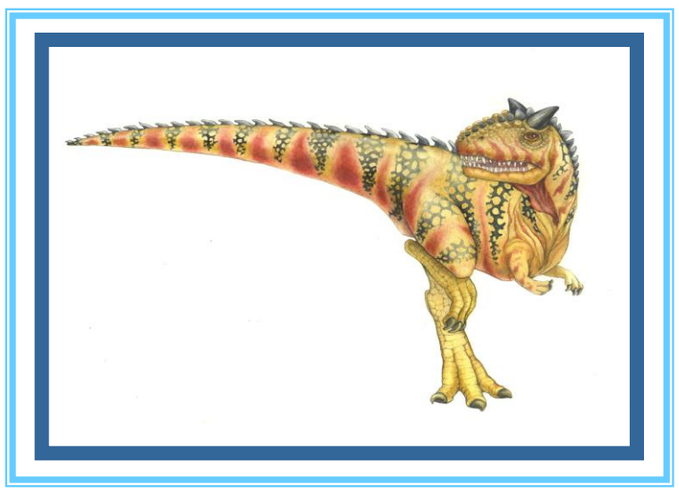
# Chapter 8:  Main Memory

التشابتر كله يتكلم عن:
كيف نظام التشغيل ينظم الذاكرة ويحميها ويخليها أسرع




Silberschatz, Galvin and Gagne ©2013



# Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one, CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

🔷 1. البرنامج لازم يدخل الذاكرة عشان يشتغل
البرنامج يكون محفوظ في (Disk)
لكن الـ CPU ما يقدر يشغله من هناك مباشرة ❌
لازم أول شيء:
يتم نقله إلى Main Memory (RAM)
✔ بعدين يتحول إلى Process ويبدأ التنفيذ
🔷 2. الـ CPU يتعامل فقط مع :
(داخل المعالج) Registers
Main Memory (RAM)
❌ ما يقدر يقرأ من الـ Disk
أي تخزين خارجي
🔷 3. السرعة
الـ Registers:
⚡ سريعة جدًا (تقريبًا في Clock واحدة)
الـ RAM:
🐢 أبطأ (تحتاج عدة Clocks)
يعني Registers أسرع بكثير من RAM
🔷 4. الكاش (Cache)
علشان يحل مشكلة السرعة:
📌 يتم وضع Cache بين: CPU و RAM
الفكرة: البيانات المستخدمة كثير يتم تخزينها في Cache
عشان الوصول يكون أسرع
🔷 5. حماية الذاكرة
ليش مهم؟
👉 لأن في أكثر من Process شغالين مع بعض
لو ما في حماية: برنامج ممكن يدخل على ذاكرة برنامج ثاني ❌
يسبب أخطاء أو كراش
✔ الحل:
كل Process له مساحة خاصة OS يمنع التداخل بينهم

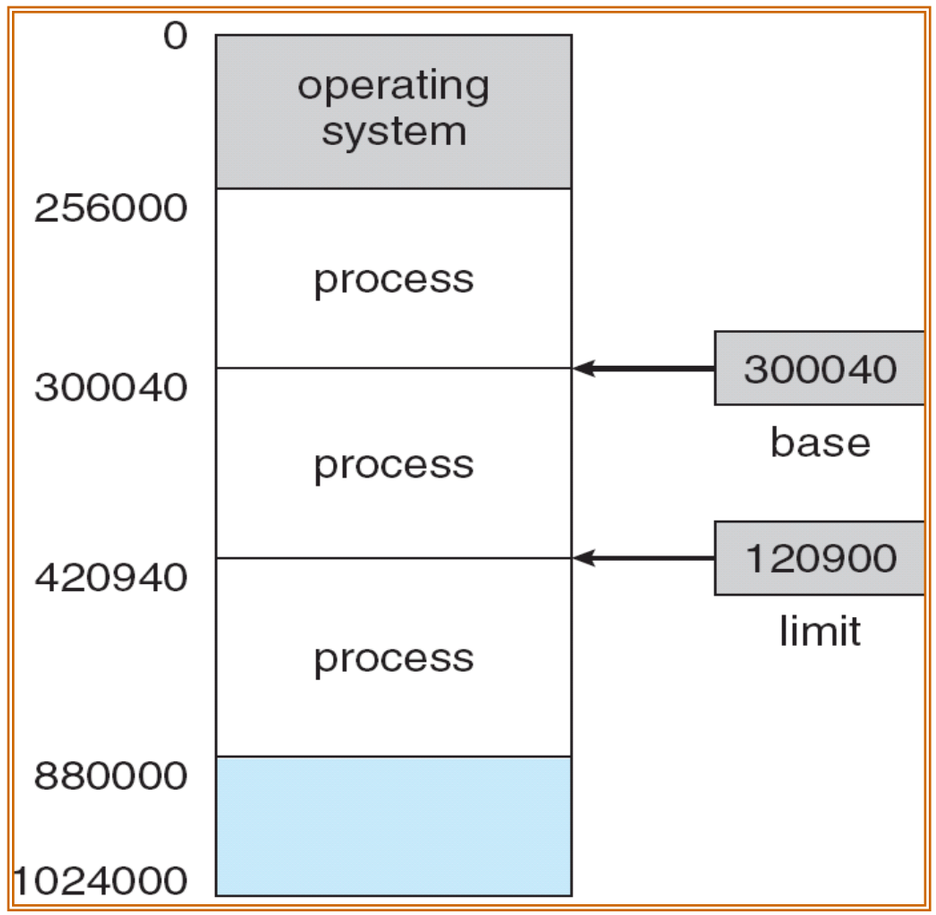


# Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space

🔷 ليش هذا مهم؟
✔ يوفر حماية:
كل **Process** ما يقدر يدخل على ذاكرة غيره

كل **Process** عنده:
**Base Register**
**Limit Register**
👉 وهذول الاثنين يحددوا:
وين تبدأ ذاكرته + كم حجمها
🔷 التعريف
✅ **Base Register** (سجل البداية)
يحتوي على:
👉 عنوان البداية في الذاكرة الفعلية (Physical Memory)
✅ **Limit Register** (سجل الحد)
يحتوي على:
👉 حجم الذاكرة المسموح بها لـ **process**
🔷 كيف يشتغلوا مع بعض؟
لما البرنامج يطلب عنوان (Logical Address):
الـ **CPU** يتحقق:
👉 هل العنوان < **limit** ؟
إذا ✔:
👉 يتم إضافة الـ **base**
👉 للحصول على العنوان الحقيقي (Physical Address)
إذا ❌:
👉 يحصل خطأ (Trap) → منع الوصول 🚫
🔷 مثال بسيط جدًا
**Base = 1000**
**Limit = 500**
يعني:
الذاكرة المسموحة من:
👉 1000 إلى 1499





# Binding of Instructions and Data to Memory

- **Address binding of instructions and data** to memory addresses can happen at three different stages.
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Swapping

Text

🔷 الفكرة الأساسية

👉 لما البرنامج يشتغل، لازم نحدد:

وين تنحط التعليمات والبيانات في الذاكرة؟

هذا يسمى: **Address Binding** (ربط العناوين)

🔷 متى يتم هذا الربط؟ في 3 مراحل:

🟢 **1. Compile Time** (وقت الترجمة)

📌 الفكرة:

إذا كنا نعرف مسبقًا مكان البرنامج في الذاكرة

👉 المترجم (Compiler) يعطي عناوين ثابتة (Absolute Addresses)

✔ مثال:

البرنامج رح ينحط دائمًا في العنوان 1000

كل العناوين تكون مبنية على هذا الرقم

❌ المشكلة:

لو تغير مكانه: لازم نعيد الترجمة (Recompile)

🟡 **2. Load Time** (وقت التحميل)

📌 الفكرة:

ما نعرف مكان البرنامج إلا وقت تحميله

👉 الحل:

نستخدم **Relocatable Code** (كود قابل للنقل)

✔ كيف يشتغل؟

العناوين تكون نسبية (Relative)

لما البرنامج ينزل في الذاكرة:

👉 يتم تعديل العناوين حسب المكان الجديد

✔ ميزة: ما يحتاج إعادة ترجمة

🔵 **3. Execution Time** (وقت التنفيذ) 🔥

📌 الفكرة:

البرنامج ممكن يتحرك أثناء التشغيل (مثلاً في swapping)

👉 الربط يتم أثناء التنفيذ نفسه

✔ يحتاج:

دعم من الهاردوير

مثل: **Base & Limit Register**

✔ كيف؟

كل عنوان يصدره البرنامج:

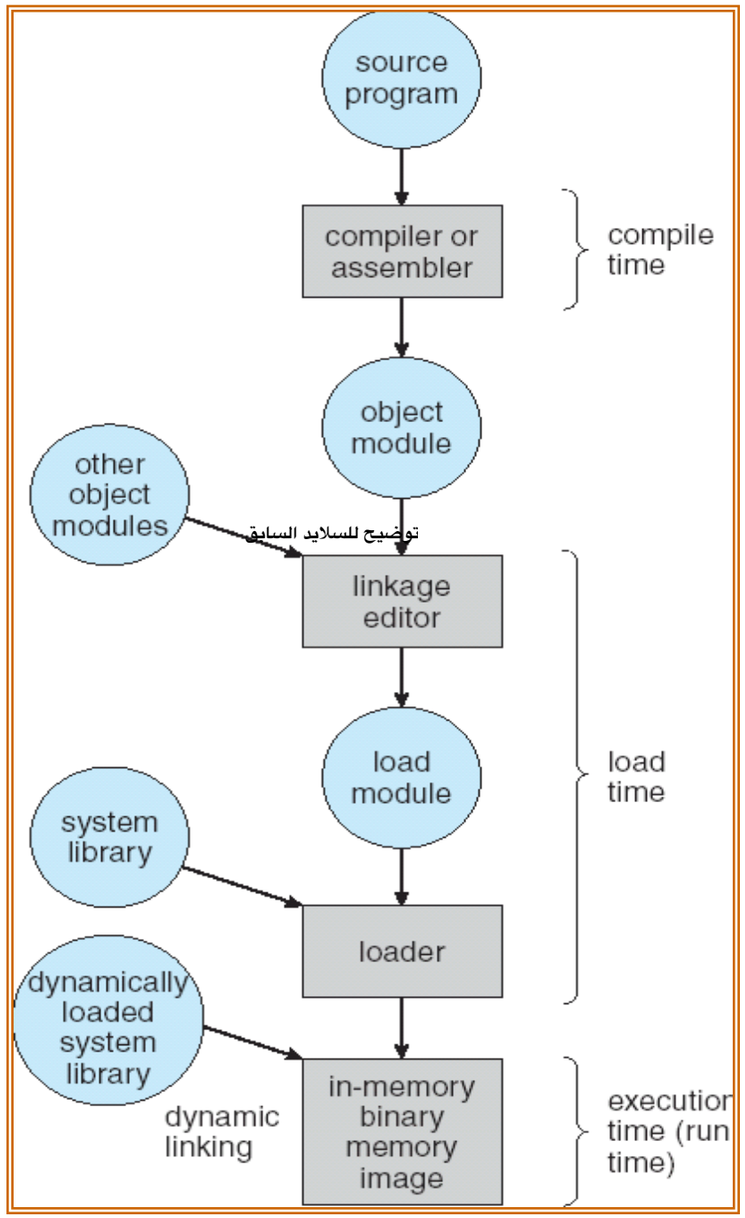
👉 يتم تحويله مباشرة إلى عنوان حقيقي

✔ ميزة:

مرونة عالية جدًا



# Multistep Processing of a User Program





# Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

عندنا نوعين من العناوين:
🔷 1. Logical Address (Virtual Address)
👉 هو العنوان اللي:
يولده CPU
يستخدمه البرنامج
✔ بمعنى:
البرنامج "يتخيل" أن هذا هو العنوان الحقيقي
🔷 2. Physical Address
👉 هو:
العنوان الحقيقي في RAM
اللي تستخدمه الذاكرة فعليًا
🧩 أهم نقطة
👉 البرنامج ما يعرف ولا يشوف الـ Physical Address
👉 هو يتعامل فقط مع Logical Address
🧠 طيب كيف يتحول العنوان؟
هنا يدخل:
👉 MMU (Memory Management Unit)
يقوم بـ:
Logical Address → Physical Address
🔷 مثال بسيط جدًا
📌 البرنامج يطلب:
Logical Address = 100
📌 لكن في الواقع:
البرنامج موجود في مكان ثاني في الذاكرة
مثلاً:
Base = 4000
👉 MMU يحول:
Physical Address = 4000 + 100 = 4100
🔥 المهم:
البرنامج يظن أنه في 100
لكن فعليًا في 4100
وهو ما يدري

🔷 متى يكونوا نفس الشي؟
🟢 Compile Time + Load Time
👉 Logical = Physical
✔ لأنه:
ما في تحويل
العنوان ثابت
🔵 Execution Time 🔥 (الأهم)
👉 Logical ≠ Physical
✔ لأنه:
في ترجمة باستخدام MMU
البرنامج ممكن يتحرك

