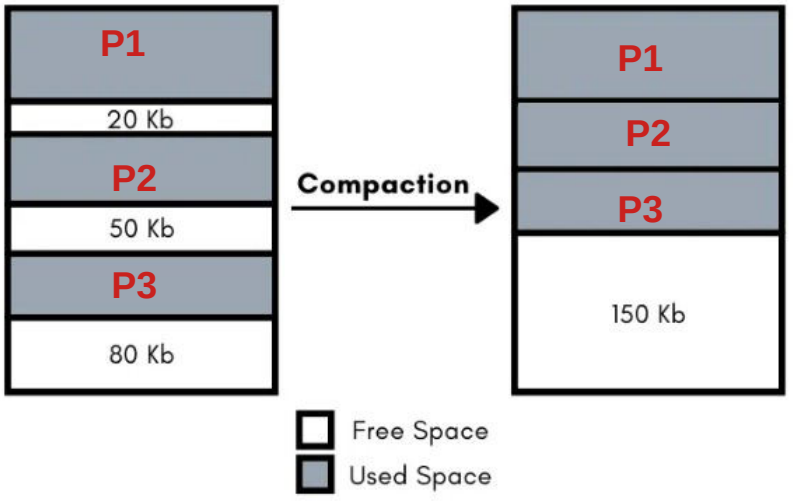
- **6k, 17k, 16k, 40k holes . Need 15k.**
- **Best fit: Find the smallest hole, larger than process. Ans: 16k**
- **Worst fit: Find the largest hole. Ans: 40k**
- **First fit: Find the “first” hole larger than the process. Ans: 17k**

Problem : External fragmentation

- **Free chunks: 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 50k**
- **Total free memory: 30+40+20 = 90k**
  - **But can’t allocate 50k !**

Solution to external fragmentation

- **Compaction !**
- **OS moves the process chunks in memory to make available continous memory region**
  - **Then it must update the memory management information in PCB (e.g. base of the process) of each process**
- **Time consuming**
- **Possible only if the relocation+limit scheme of MMU is available**



Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
  - **If total 1M memory, then 20 such chunks**
- **Allocate one or more chunks to a process, such that the total size is >= the size of the process**
  - **E.g.if request is 50k, allocate 1 chunk**
  - **If request is 40k, still allocate 1 chunk**
  - **If request is 60k, then allocate 2 chunks**
- **Leads to internal fragmentation**
  - **space wasted in the case of 40k or 60k requests above**

Fixed partition scheme

- **OS needs to keep track of**
  - **Which partition is free and which is used by which process**
  - **Free partitions can simply be tracked using a bitmap or a list of numbers**
  - **Each process's PCB will contain list of partitions allocated to it**

Solution to internal fragmentation

- **Reduce the size of the fixed sized partition**
- **How small then ?**
  - **Smaller partitions mean more overhead for the operating system in allocating deallocating**

Paging

An extended version of fixed size partitions

- **Partition = page**
  - **Process = logically continuous sequence of bytes, divided in ‘page’ sizes**
  - **Memory divided into equally sized page ‘frames’**
- **Important distinction**
  - **Process need not be continous in RAM**
  - **Different page sized chunks of process can go in any page frame**
  - **Page table to map pages into frames**

