CH.9: Main Memory

إذا الكمبيوتر ما عنده main memory معناها هو ما بقدر يشغل البرامج، لأنه إذا بدنا نشغل البرنامج بالـ CPU لازم يجيب بالأول على الـ main memory عشان يتم عمل العمليات وتنفيذها بالسي بي يو بعدين.

* Background

السي بي يو بتقدر توصل بس للـ registers و الـ main memory

- Program must be brought into memory and placed within a process for it to be run.
- CPU can access registers & Main Memory directly.
- Memory unit sees: ① adresses.
  ② Read requests & addresses.
  ③ data.
  ④ write requests.
- Register access in one CPU cycle or less
- Main memory takes many cycles causing a stall.

إذا الـ CPU بده داتا من الميموري فالموضوع رح ياخذ بس cycle وحدة، أما إذا بده يوصل للميموري رح يوخذ أكتر من cycle فبالتالي رح يضيع وقت ويتسبب بمشكلة.

الـ stall معناها ✋ أو هي الحالة اللي البروسس بضيع فيها وقت طويل وهو بستنى فيه داتا اللي بده يجيبها من الميموري.

عشان هاي المشكلة تحل حطوا الـ Cashe بين الميموري والـ CPU.

- Cashe sits between main memory and CPU registers.

* Base and Limit Registers

عشان في عدة برامج في الـ RAM محكم هاد الشي بسبب إنه البرامج أو العمليات محكمة تفتقر للسيكيورتي، احنا بدنا شي هاد الشي، يعني بدنا العملية بس تقدر توصل للـ addresses تابعتها بس، يعني توصل لعناوين ما تخصها، فلازم يكون عندها شي محصورة فيه.

- A pair of Base and Limit Registers define the logical address space.

⇒ Base Register: Specifies the smallest legal physical memory address.

⇒ Limit Register: Specifies the size of the range.

اي بمجرد اللوجيكال آدرس هو البيس و اللميت رجيستر اللي بتم تحليلهم عن طريق الـ OS طبعاً، البيس رجستر هو أول عنوان (أو أصغر) بتقدر البروسس تستخدمه أو تكون موجودة فيه، أما اللميت رجستر فهو بحدد كم آدرس مسموحلها توصل (النهاية - البداية).

السي بي يو بضل تيأكد إنه ما في بروسس حاولت توصل آدرس بروسس ثانية.

* Hardware Address Protection

⇐ كيف الـ CPU بتأكد؟

س- بيجيه الأدرس بتأكد هل هو أكبر منه أو يساوي الـ base؟ إذا لا بعطي error ، إذا آه بشوف هل هو أصغر من مجموع الـ base والـ limit ، إذا لا بعطي error ، إذا آه بخليه يفوت على الميموري.

* Address Binding

⇐ في أنظمة التشغيل الحديثة ، في أكثر من بروسس وحدة بتم تنفيذهم بنفس الوقت ، عشان ما يصير لخبطة بالعناوين أو البروسس توخذ محل محجوز أصلاً لبروسس ثانية ، بلزمنا شي ينظم ويبين هاي العناوين الي للعمليات لما يتم تحميلها

⇐ بالعادة عنا ٣ أنواع من العناوين حسب فترة حياة العملية ، وهي:

① Source code adresses (variable names as example).

② Compile code addresses (Relocatable addresses) (بتعتمد على أدرس البداية)

③ Linker or loader addresses (Absolute addresses) (الأدرس بالزبط)

* Binding of Instructions and Data to Memory

. Question: What are the different stages in which address binding can occur?

① Compile time: Absolute code can be generated If memory location known a priori.

⇐ إذا كان معروف المكان في الميموري قبل كل شي (أثناء عملية الكومبايلنغ) بتم حجز الموقع ، طبعاً إذا المكان بظل متاح لازم نعمل compile للبرنامج كمان مرة.

② Load time: Relocatable code must be generated If it's not known at compile time.

⇐ إذا ما انعرف مكان البروسس في الميموري ، لازم الكومبايلر يعمل relocatable كود ويحطه بالميموري عند أي نقطة بداية وبنقدر نوصله عن طريق الاستعانة بهاي النقطة.

③ Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

⇐ إذا العملية بنقدر ننقلها من مكان للثاني أثناء عملية التنفيذ ، بتأخر حجز العناوين لوقت التنفيذ.

⇐ آخر نوع لازم يكون موجود أشياء زي Base & limit registers

* Multistep Processing of a User Program

⇐ في رسمة بالسلايد تطلعوا عليها بعيش وقت أرسمها (دايجي الحمى).

* Logical vs. Physical Address Space

⇐ تم ابتكار مفهوم الـ Logical address space ليتم استخدام الذاكرة بشكل فعّال.

. Question: What are the logical memory address and the Physical memory address?

(Virtual Address) ⇒ Logical Address: Addresses generated by the CPU.

⇒ Physical Address: Addresses seen by the memory unit.

- ↳ Same at: compile time and load time.
- ↳ different at: execution time

- Logical address space: The set of all logical addresses generated by a program.
- Physical address space: The set of all physical addresses generated by a program.

* Memory-Management Unit (MMU)

. Question: What is the MMU?

It's a hardware device that maps the logical address to physical address at run time.

- We consider a simple scheme where we have the value in the relocation register (The starting address of the process) then we add the value of this address to every address generated by the CPU.
- To get the logical address we will add the logical address with the relocation address value and then the final value will be the physical address in the main memory.

⇐ الفكرة هون إنه بكون عنا ريلوكيشن رجستر بنجمع قيمته لقيمة اللوجيكال أدرس بنطلع عنا قيمة الفيزيكال أدرس

- Base register here called relocation register.
- User programs deals with logical addresses only.

* Dynamic relocation using a relocation register

⇐ الروتين هو مجموعة أكواد معينة لتنفيذ شغلة بسيطة (بهاد المثال الروتين هو إنه يتم إيجاد الفيزيكال أدرس)

⇐ هاد الروتين بوفر استخدام جيد للذاكرة عشان بتم استخدامه إلا إذا تم استدعاءه بالإضافة إلى إنه إذا صار غير مستخدم ما بتم تحميله بالمرة.

⇐ ما بكون في حاجة لتدخل النظام هون بس أوقات بساعد عن طريق التزويد بالمكتبات.

* Dynamic Linking

. Static linking: system libraries and program code combined by the loader into the binary program image.

. Dynamic linking: linking postponed until execution time.

⇐ الستاتك لنكنغ بصير فيه اللايبراريز جزء من الباينري كود وما بكون في داعي نناديهم لأنهم موجودين بالبرنامج.

⇐ الداينمك لنكنغ المكتبات فيه ما بتكون مربوطة مع البرنامج نفسه وبتم ربطها بس وقت تنفيذ البرنامج.

. Stub: small piece of code used to locate the appropriate memory-resident library routine.

⇐ وظيفة الـ stub إنها تلاقي وين بنقدر نلاقي هاي المكتبات يعني بكون عنده معلومات عنها أو بتواصل مع الـ OS عشان تجيب هاي المكتبات.

⇐ الـ stub بتبدل حالها مع عنوان الروتين بعدين بتنفذ الروتين، الـ OS بتأكد إذا الروتين بالبروسس ميموري أدرس وإذا ما كان موجود بضيفه.

. Dynamic linking is useful for libraries ⇒ Also known as shared libraries.

* Swapping

Swapping: process swapped out temporarily to backing a backing store, and then brought back into memory for continued execution

⇐ إذا صار في نقص بالمساحة أحياناً تم عمل swap وبتم نقل البروسس لـ backing store عشان يتم خلق شوية مساحة زيادة لأي سبب، بعدين برجع يرجع هاي البروسس.

Backing store: Fast disk large enough to accommodate copes of all memory images for all users

⇐ عادةً الباكنغ ستور هو جزء من الهارد ديسك بيتم استخدامه قبل نظام التشغيل
عشان يتم تحميل بعض البرامج عليه من الميموري ويتم استخدامه كميموري مؤقتة.
⇐ طبعاً بكونه في ready Queue لباقي البروسيز الموجودة على الباكنغ ستور.
⇐ بالعادة ما بيتم استخدام الـ swapping كأنها بتكون عبء إضافي على نظام التشغيل
بتخلي تنفيذ البرامج بيصير أبطأ.