🧩 الخلاصة المهمة جدًا
Logical Address:
👉 من CPU
👉 البرنامج يتعامل معه
Physical Address:
👉 في RAM
👉 الحقيقة
MMU:
👉 يحول بينهم
الفرق يظهر فقط في:
👉 Execution Time Binding





# Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

🔷 ما هو؟
👉 جهاز (Hardware) داخل النظام
✔ وظيفته:
يحول Logical Address → Physical Address
🔷 كيف يشتغل؟
👉 كل عنوان منطقي:
يتم إضافة قيمة عليه (من سجل اسمه relocation أو base)

🔷 مثال بسيط
Logical Address = 200
Base (Relocation Register) = 1000
👉 Physical Address =
1000 + 200 = 1200

🔷 نقطة مهمة جدًا
👉 البرنامج:
يتعامل فقط مع Logical Addresses
👉 ولا يعرف:
وين فعليًا موجود في الذاكرة ❌

🔷 ليش هذا مهم؟
✔ يعطي:
حماية (كل برنامج له مساحة خاصة)
مرونة (نقدر نحرك البرنامج)
سهولة إدارة الذاكرة



# Dynamic relocation using a relocation register

User program

0 #include …

1 #define y=…

2 main()

3 { some code

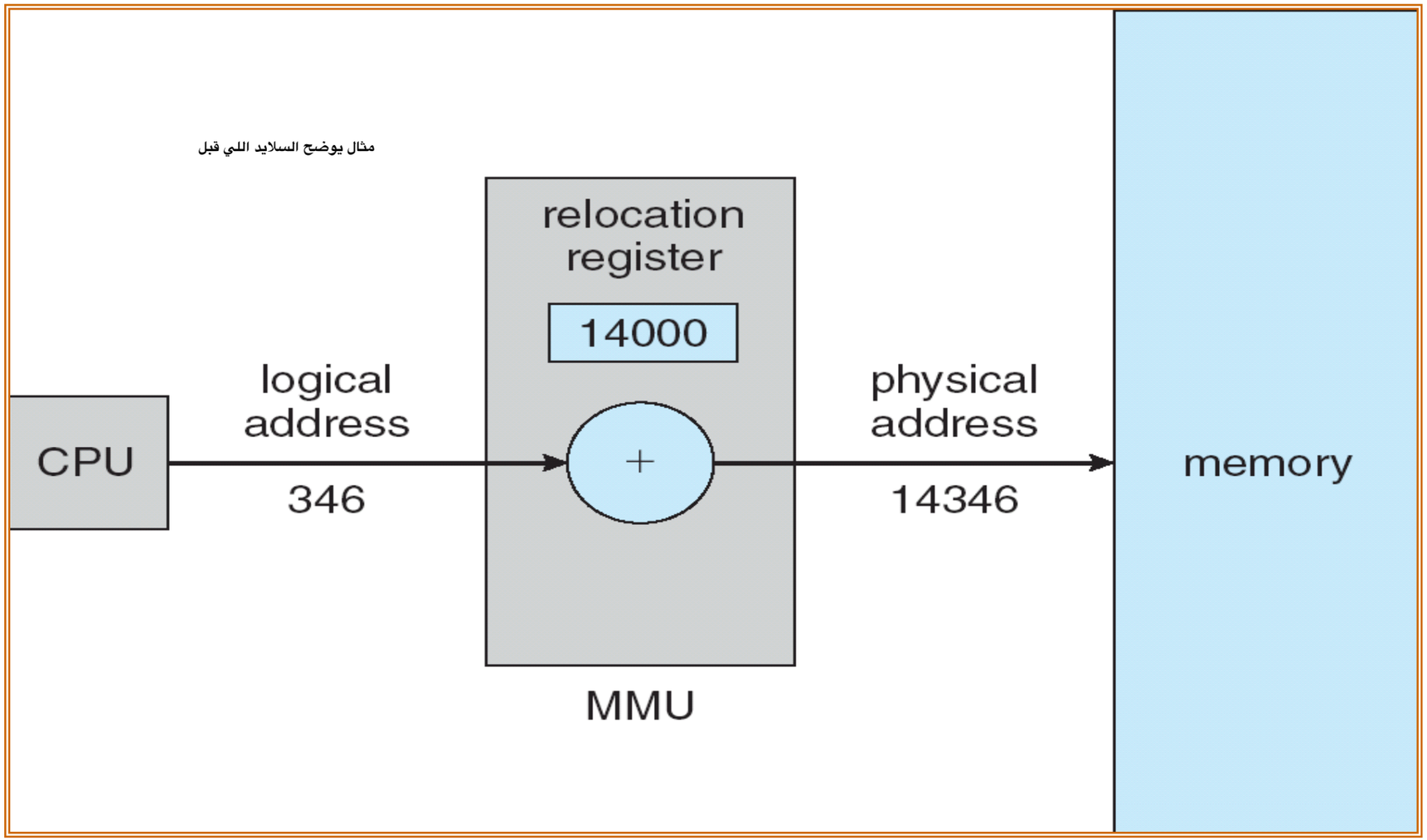
:

:

380 if( x=y ){

:

486 } //end of user program




Q.Map the logical addresses with its equivalent Physical addresses for the given program {change the code proj + table}



# Dynamic Loading and Linking

- Routine is not loaded until it is called
- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries

🔷 أولًا: Dynamic Loading (التحميل الديناميكي)
📌 الفكرة:
الدالة (الروتين) ما تتحمل في الذاكرة إلا لما تحتاجها
✔ مثال:
عندك برنامج فيه:
دالة طباعة 🖨️
دالة حساب 🧮
دالة رسم 🎨
لكن المستخدم:
استخدم فقط الطباعة
✔ إذًا: النظام يحمل فقط دالة الطباعة
والباقي ما يتحمل ❌
🎯 الفائدة:
يوفر مساحة في الذاكرة، يقلل استهلاك RAM
🔷 ثانيًا: Dynamic Linking (الربط الديناميكي)
👉 الربط مع المكتبات يتم:
أثناء التشغيل (Runtime)
وليس قبل
🔷 كيف يتم؟
1. وجود Stub (قطعة كود صغيرة)
👉 هذا كود بسيط بدل الدالة الحقيقية
2. لما يتم استدعاء الدالة:
الـ stub:
يبحث عن الدالة في المكتبة
يجيب عنوانها من الذاكرة
3. بعدين:
Stub replaces itself with the address
👉 يعني:
يحط العنوان الحقيقي
ويشغل الدالة مباشرة
🔷 مثال بسيط
تخيلي:
برنامجك يحتاج دالة من مكتبة (مثل cout <<)
بدل ما يحط الدالة كاملة:
👉 يحط stub
ولما تستدعيها:
يروح يجيبها من المكتبة
ويشغلها
🔷 ليش Dynamic Linking مهم؟
✔ لأنه:
نفس المكتبة تستخدمها برامج كثيرة
ما نكررها في كل برنامج





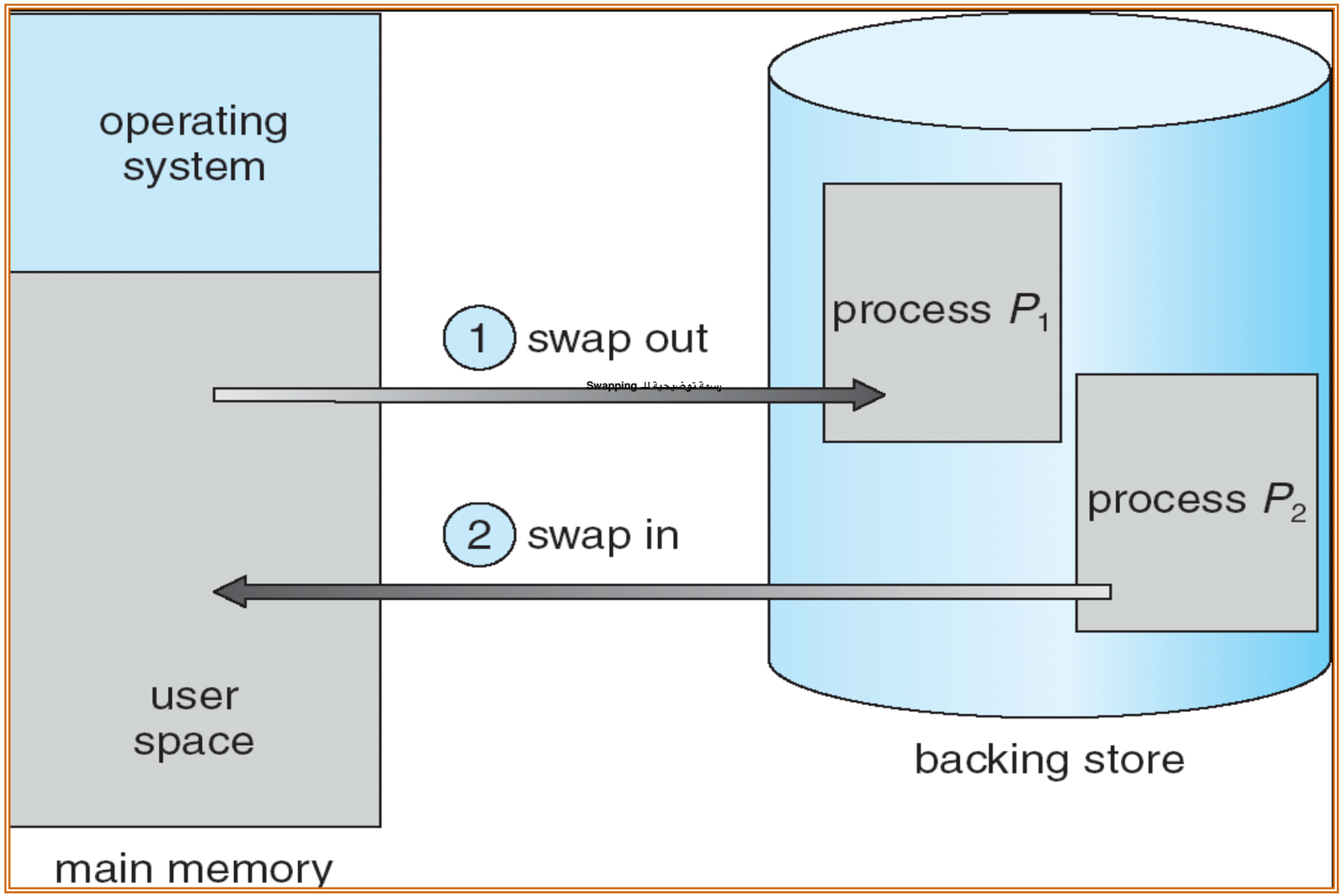
# Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images (movements of file parts from/to disk)
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

(التبديل) Swapping
🔷 الفكرة الأساسية
العملية (Process) ممكن تطلع من الذاكرة وترجع مرة ثانية
👉 يعني:
إذا الذاكرة (RAM) امتلأت
النظام يقول لعملية:
👉 "اطلعي مؤقتًا"
✔ يرسلها إلى:
➡️ Backing Store (الديسك)
وبعدين:
👉 يرجعها لما يحتاجها
🔷 Backing Store
سريع وكبير Disk
✔ يحتوي:
نسخ من العمليات (Processes)
✔ لازم يكون:
سريع نسبيًا
فيه direct access
🔷 Roll out / Roll in
👉 يستخدم مع الأولوية (Priority Scheduling)
مهم 🔥 → يدخل Process
أقل أهمية → يطلع Process
✔ مثال:
عندك Process A (low priority)
جاء Process B (high priority)
👉 النظام:
يطلع A (Roll out)
يدخل B (Roll in)
🔷 الأداء (Performance)
يعتمد على حجم البيانات Swap time
✔ كل ما زاد حجم العملية:
👉 زاد وقت النقل ⏳
⚠️ لأن:
النقل بين RAM و Disk بطيء
🔷 ملاحظة مهمة
✔ الأنظمة الحديثة:
مثل UNIX / Linux / Windows
👉 تستخدم نسخ مطورة من swapping
(غالبًا مع Virtual Memory)



# View of Swapping





# Contiguous Allocation

- Main memory usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register (start from 0) for example:

    Base=1200

    Limit=1500(300=length)
  - MMU maps logical address *dynamically*





🔷 الفكرة الأساسية
👉 كل Process ياخذ:
➡️ قطعة واحدة متصلة (Continuous block) في الذاكرة
🔷 تقسيم الذاكرة
الذاكرة تنقسم إلى:
1. جزء للنظام (OS)
يكون في:
👉 Low Memory
يحتوي:
نظام التشغيل
interrupt vector
2. جزء للمستخدمين
يكون في:
👉 High Memory
فيه العمليات (Processes)
🔷 الحماية باستخدام Base & Limit
رجعنا لموضوع قبل شوي👇
✔ Base Register:
أصغر عنوان فيزيائي
✔ Limit Register:
حجم الذاكرة المسموحة
🔷 شرط مهم
كل Logical Address لازم يكون أقل من Limit
🔷 مثال
Base = 1200
Limit = 300
👉 العناوين المسموحة:
من 0 إلى 299 (Logical)
📌 لو Logical Address = 100
✔ مسموح
👉 Physical =
1200 + 100 = 1300
📌 لو Logical Address = 400
❌ مرفوض (أكبر من limit)
🔷 دور MMU
MMU maps logical → physical dynamically
👉 كل عنوان:
يتحقق منه
ثم يتحول