Logical address seen as

Paging hardware

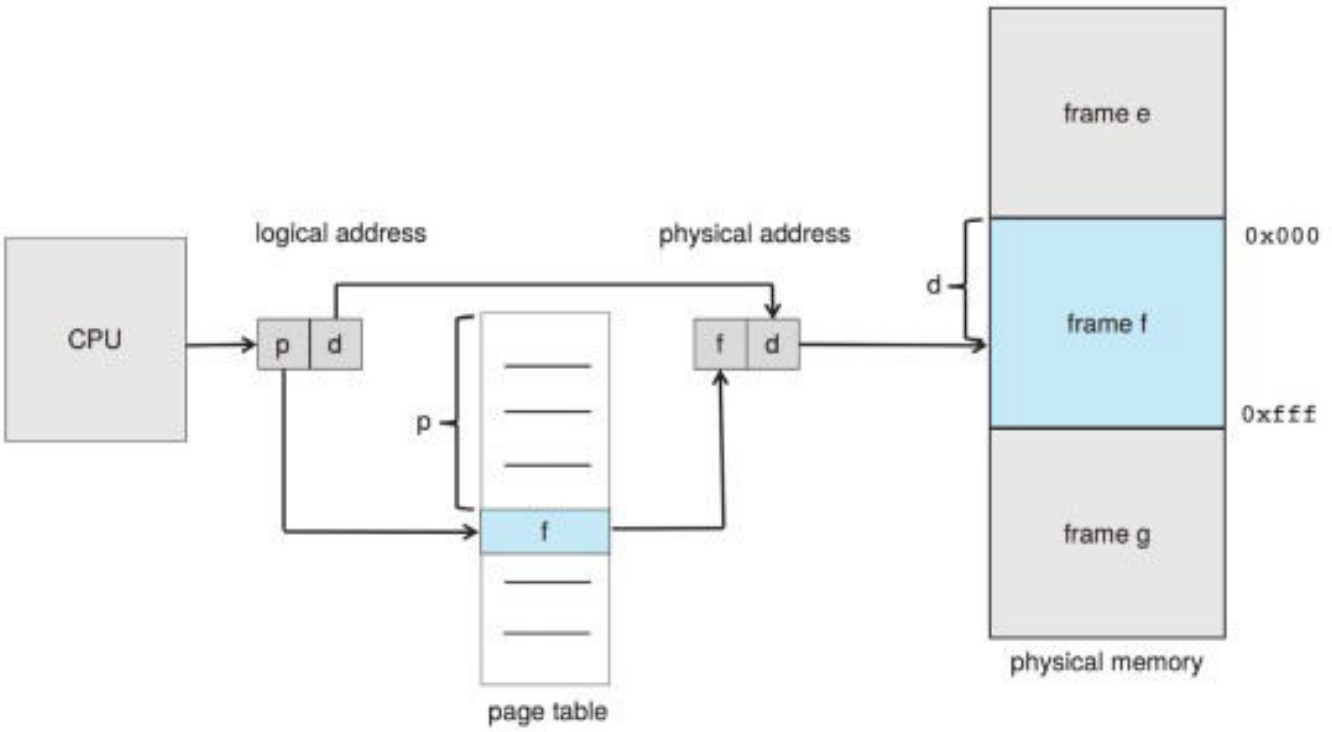


Figure 9.8 Paging hardware.

MMU’s job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- **Allocate a page table for the process, at time of fork()/exec()**
  - **Allocate frames to process**
  - **Fill in page table entries**
- **In PCB of each process, maintain**
  - **Page table location (address)**
  - **List of pages frames allocated to this process**
- **During context switch of the process, load the PTBR using the PCB**

Job of OS

- **Maintain a list of all page frames**
  - **Allocated frames**
  - **Free Frames (called frame table)**
  - **Can be done using simple linked list**
  - **Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)**
  -