* Context Switch Time including swapping

- Context switch Time can be very high.

ex 100 MB process swapping with transfer rate of 50 MB/sec.

swap out time = 2000 ms = swap in time

⇒ Total context switch = swap in + swap out

= 4000 ms = 4 sec !

- It can be reduced by size of memory swapped
- request_memory() } system calls
- release_memory() }

⇐ في كمان مشاكل للـ swapping زي مشاكل الـ I/O Device ما بنلغيه عبد على الجهاز.
- Standard swapping not used in modern OSs.

* Swapping on Mobile Systems

⇐ بالعادة الـ swapping مش مدعوم للأجهزة الخلوية لأنه مساحة الفلاش ميموري
صغيرة ومحدودة.
⇐ تبعد عمليات ثانية لكل مشكلة بتم استهلاك الذاكرة زي حذف معلومات الـ Read-only
من الذاكرة وبرجع فيلما من الفلاش ميموري إذا لزم الأمر، وغيرها مثل إنهاء التطبيق
من الأوساخ.
⇐ الـ iOS والـ Android بدعموا الـ paging.

* Contiguous Allocation

⇐ الميموري مش بس حكر للـ OS، عشان هيك إله عدد محدود من الريسورس بلازم
يتم استخدامها بشكل معقول

- Main Memory
  - → User processes (In high memory)
  - → Resident OS (In low memory)

⇐ كل جزء من هول العمليات بحتوي على جزء متصل

- Relocation registers used to protect user processes from each other, and from changing OS code and data.
  - Base register ⇒ Value of smallest physical address
  - Limit register ⇒ range of logical addresses (must be less than the limit register)
  - Memory-Management Unit maps logical addresses dynamically.

* Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- Variable-partition sizes for efficiency.

⇐ القطع بكون متغير حسب البروسس الموجودة، مش دايمًا بحجم بروسس معين

- Hole : Block of available memory

⇐ لما توصل عملية جديدة بيتم تخصيص جزء من الذاكرة إلها (بكون مثقب باسمه (Hole)) عشان يتوضعوا وتشتغل فيه.

- Operating system maintains information about:
  1) Allocated partitions.
  2) Free partitions

* Dynamic Storage-Allocation Problem

⇐ أحيانًا تصير مشكلة إذا كانت البروسس موزعة بطريقة مش زي ما بدنا مثلًا صير معنا كتير Holes أحجام صغيرة ما بتتسع الـ processes اللي حجمها كبير

Question: How to solve the problem of dynamic storage allocation?

* First-fit
Allocate the first hole that is big enough.

* Best-Fit
Allocate the smallest hole that is big enough.

⇒ must search entire list.

* Worst-Fit
Allocate the largest hole

→ must search entire list.

* Fragmentation

Question: What are types of fragmentation?

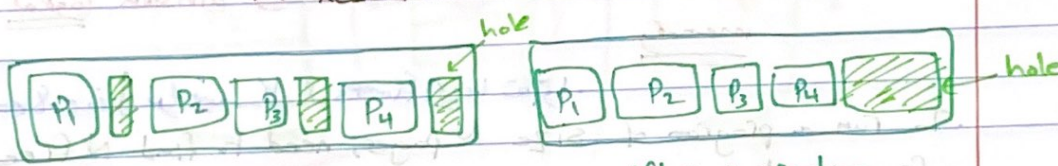
⇒ External Fragmentation: total memory space exists to satisfy a request, but is not contiguous.

⇒ Internal Fragmentation: Allocated memory may be slightly larger than requested memory resulted in a size difference that is memory internal to a partition but is not used.

* المشكلة هون إنه بصير في عنا مساحة مش قادرين نستخدمها لأشي.

Question: How to reduce external fragmentation?

By [Compaction]: shuffle memory contents to place all free memory together in one large block.
↳ possible if only relocation is dynamic & done in execution time.



before compaction — After compaction

* Segmentation (مش مطلوب بالعيد)

. Memory-management scheme that supports user view of memory.