# Swapping on Mobile Systems



- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory

(يسأل التطبيقات للتخلي عن طواعية الذاكرة المخصصة)

    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes application state to flash for fast restart
  - Both OSes support paging

🔷 الفكرة الأساسية
الموبايلات غالبًا ما تستخدم swapping التقليدي
❌ ليش ما يستخدمونه؟
1. يعتمدون على Flash Memory
مو مثل الهارد العادي
⚠️ مشاكل الـ Flash:
🔶 مساحة صغيرة
ما تقدر تخزن عمليات كثيرة
🔶 عدد كتابة محدود
كل ما تكتب/تمسح → يقل عمر الذاكرة
🔶 بطيء نسبيًا
النقل بين CPU و Flash أبطأ من RAM
🔷 بدل swapping، يستخدمون حلول أخرى:
🍎 في iOS:
👉 النظام يقول للتطبيقات:
"خففي استخدامك للذاكرة"
✔ إذا استجاب:
يحذف بيانات غير مهمة (مثل cache)
✔ مثال:
بيانات للقراءة فقط (read-only)
👉 تنحذف، وإذا احتجناها:
👉 نرجع نحملها من الـ flash
❌ إذا ما استجاب:
👉 يتم إغلاق التطبيق (terminate)
🤖 في Android:
👉 إذا الذاكرة قليلة:
النظام يبدأ يحذف التطبيقات
لكن قبل:
✔ يحفظ حالتها (state) في الـ flash
👉 عشان:
✔ ترجع بسرعة لما تفتحها
🔷 نقطة مهمة جدًا
✔ الاثنين (iOS و Android):
👉 يدعمون Paging
❌ لكن:
👉 ما يعتمدون على swapping التقليدي





# HW address protection with base and limit registers

User program

0 #include …

1 define y=…

2 main()

3 { some code

.

.

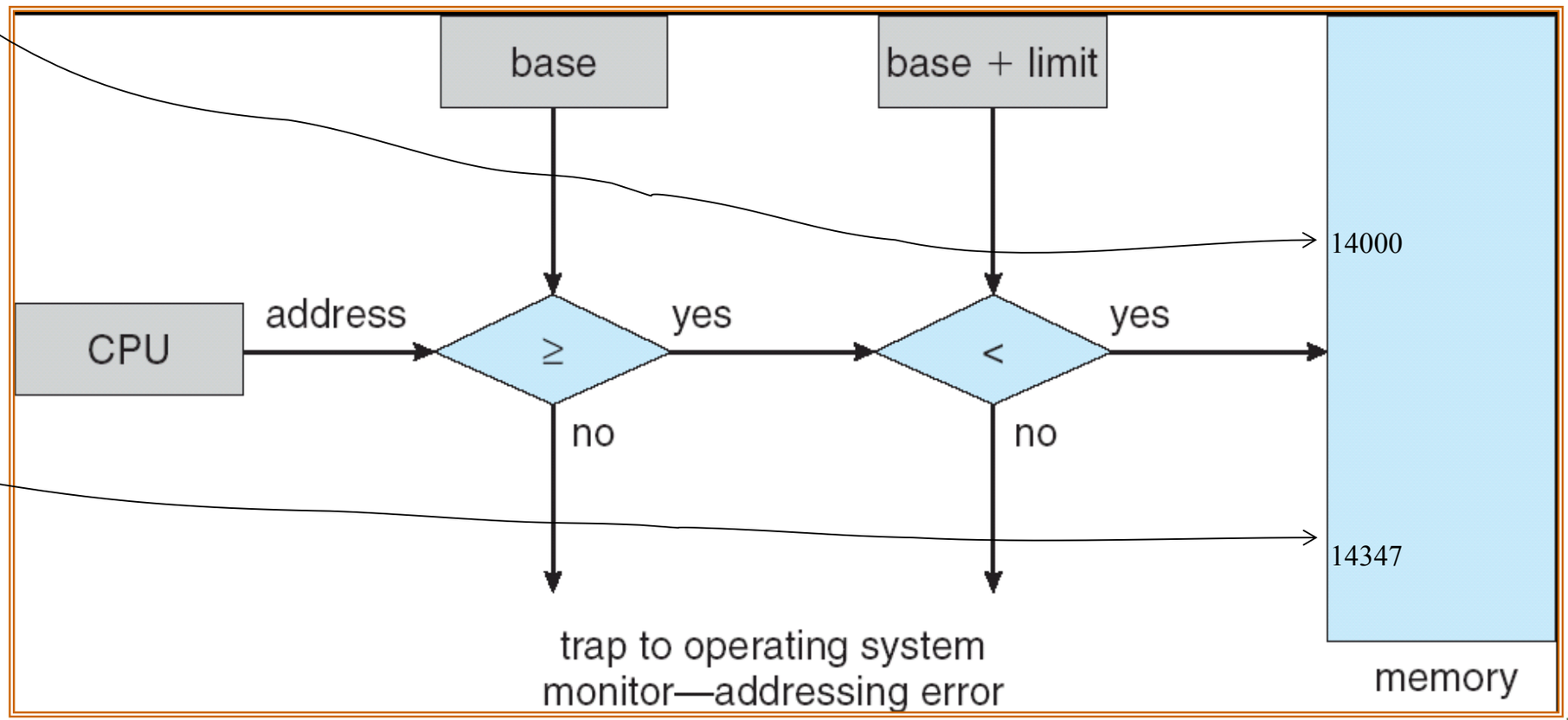
340 if( x=y ){

.

. }

346 // end of user

program








# Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions  b) free partitions (hole)

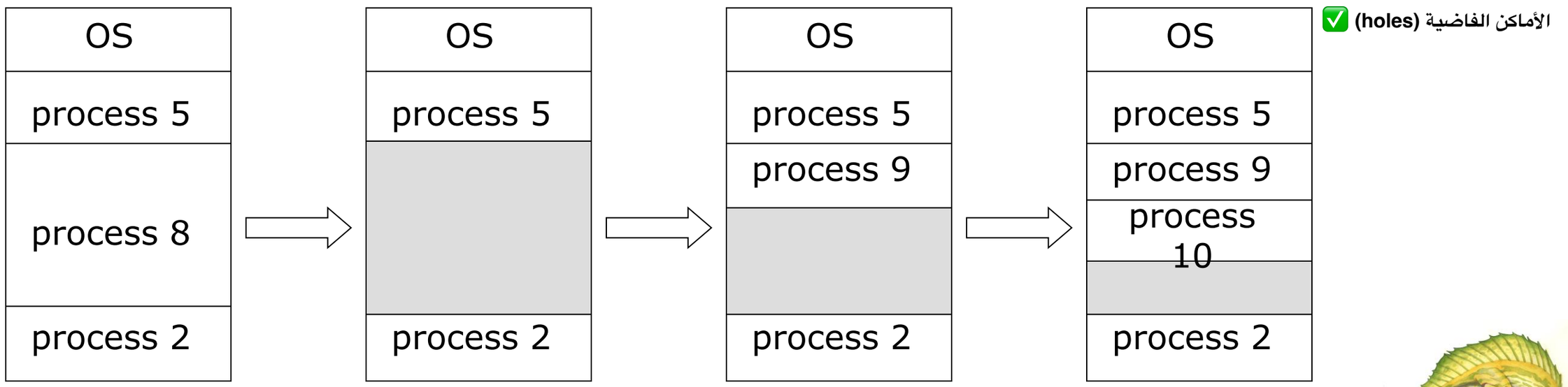
مفهوم Hole (مهم جدًا)
مساحة فاضية في الذاكرة = Hole

👉 بعد حذف عمليات:
تتكون فراغات بأحجام مختلفة

🔷 كيف يتم التخصيص؟

👉 لما تجي عملية جديدة:
النظام يبحث عن Hole مناسب

🔷 النظام يحتفظ بمعلومات عن:
الأماكن المستخدمة ✅
الأماكن الفاضية (holes) ✅








# Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

🔷 المشكلة:
كيف نختار Hole مناسب لعملية حجمها n؟
🔶 1. First-Fit
👉 خذ أول Hole مناسب
✔ أسرع
✔ ما يحتاج بحث طويل
🔶 2. Best-Fit
👉 خذ أصغر Hole يكفي
✔ يقلل الهدر
❌ يحتاج بحث في كل القائمة
🔶 3. Worst-Fit
👉 خذ أكبر Hole
✔ يترك مساحة كبيرة
❌ غالبًا سيء في الاستخدام





# Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - [redacted]

🔷 الفكرة الأساسية
👉 مع توزيع الذاكرة، يصير عندنا:
فراغات غير مفيدة
أو مساحة ضائعة
🔶 1. External Fragmentation (تجزئة خارجية)
فيه مساحة كافية، لكن مو متصلة
📌 يعني:
عندك فراغات كثيرة
لكن متفرقة ❌
ما تقدر تستخدمها لعملية كبيرة
🔶 2. Internal Fragmentation (تجزئة داخلية)
المساحة المخصصة أكبر من المطلوبة
📌 يعني:
العملية أخذت مساحة أكبر من احتياجها
والزيادة ضاعت
🔷 كيف نقلل External Fragmentation؟
✔ الحل: Compaction
نجمع الفراغات في مكان واحد
📌 كيف؟
نحرك العمليات
نجمع كل الفراغات → بلوك واحد كبير
⚠️ شرط مهم:
لازم يكون عندنا Dynamic Relocation
👉 يعني:
نقدر نغير مواقع العمليات أثناء التشغيل
❌ المشكلة:
بطيء ومكلف ⏳





# Paging



- Logical address space of a process can be noncontiguous; process is allocated in physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation with in the page

🔷 الفكرة الأساسية
ما نحتاج الذاكرة تكون متصلة
👉 هذا حل مباشر لمشكلة:
✔ External Fragmentation
🔷 كيف يشتغل؟
1. نقسم الذاكرة:
✔ Physical Memory:
👉 إلى Frames (إطارات)
✔ Logical Memory:
👉 إلى Pages (صفحات)
✔ نفس الحجم تمامًا
🔷 ليش الحجم ثابت؟
عادة يكون:
👉 قوة للعدد 2 (مثل 512, 1024...)
🔷 التنفيذ
👉 إذا البرنامج عنده n صفحات:
نحتاج:
👉 n Frames فاضية
✔ ونوزعهم في أي مكان في الذاكرة
🔷 Page Table
جدول يربط Page → Frame
👉 يستخدمه النظام لتحويل:
Logical Address → Physical Address
🔷 مثال بسيط
Page 0 → Frame 5
Page 1 → Frame 2
👉 لما البرنامج يطلب:
Page 1
النظام يروح:
👉 Frame 2
🔷 أهم ميزة
✔ ما في ❌ External Fragmentation
🔷 لكن...
Internal Fragmentation still exists





# Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

| page number | page offset |
|---|---|
| *p* | *d* |
| *m - n* | *n* |

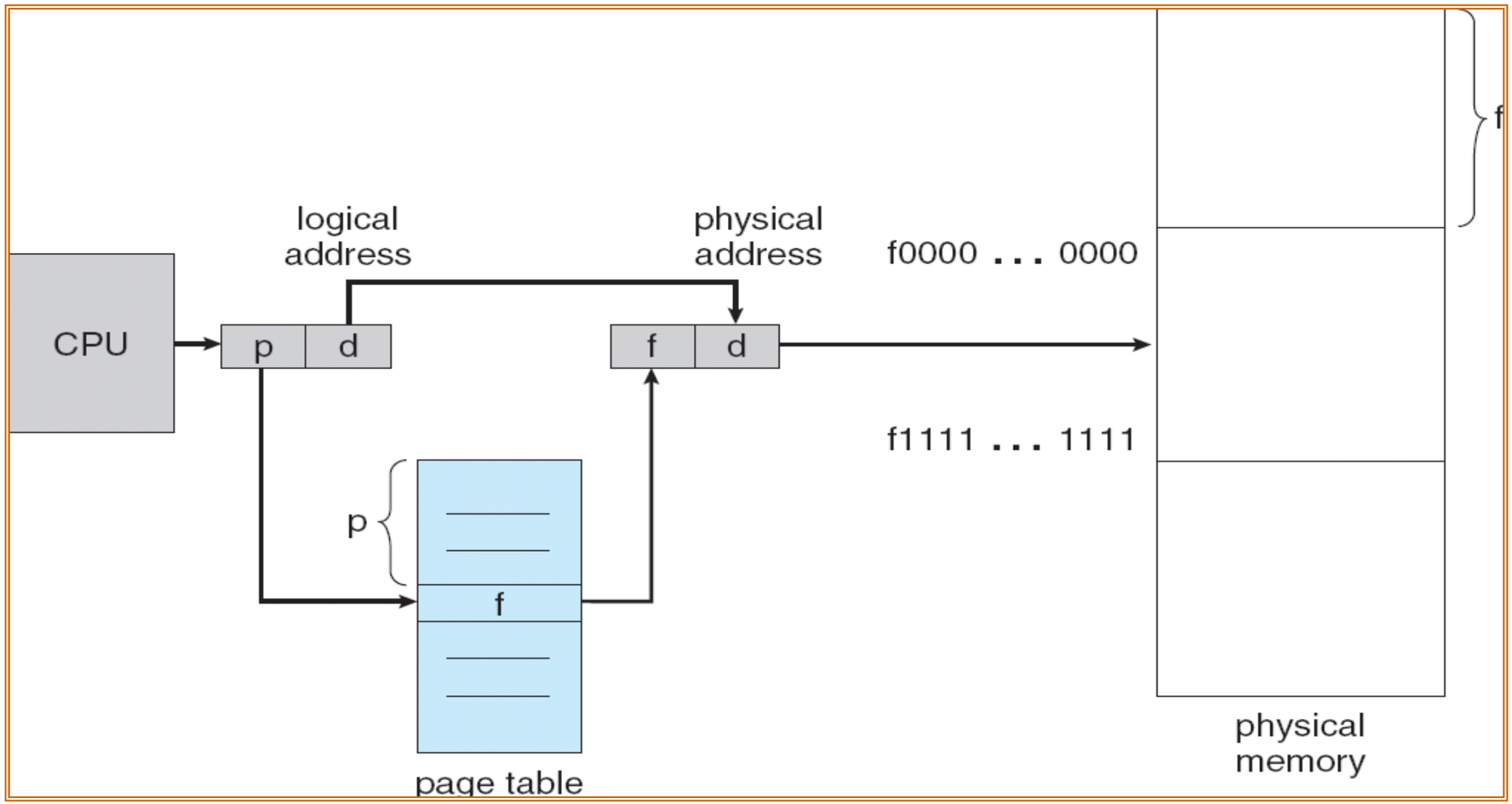
  - For given logical address space $2^m$ *and page size* $2^n$