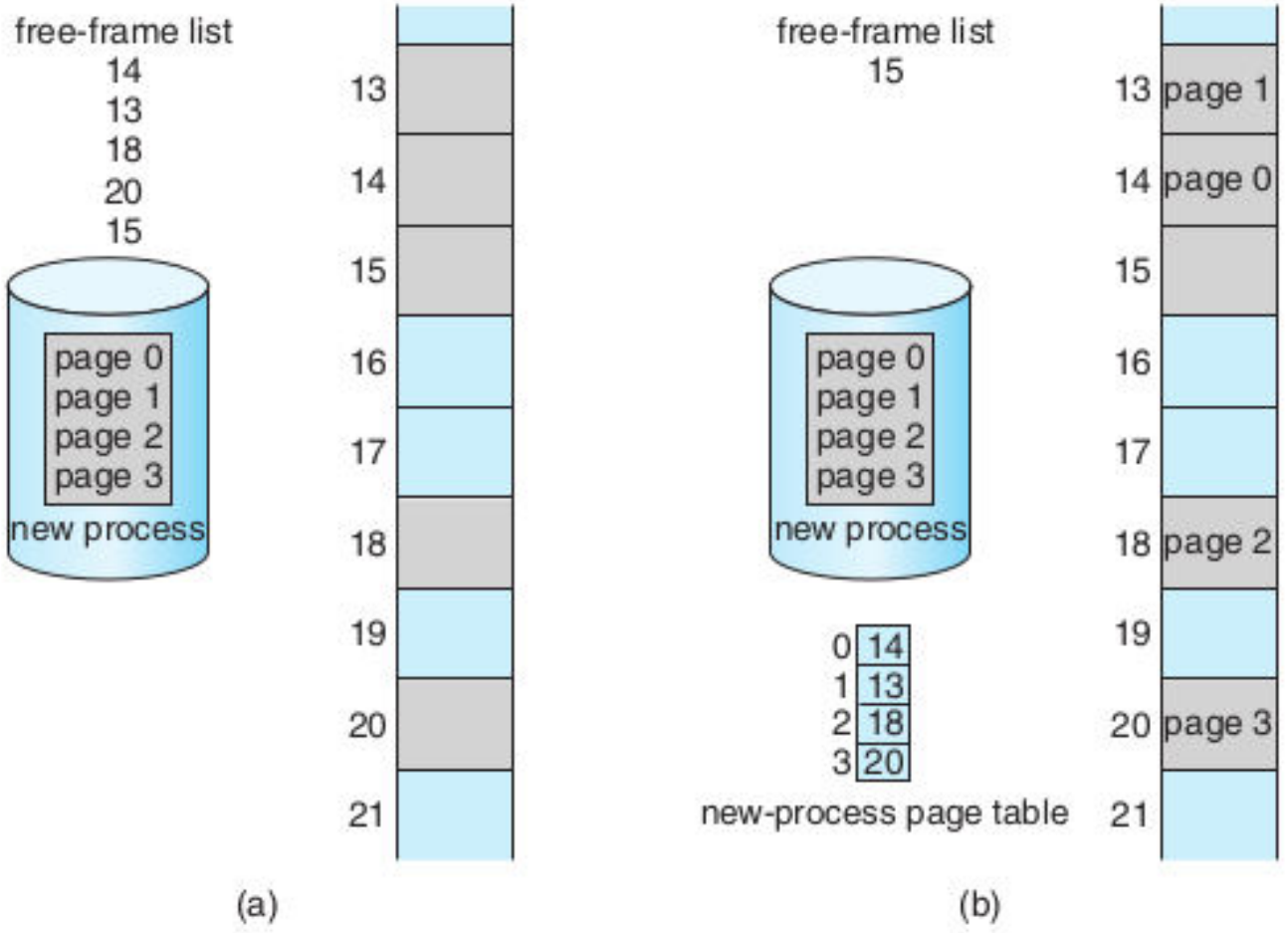


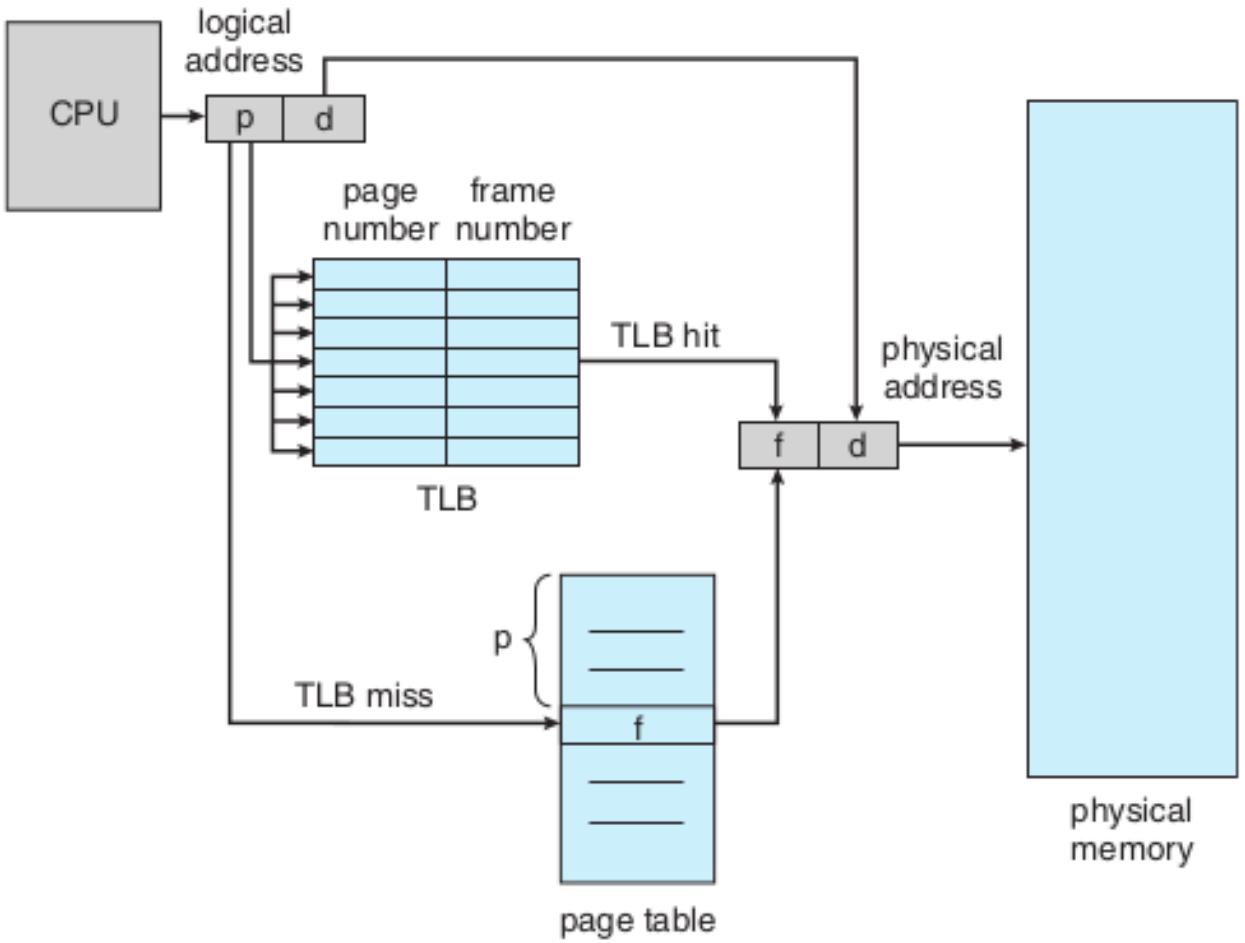
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- **Each memory access results in two memory accesses!**
  - **One for page table, and one for the actual memory location !**
  - **Done as part of execution of instruction in hardware (not by OS!)**
  - **Slow down by 50%**

Speeding up paging

- **Translation Lookaside Buffer (TLB)**
- **Part of CPU hardware**
- **A cache of Page table entries**
- **Searched in parallel for a page number**



Speedup due to TLB

- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time
- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