⇐ بصير البرنامج مكون من segments (قطع) و بسمح بتحويل البرنامج من جزء إلى أجزاء يكسر, ويضعها في مناطق غير متجاورة من الذاكرة.

* Segmentation Architecture (مش مطلوب بالعيد)

. logical address consists of two tuple: <segment-number, offset>.

. segment table: maps two-dimensional physical addresses; each table entry has: ① base: starting physical address where the segments reside in memory.
② limit: specifies the length of the segment

- Segment-table base register (STBR): points to the segment table's location in memory. (يدلّ على موقع الجدول الخاص بالقطع في الميموري)
- Segment-table ~~[illegible]~~ lenght register (STLR): indicates number of segments used by a program. (يدلّ على عدد القطع [illegible])

⇐ طبعاً حاي القطع محمية من الأشياء اللي مش موقعها تدخلها عن طريق وجود بتات بتحميها.

* Paging

⇐ الفكرة من ال paging إنه ممكن أعطي عناوين مختلفة وغير متصلة مع بعضها للبروسس.

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available. ⇐ طالما المكان متاح بنحط البروسس فيه عادي بدون شروط.
- Frames: fixed-sized blocks caused by division physical memory. (Size is power of 2, between 512 bytes and 16 Mbyte)
- Pages: Blocks of same size caused by division logical memory.

⇐ لازم نظل مراقبين كل ال frames اللي ما تعبوا.

- To run a program of size N pages, need to find N free frames and load program.
- Page table: to translate logical to physical addresses.

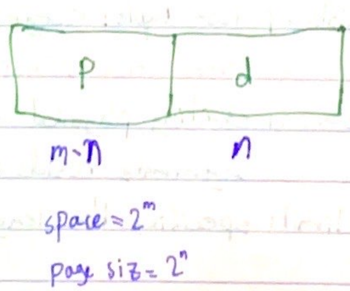
⇐ بنضل عندي ال Backing store بكون مقسّم ل Pages.

- Still have internal fragmentation

* Address Translation Scheme

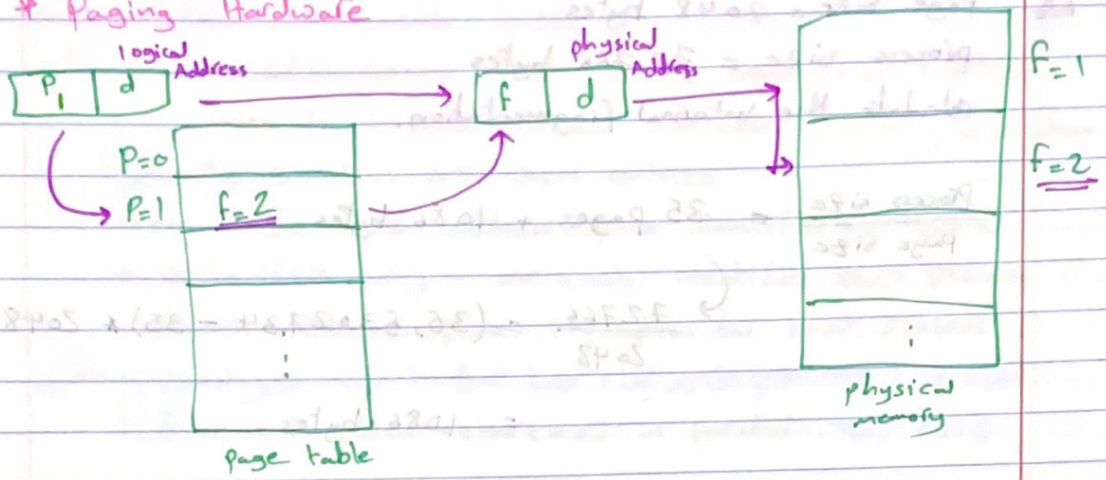
Address generated by CPU is divided into: Page number (p)
Page offset (d)

يُستخدم ⇐ Page number: used as an index into a page table which contains base address of each page in physical memory. (كفهرس لل page table اللي بحتوي على ال base address.)