🔷 الفكرة الأساسية
العنوان اللي يطلعه الـ CPU (Logical Address) ينقسم إلى جزئين
🔶 مكونات العنوان:
1. Page Number (p)
رقم الصفحة
يستخدم كـ:
👉 index داخل الـ Page Table
2. Offset (d)
موقع البيانات داخل الصفحة
🔷 شكل العنوان:
Logical Address = (p, d)
🔷 كيف نحول إلى Physical Address؟
نأخذ p
نروح لـ Page Table
نجيب:
👉 رقم الـ Frame
نضيف له offset (d)
👉 الناتج:
Physical Address = (frame, d)
🔷 مثال بسيط
Page size = 1000
Logical Address = 2500
👉 نقسم:
p = 2
d = 500
لو:
Page 2 → Frame 7
👉 Physical Address:
Frame 7 + offset 500



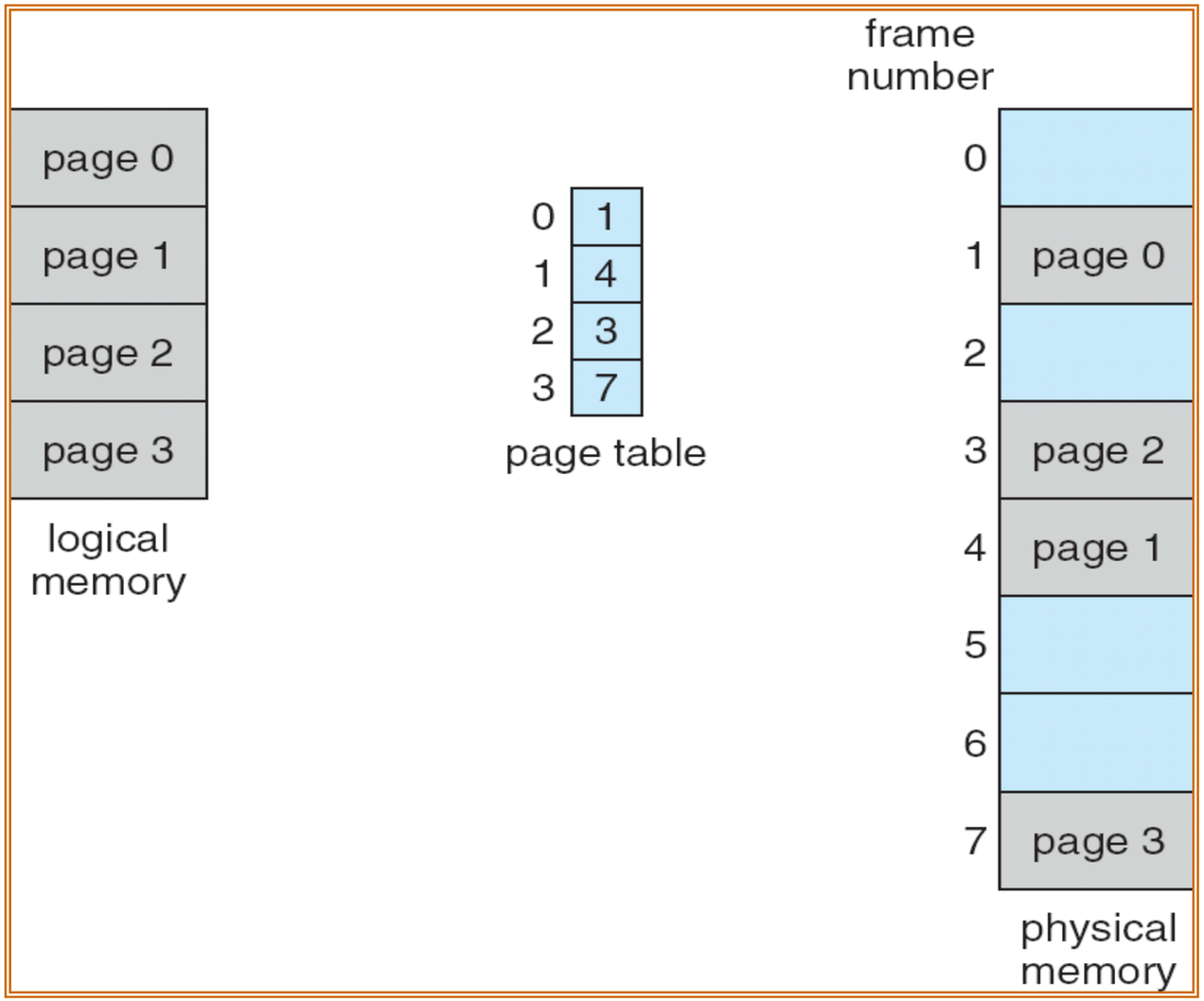


# Paging Hardware





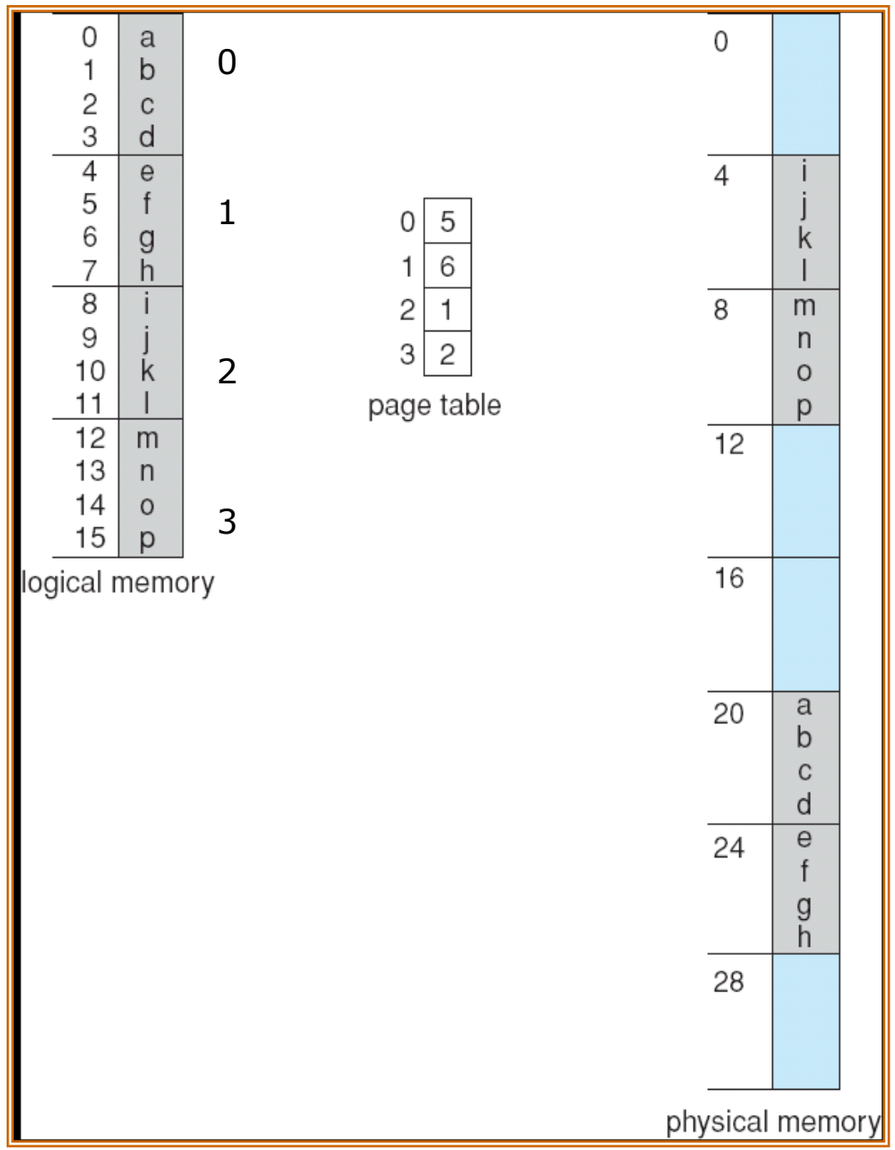

# Paging Model of Logical and Physical Memory





# Paging Example

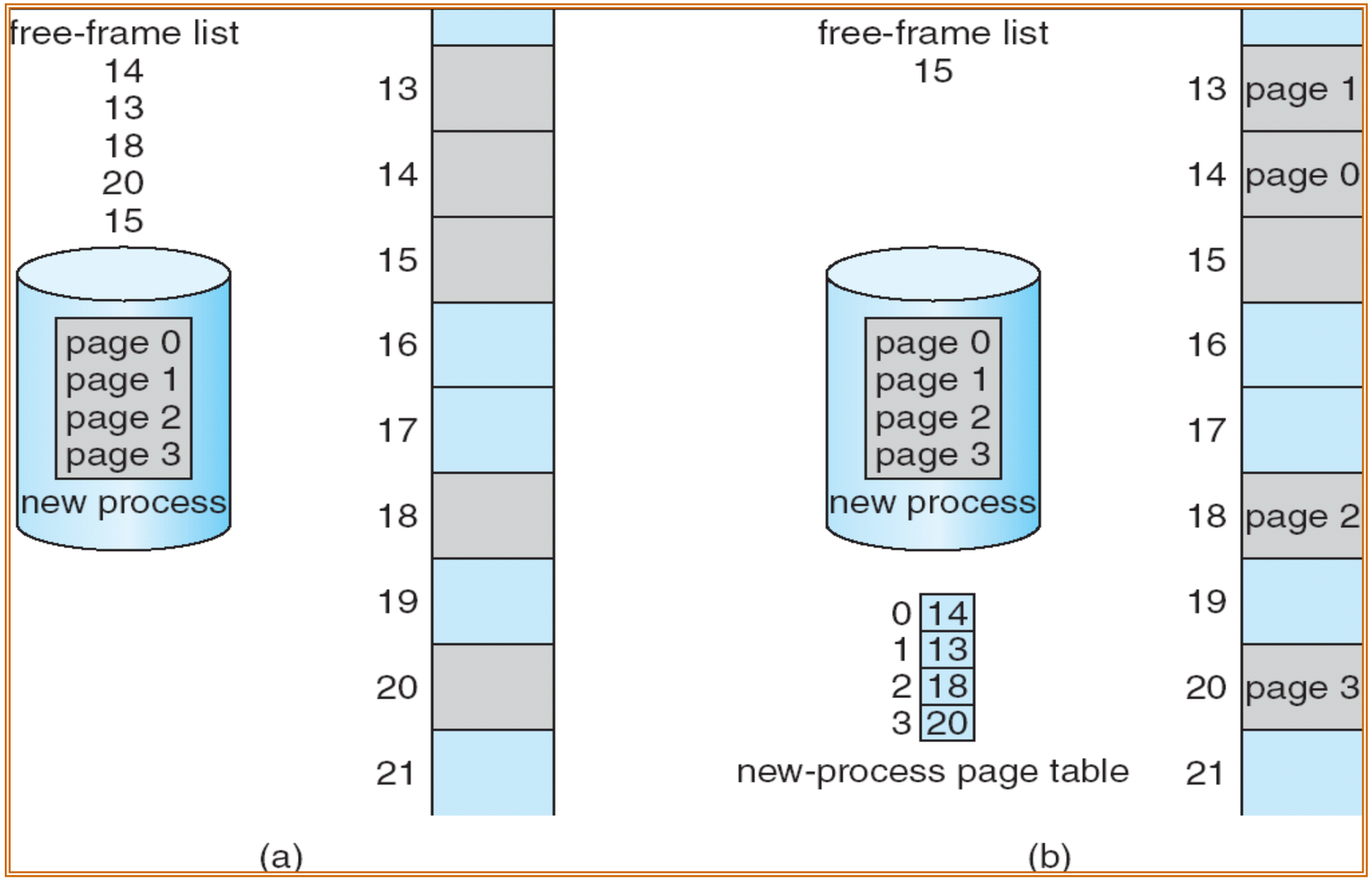
As a concrete (although minuscule) example, consider the memory in Figure 8.9. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 × 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 × 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 × 4) + 0). Logical address 13 maps to physical address 9.