  = 12 nanoseconds

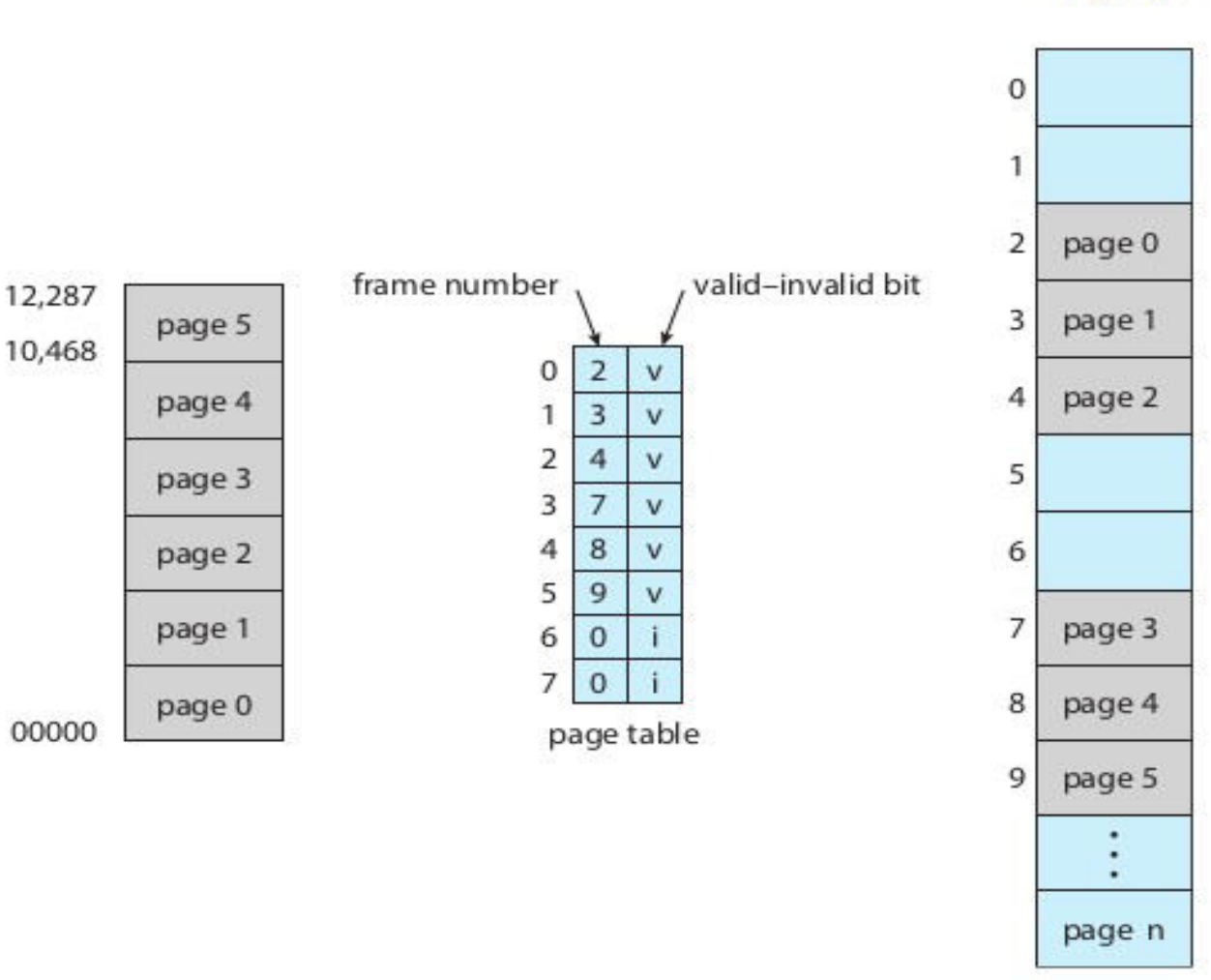
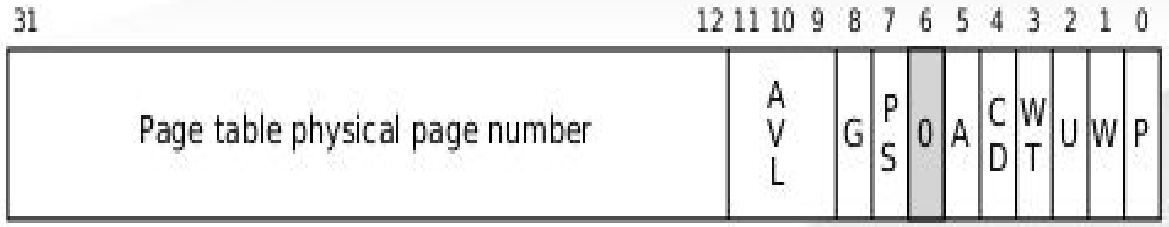


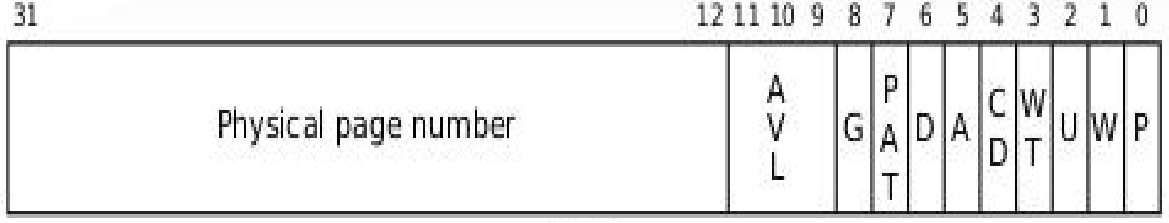
Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