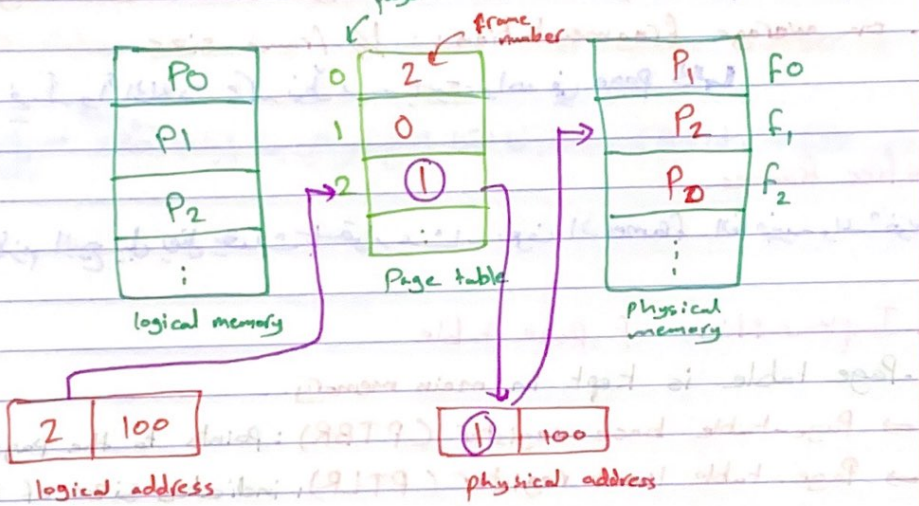


⇒ Page offset: combined with base address to define the physical memory address that is sent to the memory unit.

* Paging Hardware

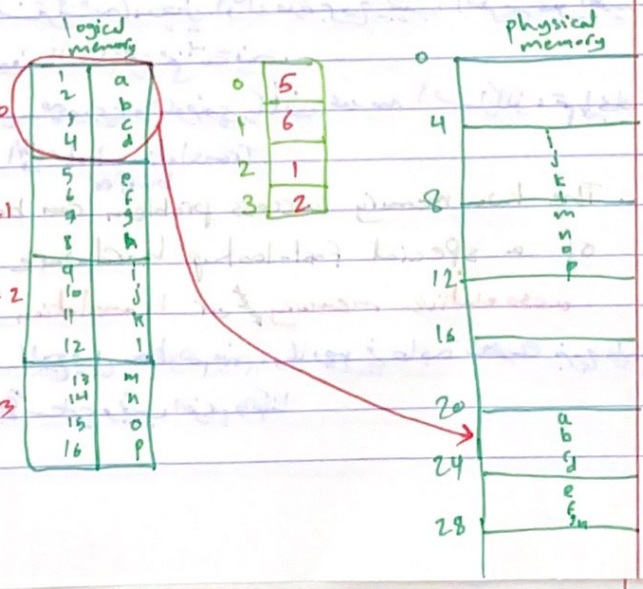


بطريقة مبسطة أكثر :



* Paging Example

- Page size = 4 byte = frame size
- memory size = 32 byte = 8 pages
- page = $\frac{\text{logical address}}{4}$
- offset = logical % 4



ex Page size = 2048 bytes
process size = 72 766 bytes
calculate the internal fragmentation.

$\frac{\text{Process size}}{\text{Page size}}$ = 35 pages + 1086 bytes

$\frac{72766}{2048} = (35,5302734 - 35) * 2048$

= 1086 bytes

- Worst case fragmentation : ~~1 byte~~ frame size - 1 byte
- on average fragmentation : ½ frame size

⇐ في أسوأ الحالات ممكن نغلط بـ بايت واحد في كل الـ page

* free frames

⇐ لازم البيج تيبل يضل يتحدث باستمرار عشان نعرف الـ frames الفاضين والمشغولين

* Implementation of page table

- Page table is kept in main memory.