32-byte memory and 4-byte pages



# Free Frames




Before allocation

After allocation



# Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

🔷 وين نخزن الـ Page Table؟
في الـ Main Memory (RAM)
🔷 Registers مهمة:
✔ PTBR (Page Table Base Register)
يشير إلى:
👉 بداية الـ Page Table
✔ PTLR (Page Table Length Register)
يحتوي:
👉 حجم الـ Page Table
🔷 المشكلة الكبيرة ⚠️
كل عملية تحتاج مرتين وصول للذاكرة
❗ ليش؟
نجيب Frame من Page Table
نجيب البيانات نفسها
👉 يعني:
⏳ بطيء
🧠 الحل: TLB (Translation Lookaside Buffer) 🔥
🔷 ما هو؟
كاش سريع جدًا للعناوين
🔷 كيف يشتغل؟
إذا الصفحة موجودة في TLB:
👉 نجيب Frame فورًا ⚡
إذا لا:
👉 نروح للـ Page Table
🔷 النتيجة:
✔ يقلل عدد مرات الوصول للذاكرة
✔ يسرّع النظام
🔷 ASIDs (معلومة امتحانية)
بعض TLBs تحتوي ASID
👉 يعني:
كل عملية لها ID خاص
يمنع تداخل العناوين بين العمليات
✔ حماية + سرعة





# Associative Memory

- Associative memory – parallel search

| Page # | Frame # |
|---|---|
| | |
| | |
| | |
| | |

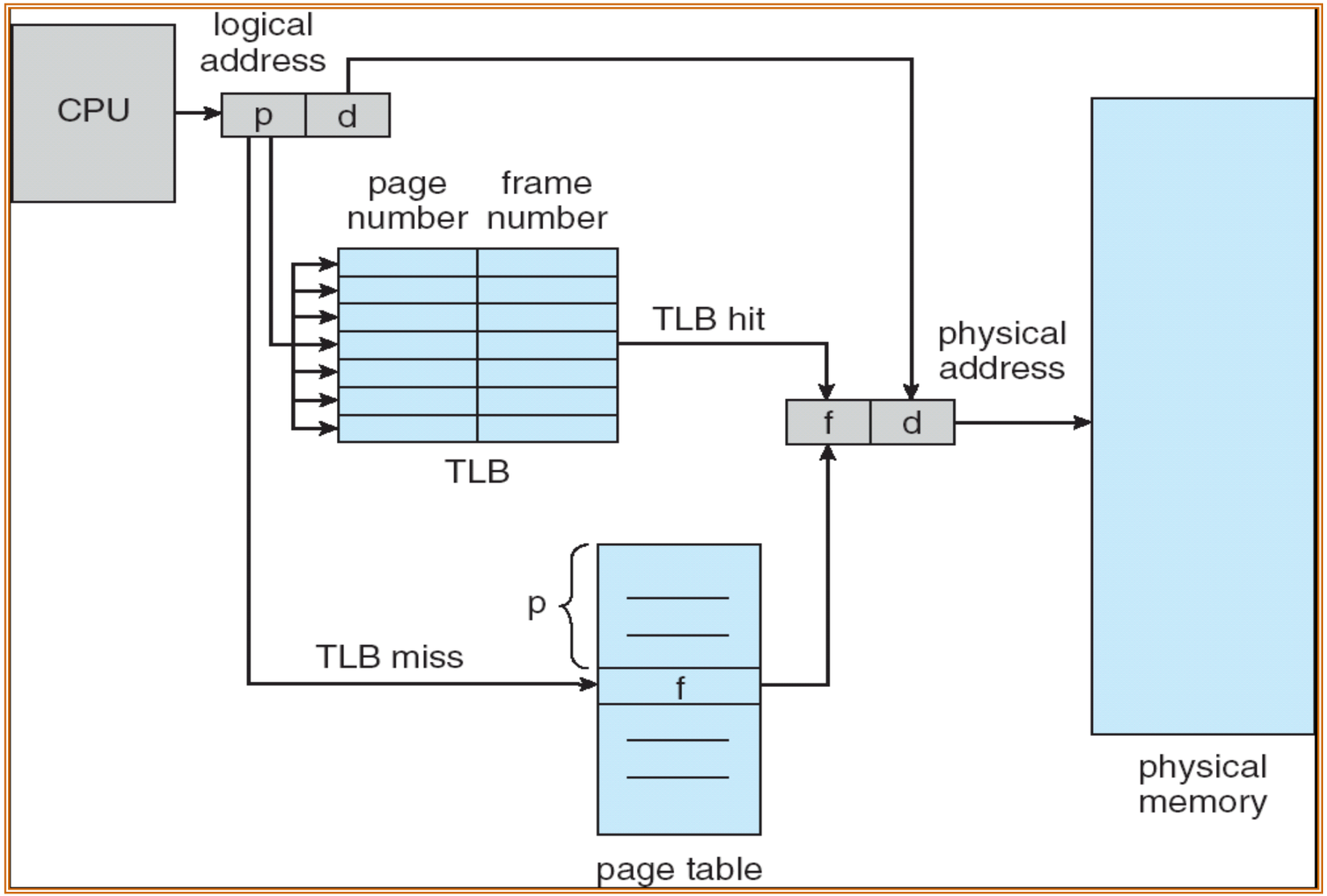
Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

🔷 الفكرة
بحث سريع جدًا (parallel search)
🔷 كيف يستخدم في TLB؟
عندنا (p, d):
✔ الحالة 1: p موجود في TLB
👉 نجيب Frame مباشرة ⚡
❌ الحالة 2: p غير موجود
👉 نروح:
Page Table في RAM
👉 نجيب Frame
👉 نحدث TLB



# Paging Hardware With TLB hit





# Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

العثور

🔷 الفكرة

👉 هذا يقيس:

كم يستغرق وقت الوصول للذاكرة مع وجود TLB

🔷 عندنا حالتين:

✔ 1. Hit (TLB الصفحة (موجود فيها

نلاقي الصفحة بسرعة في TLB

وقت الوصول = سريع

👉 الزمن:

1 + ε

❌ 2. Miss (TLB غير موجودة في)

نروح Page Table في الذاكرة

ثم نجيب البيانات

👉 الزمن:

2 + ε

🔷 Hit Ratio (α)

👉 نسبة نجاح البحث في TLB

α = نسبة الـ Hit

(1 – α) = Miss

🔥 المعادلة النهائية:

EAT = (1 + ε)α + (2 + ε)(1 – α)

تبسيطها:

EAT = 2 + ε – α

🎯 الفكرة المهمة:

✔ كل ما زاد Hit Ratio → السرعة تزيد





# Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

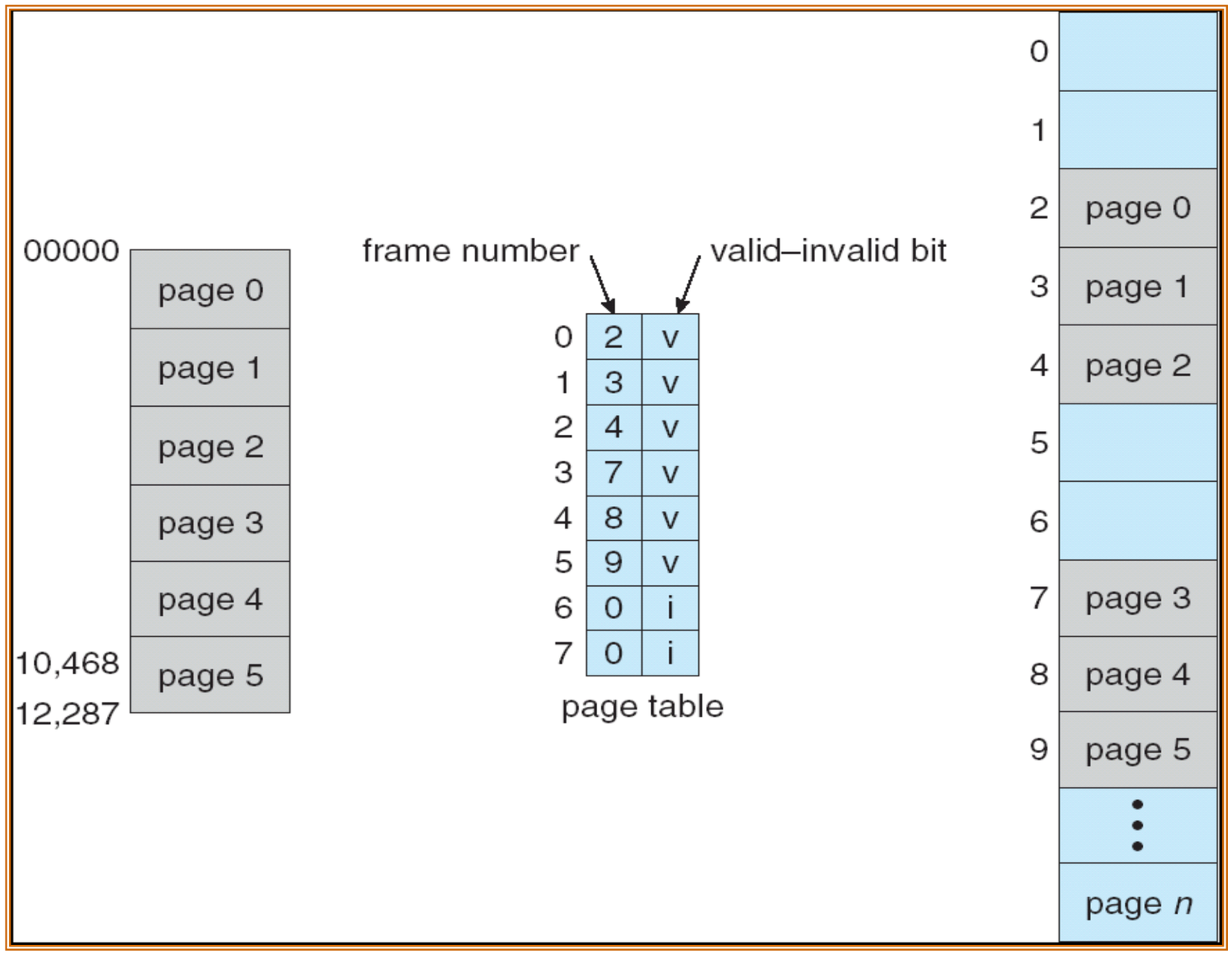
**الفكرة الأساسية:**
**النظام لازم يمنع أي برنامج إنه:**
**يدخل على ذاكرة برنامج ثاني ❌**
**أو يستخدم ذاكرة غير مخصصة له ❌**
**كيف نحمي الذاكرة؟**

**1. Protection Bit (لكل Frame)**
**كل frame في الذاكرة له بت يحدد:**
**هل الصفحة:**
**read-only (قراءة فقط)**
**أو read/write**
**👉 مثال:**
**برنامج يحاول يكتب في صفحة read-only → ❌ يطلع error (trap)**

**2. Valid / Invalid Bit (في Page Table)**
**لكل صفحة عندك بت مهم جدًا:**
**✅ Valid**
**الصفحة ضمن مساحة البرنامج (logical address space)**
**يعني مسموح استخدامها**
**❌ Invalid**
**الصفحة مو تابعة للبرنامج**
**الوصول لها = خطأ (segmentation fault)**