X86 PDE and PTE



PDE



PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

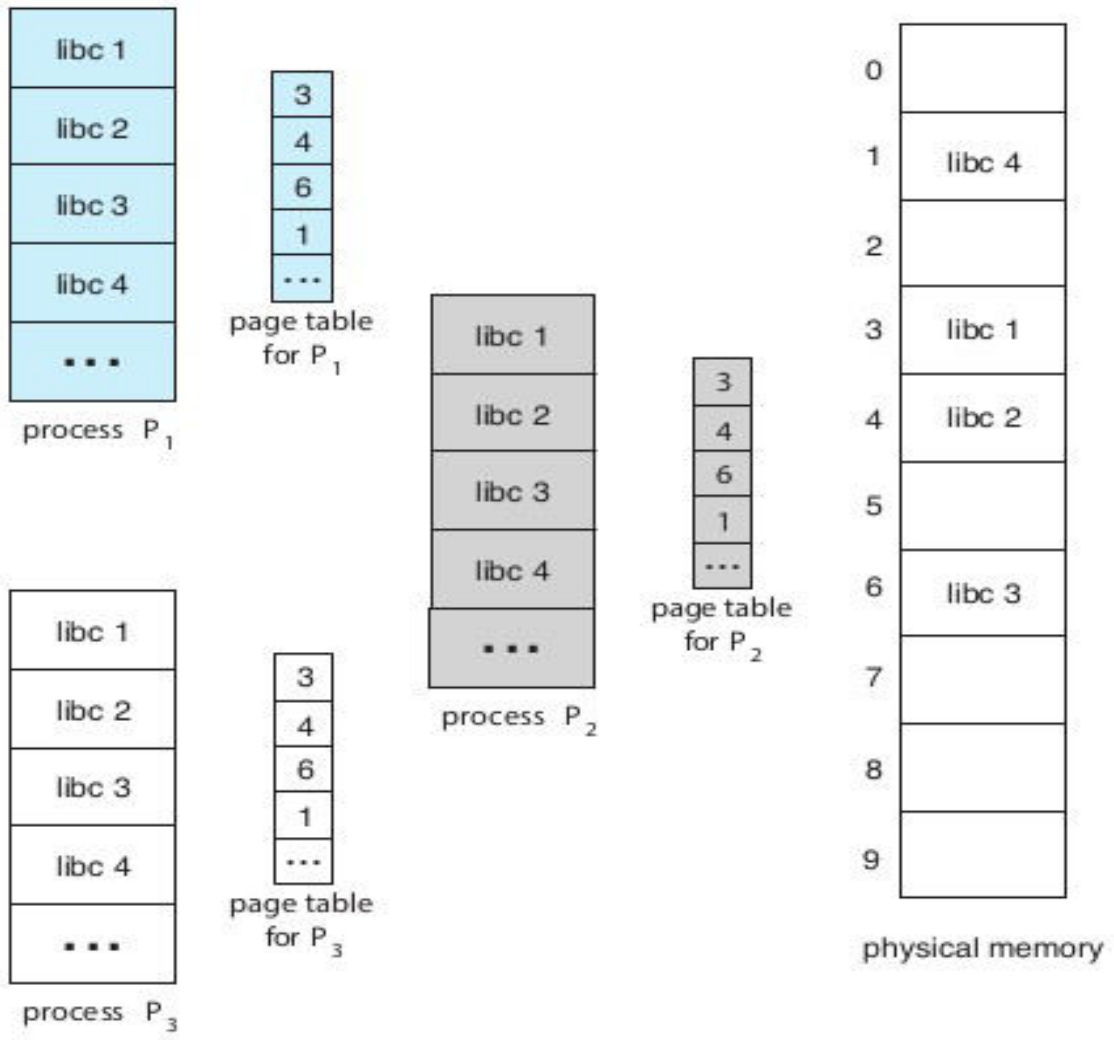


Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

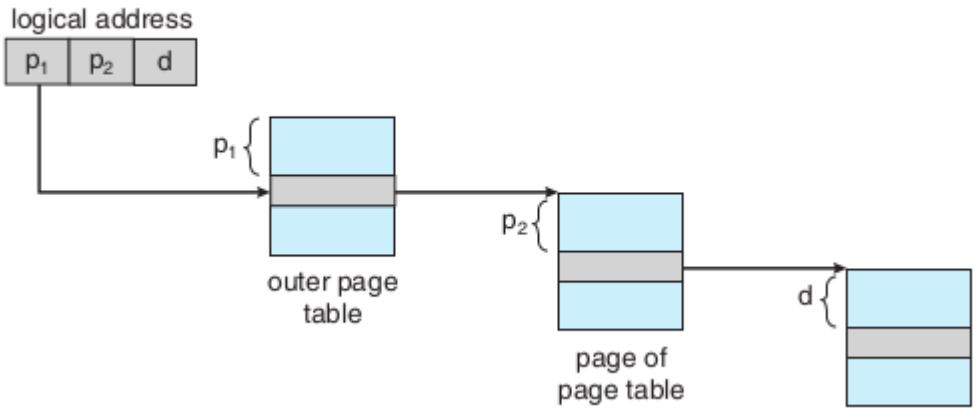
Paging: problem of large PT

- **64 bit address**
- **Suppose 20 bit offset**
  - **That means 2^20 = 1 MB pages**
  - **44 bit page number: 2^44 that is trillion sized page table!**
  - **Can’t have that big continuous page table!**

Paging: problem of large PT

- **32 bit address**
- **Suppose 12 bit offset**
  - **That means 2^12 = 4 KB pages**
  - **20 bit page number: 2^20 that is a million entries**
  - **Can’t always have that big continuous page table as well, for each process!**