⇒ Page-table base register (PTBR) : points to the page table

⇒ Page-table limit register (PTLR) : indicates size of the page table

⇐ بهاد النموذج (paging) بلزمنا وصولين للذاكرة (واحد عشان البيج تيبل وواحد عشان نجيب الداتا أو الانستركشن)، يعني أول مرة بروح يجيب الفريم نمبر بعدين بعد ما يجيبه برجع عشان يشوف الداتا الي مخزنة فيه.

⇐ بدل ما يروح أكثر من مرة للميموري والموضوع يكلف عدد من السايكلز، تم إيجاد شيء اسمه associative memory أو Translation look-aside buffer.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cashe called associative memory or translation look-aside buffers (TLBs).

⇐ فكرتهم إنهم بياخدوا جزء من البيج تيبل بحطوهم فيهم، ولما يصير في حاجة لمعلومة بروح عليهم، إذا مالقينا ما بنضطر نروح للبيج تيبل الأساسي ونجيب الفريم وهكذا

* Translation Look-aside Buffer

TLB: a CPU cache that memory management hardware uses to improve virtual addresses (speed) translation

- Typically small: 64-1024 entries
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry - uniquely identifies each process to provide address-space protection for that process.
- TLB is associative - searched in parallel.

* Effective Access Time

- Hit ratio: percentage of times that a page number is found in the TLB.

ex Hit ratio = 80%
time to find in TLB = 10 ns
time to access memory = 20 ns

$$EAT = (\alpha * \text{time to find in TLB}) + (1-\alpha) * \text{time to access memory}$$
$$= (0.8 \times 10) + (0.2 \times 20) = 12 \text{ ns}$$

* Memory Protection

- Implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed.
- Valid-Invalid bit attached to each entry in the page table.

* Shared Pages

⇒ Shared code

⇐ بتكون في نسخة للقراءة فقط بتتم مشاركتها بين البروسس وهي مفيدة للتواصل بين البروسس بين بعضها (واذا كانت صح الكتابة عليها).

⇒ Private code and data

⇐ كل بروسس منها نسخة خاصة فيها من الكود والبيانات.

* Structure of the page table

⇐ إذا بدنا نستخدم الطريقة العادية اللي ذكرناها بح يكون حجم البيج تيبل كتير كبير يعني لو مثلاً هنا ٣٢ بت لو فيكال آدرس سبيس و حجم كل بيج ٤ كيلو بايت مرح يضطاج تقريباً $\frac{2^{32}}{2^{12}} = 2^{32}$ إنتري مليون للبيج تيبل.

⇐ هنا طورو أو تسللات معينة عشان نرتب البيج تيبل بطريقة عاملة وما بكلف كتير

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

* Hierarchal Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique → two-level page table.
- Then page the page table.

⇐ بقسم البيج تيبل نفسه لأجزاء، وهاي الأجزاء بنعاملها زي بيج تيبل عادي هنا.

* Two-level Paging example

20-bit page number + 12-bit page offset

10-bit outer page table — 10-bit inner page table

⇒ Known as forward-mapped page table

$K = 2^{10}$
$M = 2^{20}$
$G = 2^{30}$

* 64-bit logical Address space

⇐ برضو طريقة الـ two-level paging هاي مش فعّالة كثير، إذا كان حجم البيج 4KB (2^{12})

⇐

p_1	p_2	d
42	10	12

⇐ 2^{10} 4KB = Inner page entries

⇐ عشان نحسّن الموضوع شوي بنضيف كمان levels من البيج تيبل بس رح تقل نسبة الأكسس للميموري عادية كل ما زاد عدد الـ levels

* 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.

⇐ فكرته بالزبط زي الهاش تيبل بالداتا ستركشر

- Each element contains:
  1. The virtual page number
  2. The value of the mapped page frame
  3. Pointer to the next element.

⇐ لما ندوّر على شي معين بنشوف بالسلسلة كلها، إذا لقاها بوخذ رقم الفريم

* Inverted Page Table

⇐ بدل ما نخطّ رقم البيج نخطّ رقم الفريم، وجوا نخط رقم البيج. بكون هيك عمل مواصفة للتخزين في الذاكرة، بس إذا بدنا نشوف رقم البيج رح نوخذ وقت طويل في البحث.

⇐ ممكن نستخدم الهاش تيبل فيه