# Valid (v) or Invalid (i) Bit In A Page Table





# Shared Pages

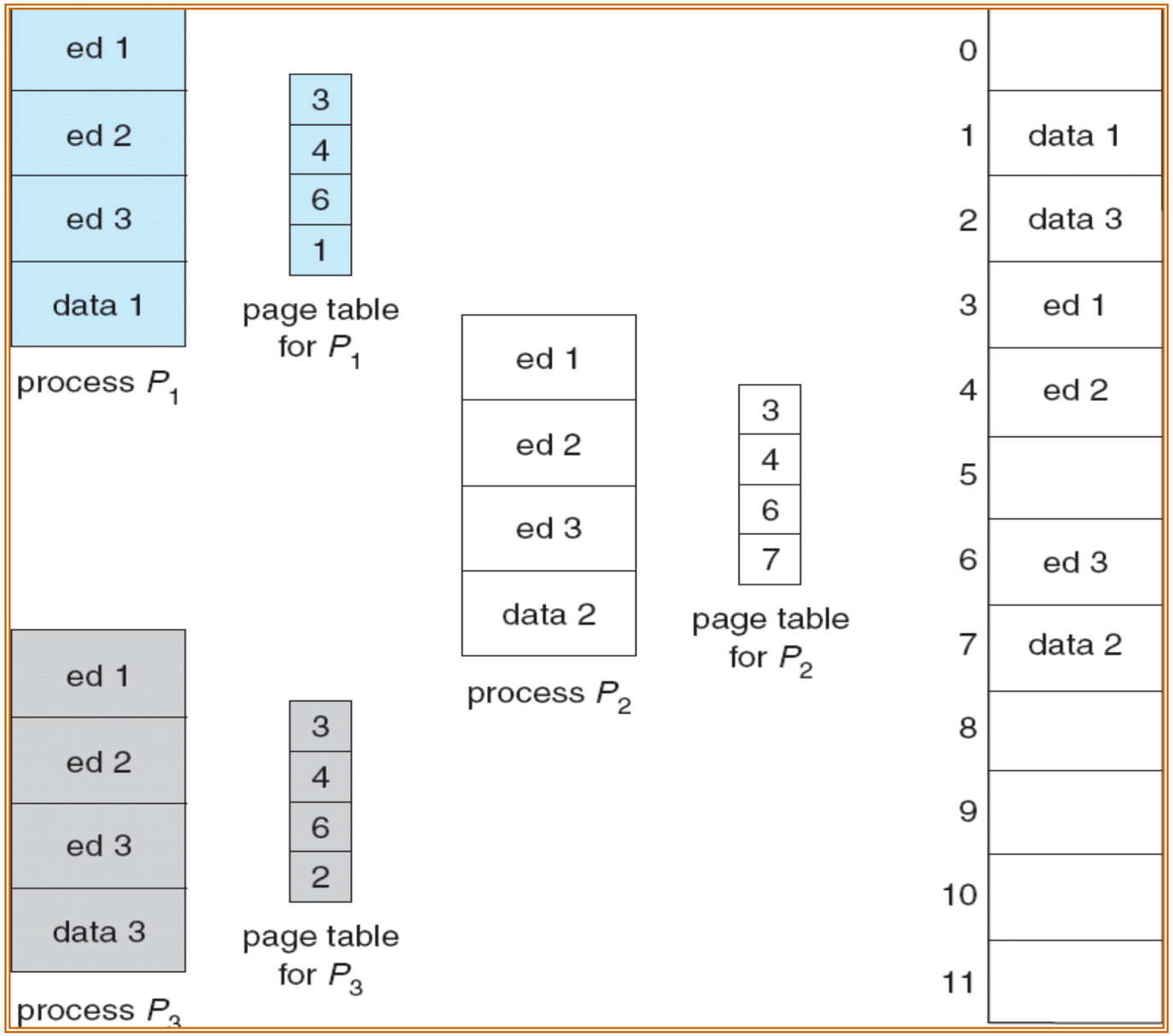
- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

الفكرة: بدل ما كل برنامج يحتفظ بنسخة من نفس الكود → نشارك نسخة وحدة 💡
1. Shared Code (Read-Only)
كود واحد يُستخدم من عدة برامج
لازم يكون:
read-only
reentrant (ما يتغير أثناء التنفيذ)
🔥 مثال:
برامج كثيرة تستخدم:
text editor
compiler
👉 بدل ما كل برنامج يحفظ نسخة:
كلهم يشيرون لنفس الصفحة في الذاكرة
✔ يوفر memory
✔ أسرع
⚠️ شرط مهم:
لازم الكود يكون في نفس المكان (logical address) لكل العمليات
2. Private Code & Data
هذي الأشياء ما تنشارك:
المتغيرات
البيانات
بعض أجزاء الكود الخاصة
👉 كل process لها نسخة خاصة



# Shared Pages Example





# Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

لما يكبر النظام، page table تصير ضخمة 😵‍💫
ففي طرق لتنظيمها:
1. Hierarchical Paging (متعدد المستويات)
بدل جدول واحد كبير:
نقسمه إلى عدة مستويات
👉 مثال:
Page directory
Page tables
✔ يوفر memory
✔ ما نحمل إلا الجزء اللي نحتاجه
2. Hashed Page Table
يستخدم hash function:
نحول virtual page → index
نلقى الصفحة بسرعة
👉 مفيد في:
الأنظمة الكبيرة (64-bit)
3. Inverted Page Table
بدل:
جدول لكل process ❌
نسوي:
جدول واحد للنظام كله ✅
كيف يشتغل؟
كل entry يمثل frame في الذاكرة
ويخزن:
أي process
أي page
الميزة:
✔ يوفر memory بشكل كبير
العيب:
❌ البحث أصعب (يحتاج hashing)





# Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

🧠 الفكرة الأساسية

في الأنظمة الكبيرة:

الـ logical address space يكون ضخم 😵‍💫

وبالتالي:

يصير حجمه كبير جدًا page table 👉

الحل؟

👉 نقسمه إلى عدة مستويات (Hierarchical Paging)

📚 Two-Level Page Table (أبسط شكل)

بدل ما يكون عندنا:

جدول واحد كبير ❌

نخليه:

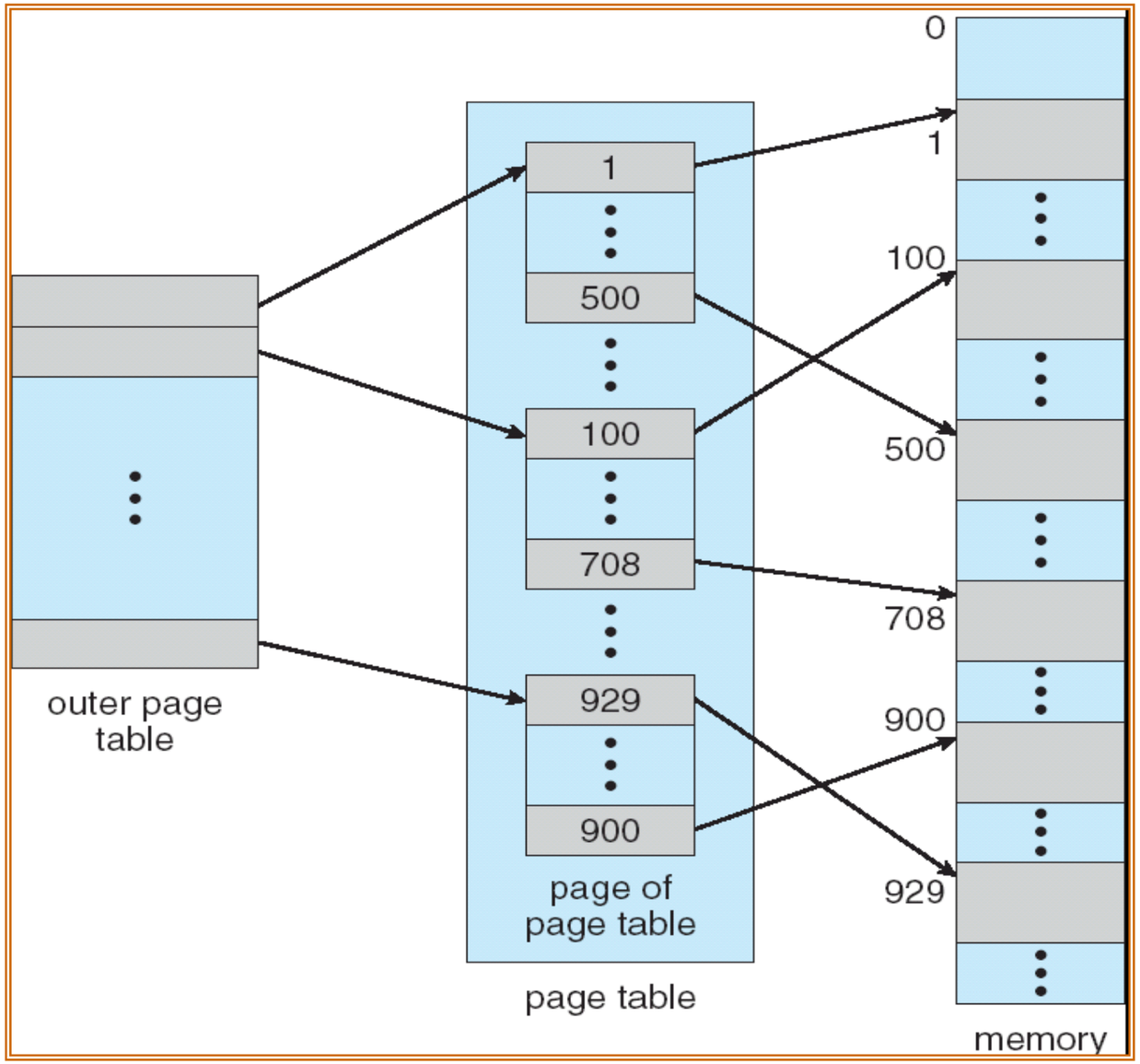
جدول أول (Page Directory)

يشير إلى

جداول ثانية (Page Tables) ✅



# Two-Level Page-Table Scheme





# Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

| page number | | page offset |
|---|---|---|
| $p_i$ | $p_2$ | $d$ |
| 12 | 10 | 10 |

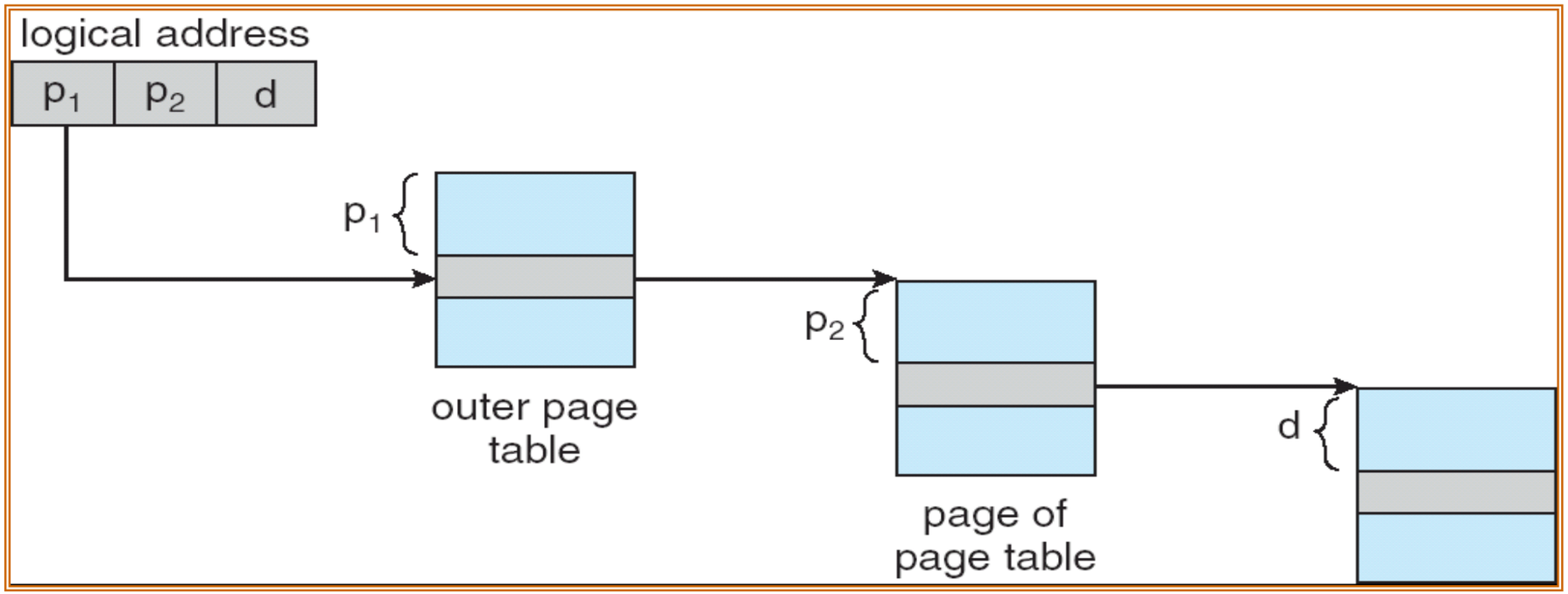
for example 02 100 05

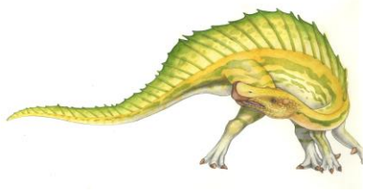
where $p_i$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table





# Address-Translation Scheme



# Three-level Paging Scheme

| outer page | inner page | offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 42 | 10 | 12 |

| 2nd outer page | outer page | inner page | offset |
|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $d$ |
| 32 | 10 | 10 | 12 |



# Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

الفكرة:

بدل البحث المباشر:

👉 نستخدم hash function

كيف يشتغل؟

نأخذ virtual page number

نطبّق عليه hash

نروح لمكان معين في table

نلقى chain (linked list)