Hierarchical paging



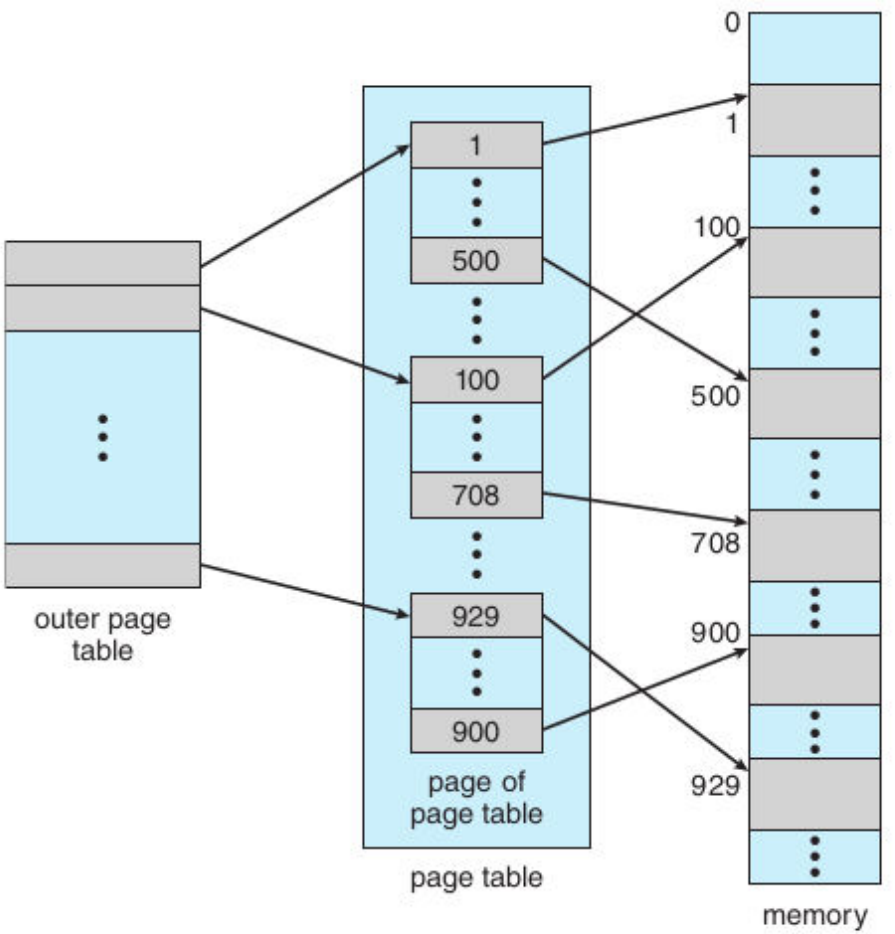
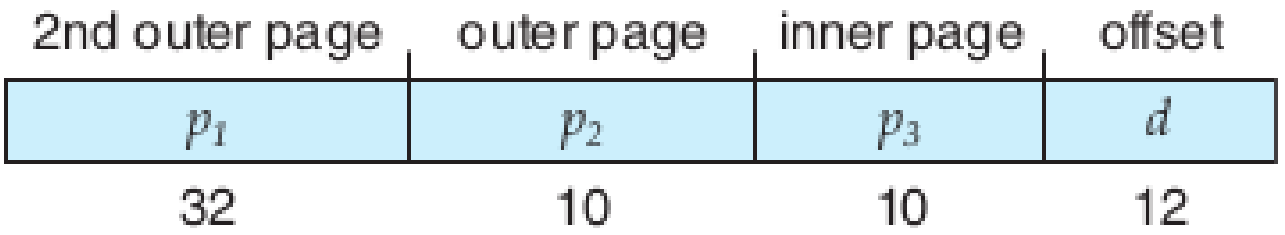


Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

More hierarchy



Problems with hierarchical paging

- **More number of memory accesses with each level !**
  - **Too slow !**
- **OS data structures also needed in that proportion**