بعدها:

نقارن القيم داخل الـ chain

إذا لقينا ✅ match

👉 نطلع frame number

متى نستخدمه؟

✔ في الأنظمة الكبيرة (مثل 64-bit)

لأن page table يكون ضخم جدًا

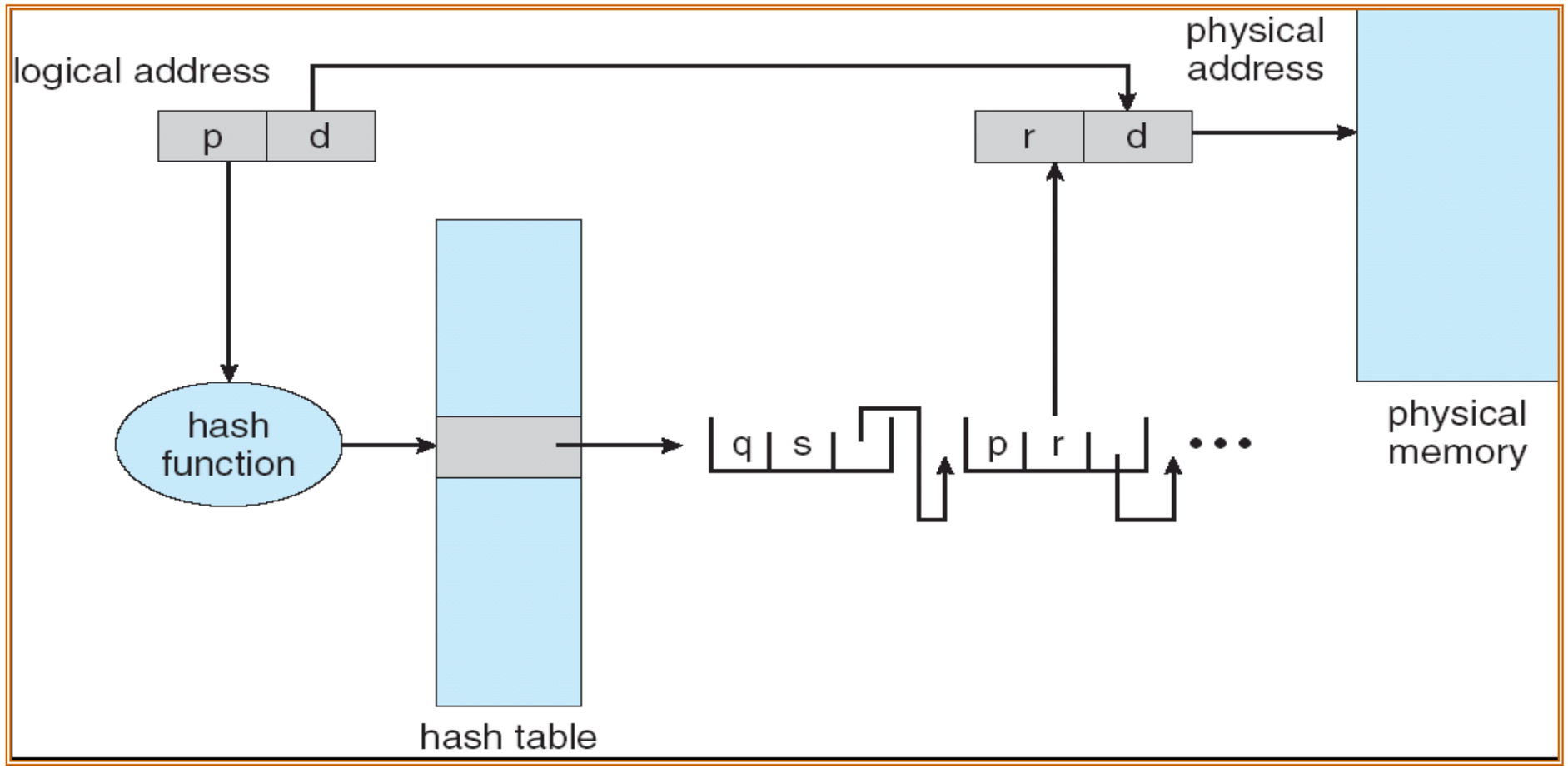
العيب:

❌ ممكن يكون في أكثر من عنصر (collision)