Hashed page table

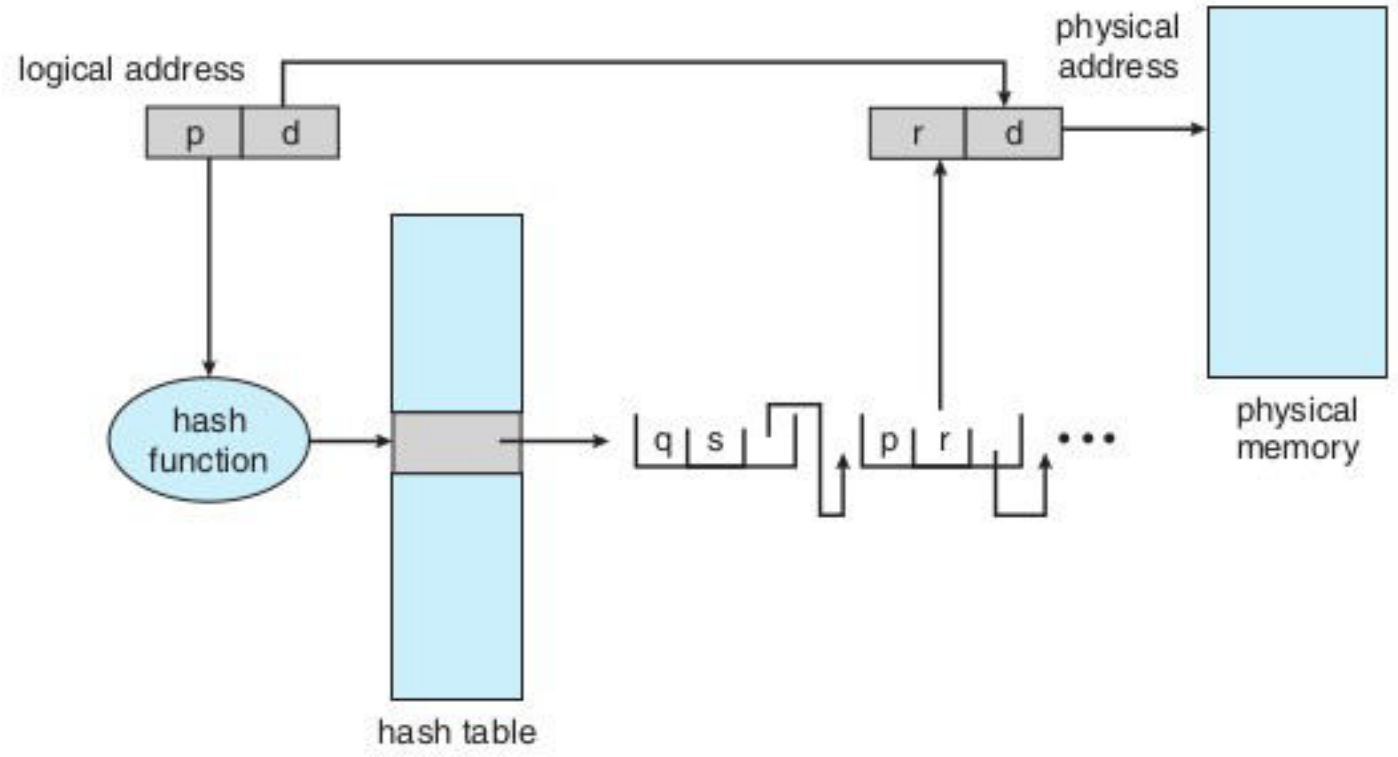


Figure 9.17 Hashed page table.

Inverted page table

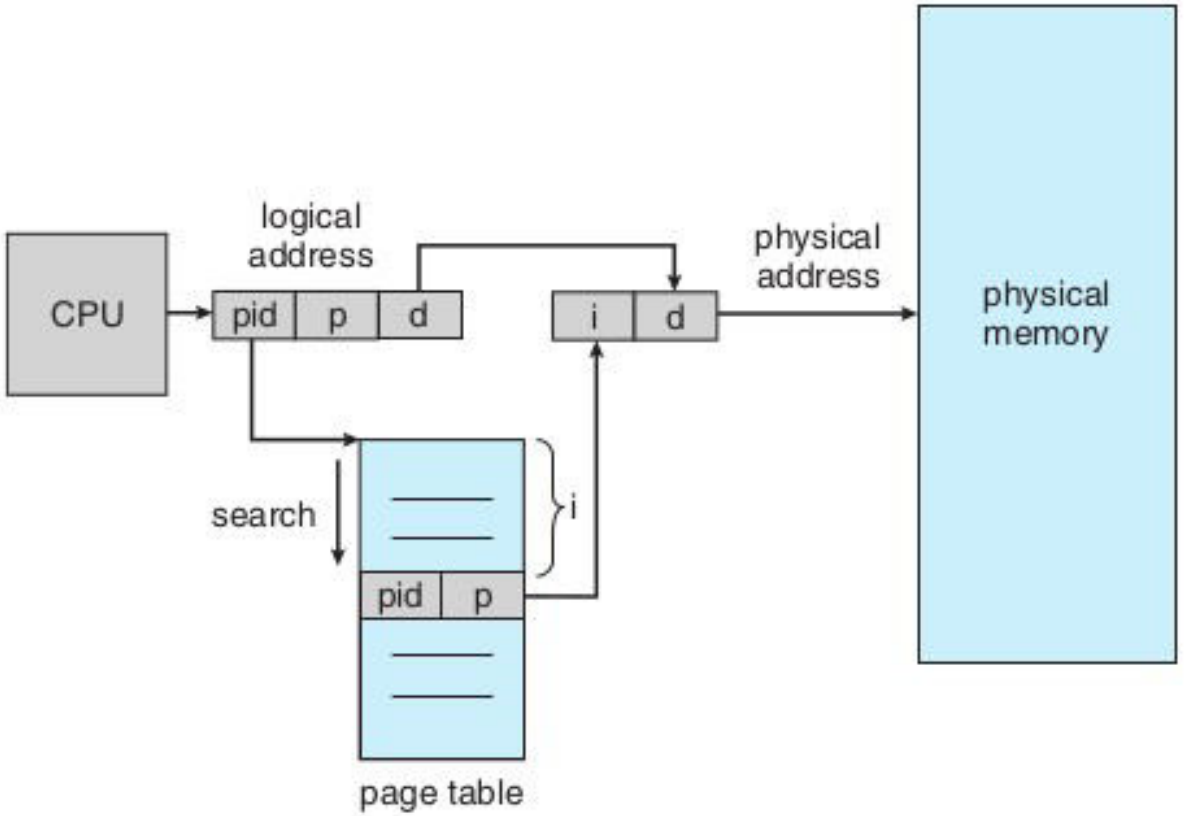


Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page
tables include the 64-bit Ultra SPARC and Power PC

virtual address
consists of a triple:
<process-id, page-number, offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- **Cashing levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)**
  - **CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.**
  - **A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page**
  - **When a virtual address reference occurs, the hardware searches the TLB for a translation.**
  - **If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup**

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

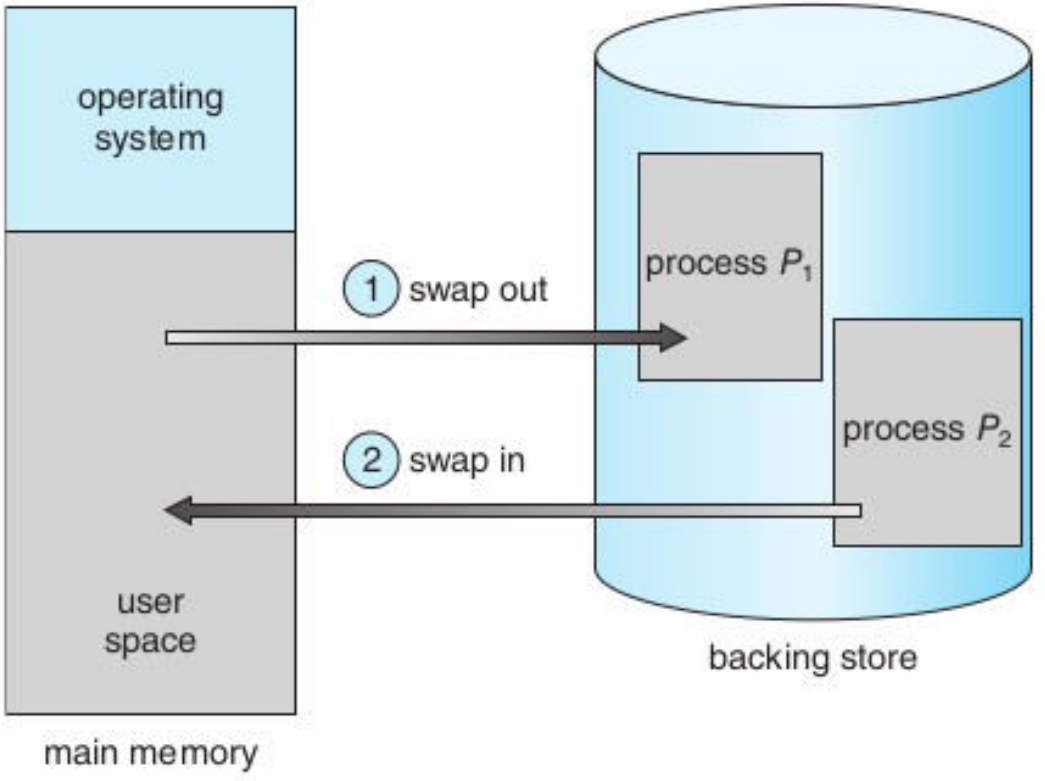


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping**
  - **Entire process swapped in or swapped out**
  - **With continous memory management**
- **Swapping with paging**
  - **Some pages are “paged out” and some “paged in”**
  - **Term “paging” refers to paging with swapping now**

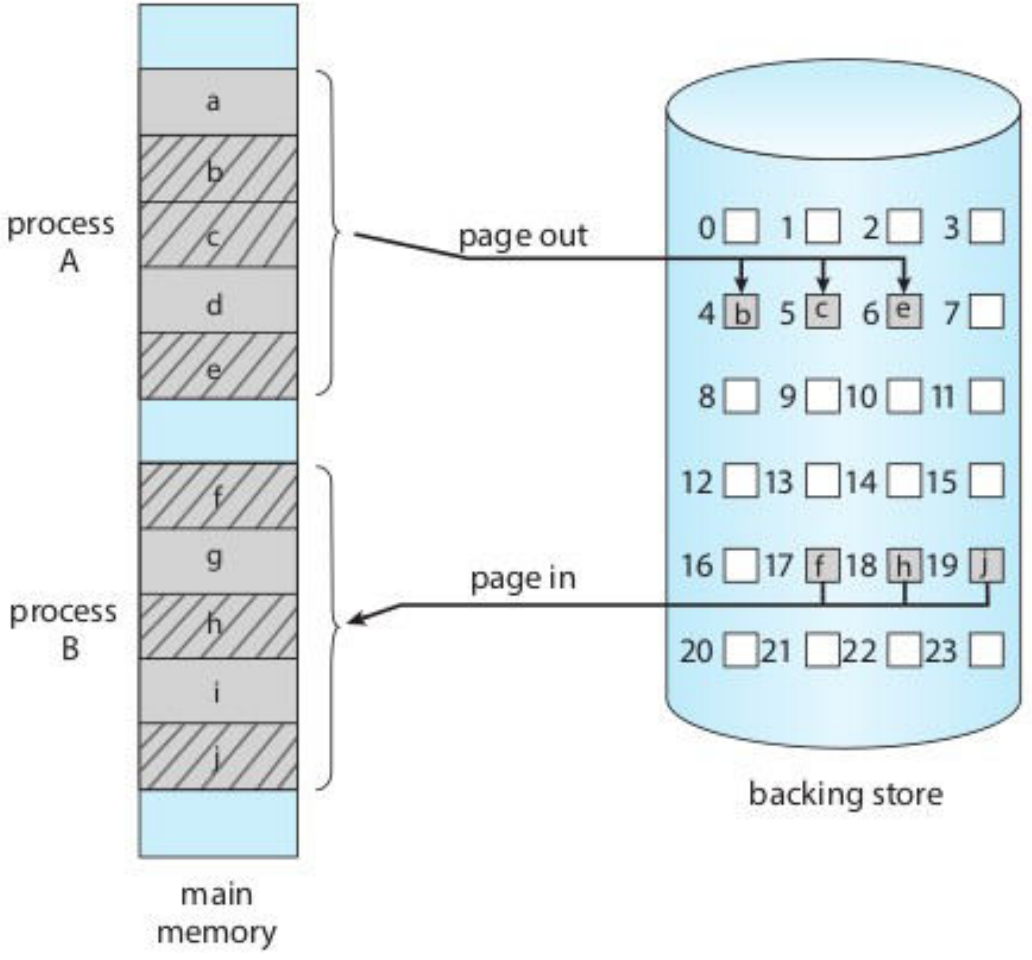


Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

- **Not as simple as it sounds when it comes to implementation**
  - **Writing OS code for this is challenging**