👉 لازم نبحث داخل chain



# Hashed Page Table





# Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

**بدل:**

**جدول لكل process ❌**

**نسوي:**

**جدول واحد لكل النظام ✅**

**كيف؟**

**كل entry يمثل frame في الذاكرة**

**ويخزن:**

**virtual page**

**process ID**

**المميزات:**

**✔ يوفر memory بشكل كبير**

**العيوب:**

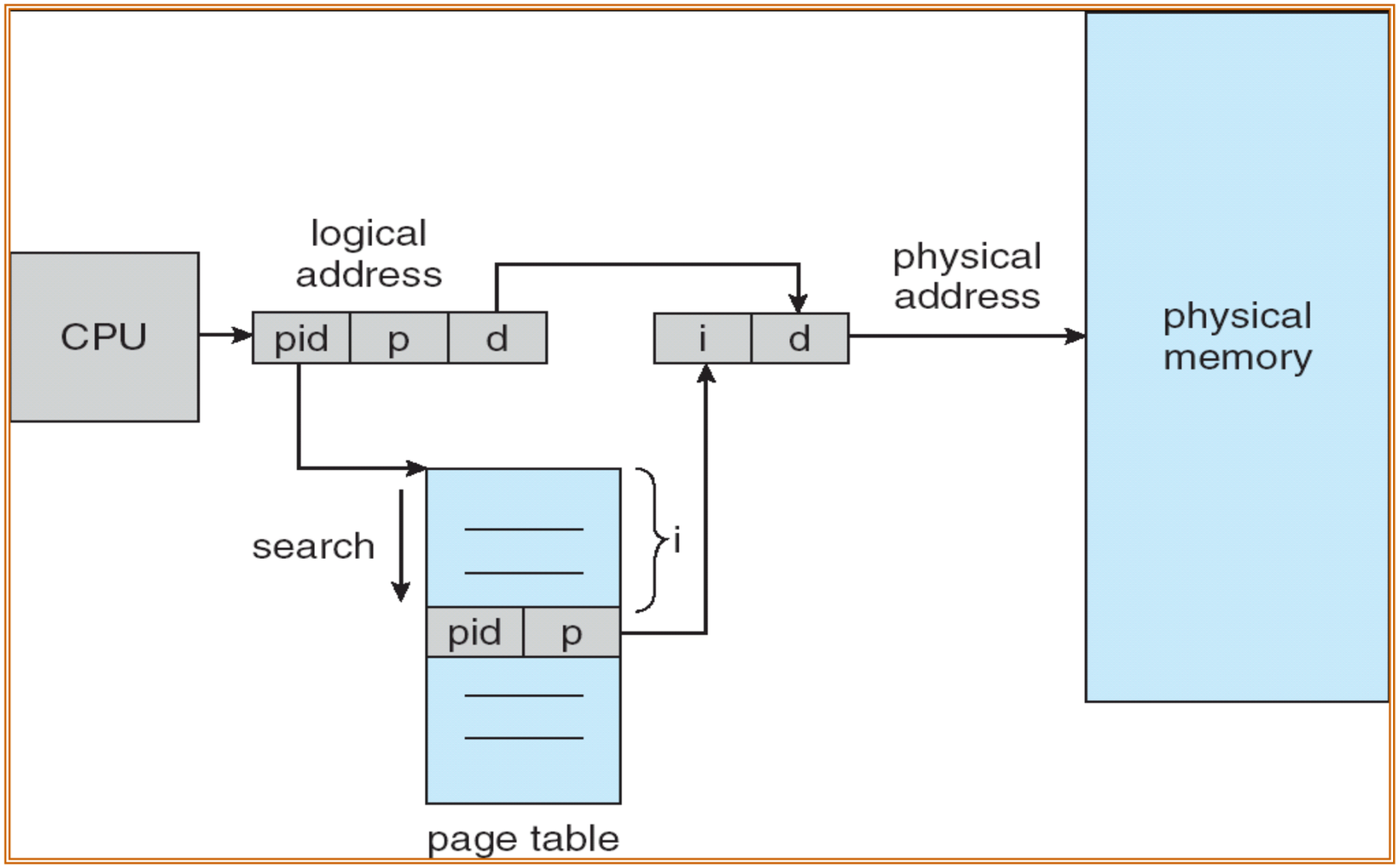
**❌ البحث بطيء**

**👉 لأنك تدور على الصفحة**

**✔ الحل: نستخدم hashing**



# Inverted Page Table Architecture





# Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

**الفكرة:**

**بدل ما نقسم الذاكرة "صفحات" فقط**

**👉 نقسمها حسب منطق البرنامج**

**البرنامج يتكوّن من Segments:**

**main program**

**functions**

**variables**

**stack**

**arrays**

**🎯 مثال واقعي:**

**برنامج:**

**main()**

**function A()**

**function B()**

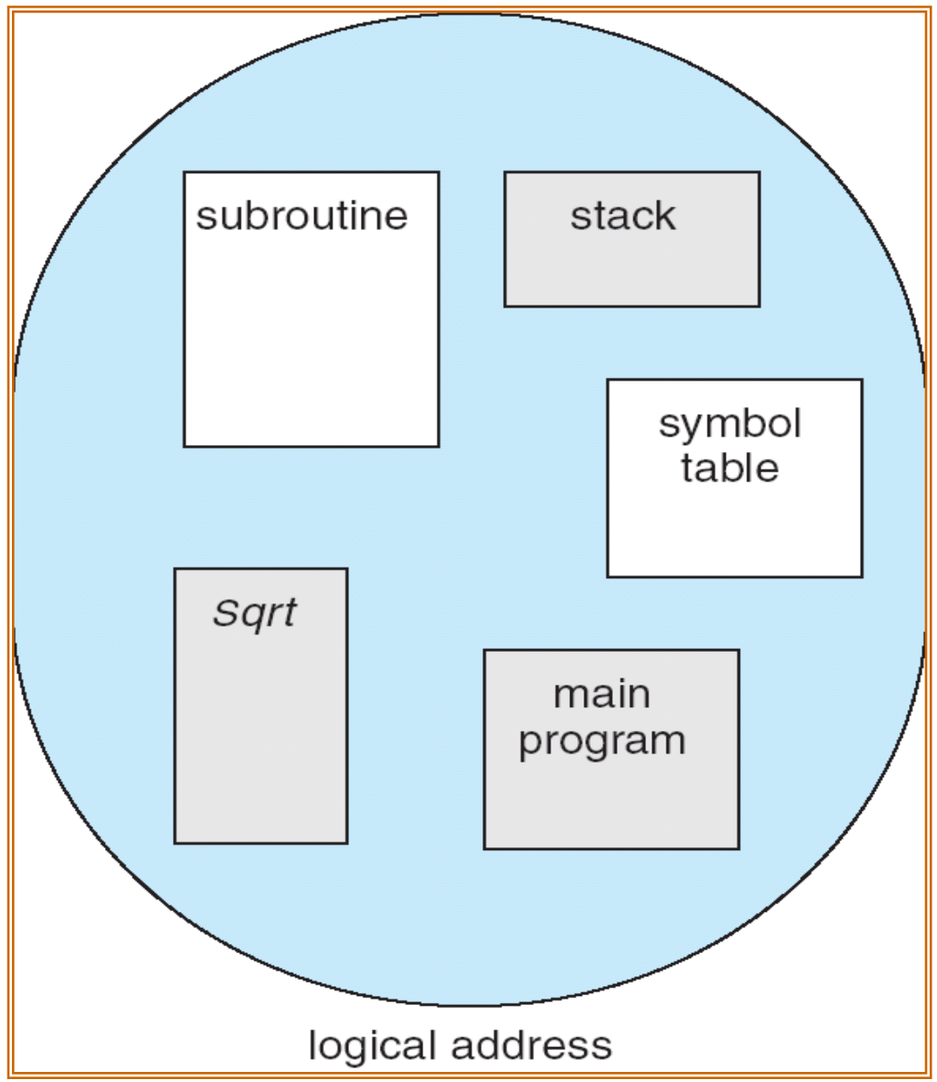
**array X**

**👉 كل واحد = segment مستقل**



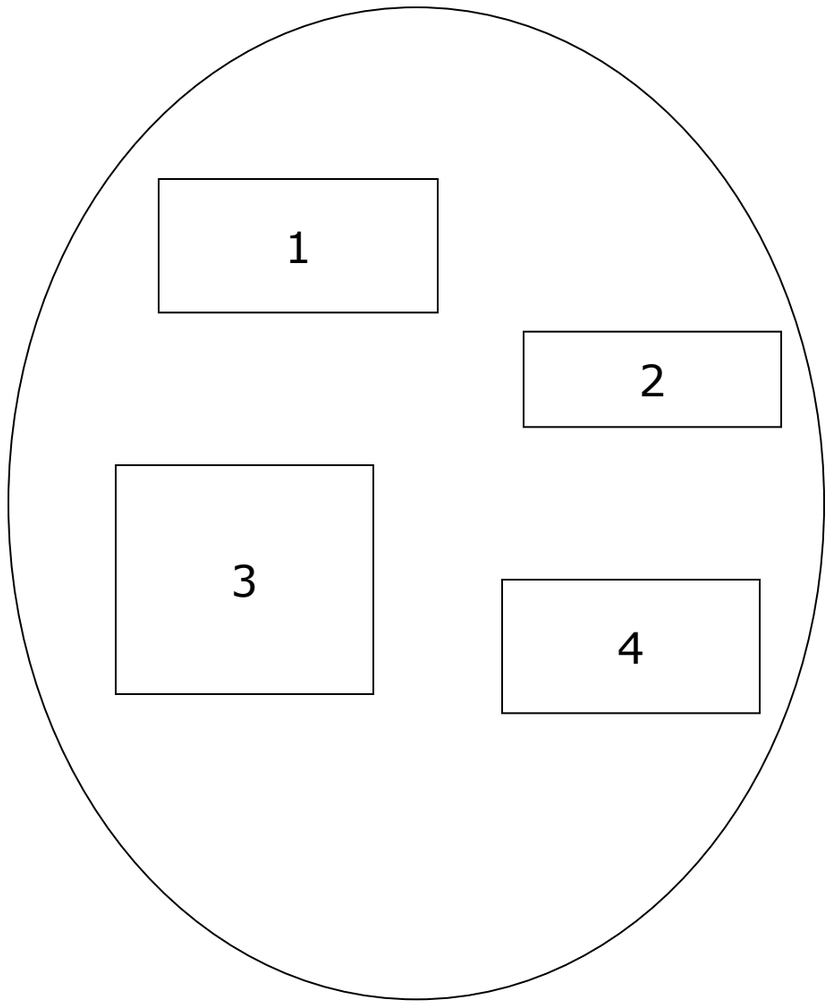


# User's View of a Program

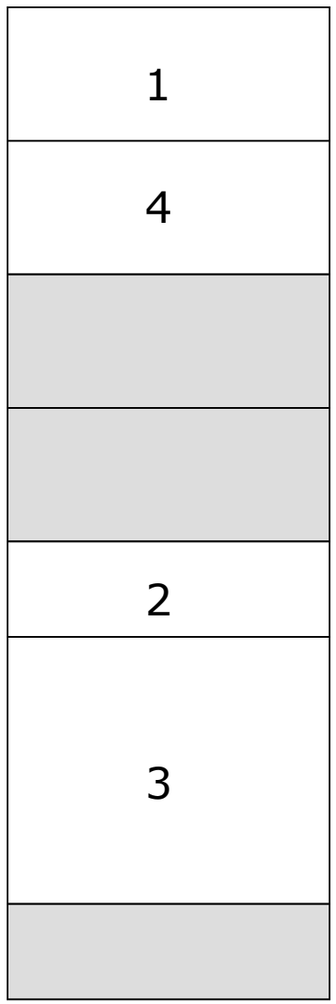





# Logical View of Segmentation




user space




physical memory space



# Segmentation Architecture

بدل ما نقسم الذاكرة صفحات متساوية (Paging)
👉 هنا نقسمها حسب أجزاء البرنامج المنطقية
🔷 شكل العنوان المنطقي
العنوان ما يكون رقم واحد، بل زوج:
Logical Address=⟨s,d⟩
المعنى:
رقم السيجمنت → s (segment number)
الموقع داخل السيجمنت → d (offset)
📊 ثانيًا: Segment Table
كل برنامج عنده segment table خاص فيه
كل entry يحتوي:
1. Base
بداية السيجمنت في الذاكرة الفعلية
2. Limit
طول السيجمنت
🧭 كيف يتم الوصول؟
الخطوات:
ناخذ segment number (s)
نروح للـ segment table
نجيب:
base
limit

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

1. STBR (Segment Table Base Register)
يشير إلى مكان الـ segment table في الذاكرة
2. STLR (Segment Table Length Register)
عدد السيجمنتات
🔥 تحقق مهم:
إذا:
s<STLR
✅ صحيح





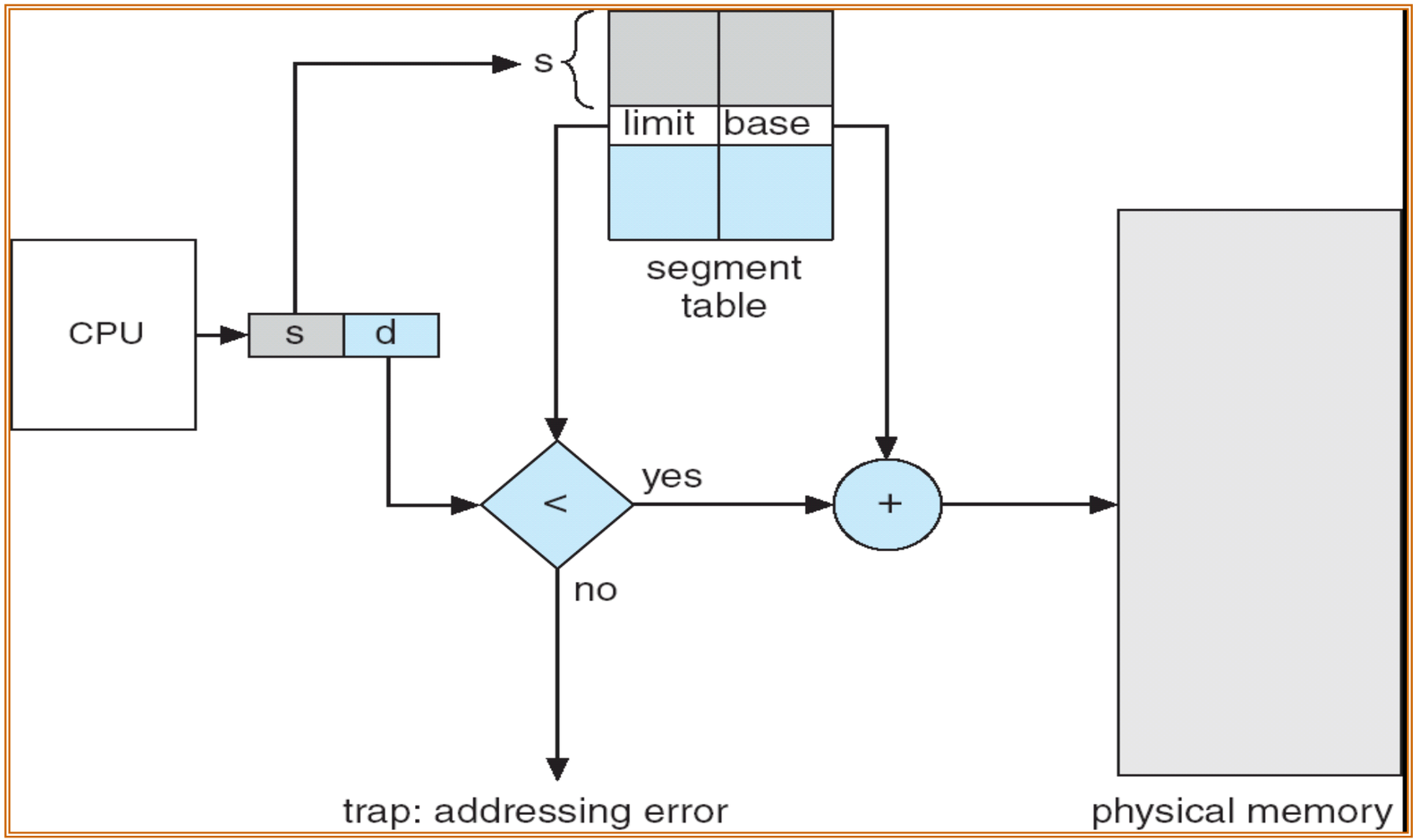
# Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Protection في Segmentation
كل segment له:
1. Validation Bit
❌ غير صالح segment → 0
✅ صالح → 1
2. Permissions
read
write
execute
🎯 ميزة قوية:
👉 الحماية تكون على مستوى segment كامل
مثال:
الكود → read + execute فقط
البيانات → read + write
🔗 Sharing
ممكن نشارك segment كامل
مثل:
shared libraries
code
⚠️ مشكلة Segmentation
لأن السيجمنتات:
أحجامها مختلفة !
👉 يصير عندنا مشكلة:
Dynamic Memory Allocation
fragmentation (فراغات في الذاكرة)

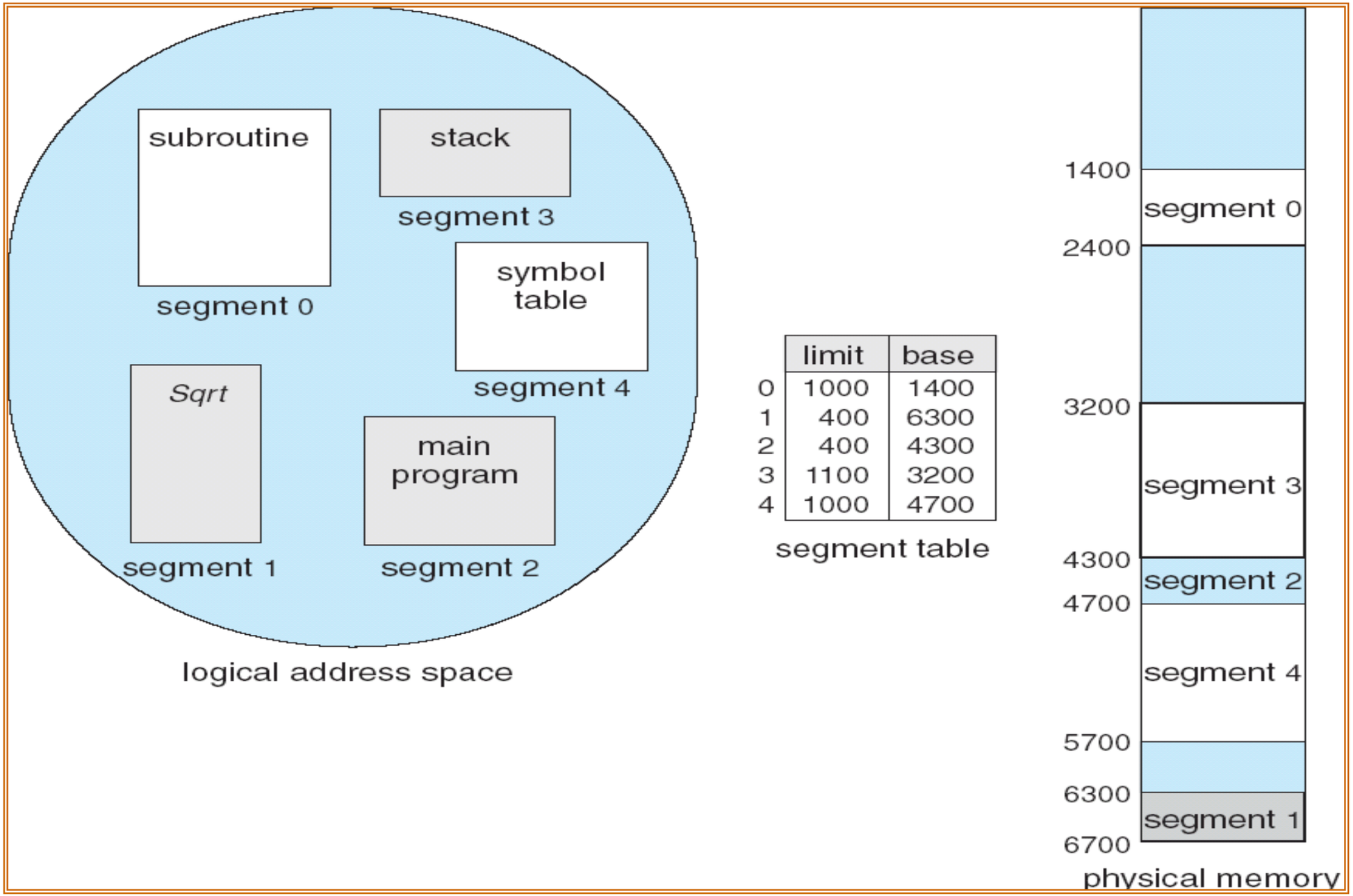


# Segmentation Hardware




**The condition: segment number *s* is accepted if *s* < segment table length**

# Example of Segmentation





# Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of **MMU**

مثال حقيقي — Intel Pentium
هذا مهم لأنه يربط بين كل المفاهيم 👇
كيف يشتغل؟
1. CPU يولد:
Logical Address
2. يروح لـ Segmentation Unit
👉 يحول:
Logical Address→Linear Address
3. بعدها يروح لـ Paging Unit
👉 يحول:
Linear Address→Physical Address
🎯 الفكرة:
Segmentation → ينظم البرنامج منطقيًا
Paging → ينظم الذاكرة فعليًا
🔥 النتيجة:
👉 الاثنين مع بعض = MMU (Memory Management Unit)



# Logical to Physical Address Translation in Pentium




| page number | | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 10 | 10 | 12 |