CH.10: Virtual Memory

الـMain memory بتكفيش نحط فيها كل البرامج عشان هيك عنا ميموري افتراضية مش حقيقية.
في حال كانت الـMain memory ممتلئة رح نشيل بعض البرامج ونوديها للـVirtual memory على أساس ألغوريثمز معينة رح نتعلمها بهذا الشابتر.

. السؤال: ليش أو شو الهدف من إننا نطلع بعض البرامج على أساس ألغوريثمز معينة؟
⇐ حتى نقدر نستعمل الذاكرة بأفضل شكل ممكن لضمان فعاليتها

* Background

في أغلب الحالات ما بلزم يكون البرنامج جميعه موجود في الميموري عشان يتم تنفيذه، وحتى لو لزم يكون جميعه موجود ما بتم تنفيذه جميعه في نفس الوقت، بتم تنفيذه جميعه بأوقات مختلفة خلال عملية المعالجة.
فما في داعي نجيب كل البرنامج ونوخذ مساحة من الميموري و يضل البرنامج يدخل ويطلع بجميع أجزاءه.

- Advantages of execute partially-loaded program:
  1. Program not constrained by limits of physical memory.
  2. Each program takes less memory while running
     ⇒ more programs run at the same time.
  3. Increased CPU utilization and throughput.
  4. Less I/O needed to load or swap programs into memory

* Virtual memory

Virtual Memory: separation of user logical memory from physical memory.

⇐ الفصل بين مساحة البرنامج الافتراضية التي يراها المستخدم، ومساحته الفعلية التي يحتاجها الجهاز عند تحميل وتشغيل البرنامج.

- only part of the program needs to be in memory to execution.
- logical address space ≫ physical address space.
- Allows address space to be shared by several processes.
- Allows for more efficient process creation.
- More programs running concurrently.
- Less I/O needed to load or swap process.

- Virtual Address space: logical view of how process is stored in memory.

المساحة الافتراضية بتفترض إنه العنوان الافتراضي بيلش من صفر، وإنه مساحة البرنامج بتكون بشكل مستمر ومتصل، بينما في الواقع البرامج مقسمة لـ pages بحسب شو بتوفر إله الميموري مساحة فاضية للبروسس.

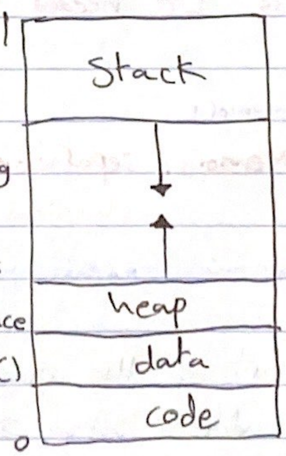
* MMU (Memory Management Unit) must map logical to physical.

الذاكرة الافتراضية بتعطي المستخدم صورة بأنه البرنامج تم تحميله بالكامل على الذاكرة خلال المعالجة، وإنه مخزن بشكل متصل، بينما في الواقع البرنامج مجزأ إلى أجزاء، ولما يتم تحميل البرنامج عشان تتم معالجته بتم تحميل بعض الأجزاء بس، والبعض الآخر بتم وضعه في مساحة على الـ Hard Disk اسمها Backing store.

← نظام التشغيل دايماً بيبحث عن الأجزاء الغير مستخدمة في الميموري بينقلها على الهارد ديسك عشان يتيح مساحة أكبر في تشغيل برامج تانية

- Virtual memory can be implemented via:
  1. Demand paging.
  2. Demand segmentation.

* Virtual Address space



بهذا التصميم بنزيد استخدام الـ Address space وما بنحتاج فيه لذاكرة فعلية

- System libraries shared via mapping into virtual address space.
- shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork() speeding process creation

* Demand Paging

. Bring a page into memory only when it is needed

⇐ طريقة آلية أفضل بكثير منه إننا نجيب البرنامج كامل على الذاكرة، منا بقى نحمل جميع الصفحات. بس بجيب الصفحات اللي بتطلب منه الهارديسك.

↳ Less I/O needed

↳ less memory needed

↳ Faster response (كأنه عدد الصفحات أقل)

↳ more users

. Lazy swapper : never swaps a page into memory unless page will be needed.

⇐ ما رح يجيب الصفحة إلا إذا انطلبت.

. swapper that deals with pages is a pager.

* Basic Concepts

الـ pager بجيب صفحات للذاكرة أول مرة، بحاول يتنبأ بالصفحات اللي رح يرجع يحتاجها قبل ما يرجعها كمان مرة على الهارديسك.

. If pages needed are already memory resident

⇒ No difference from non demand paging (كأنها موجودة أصلاً ما بطلبها)

. If pages needed and not memory resident

⇒ Need to detect and load ~~men~~ page into memory from storage.

* Valid - Invalid Bit

V : in memory

i : not in memory

* كل page table فيه مكان لخانة الـ V/i

⇐ مبدئياً بكونوا كلهم i عشان احنا لسا ما وصلنا لشي على الميموري.

⇐ إذا كانت i ⓐ إما تكون مش موجودة في الميموري (Page fault)

ⓑ مش مسموح يوصلها البروسس (abort process)

* Steps in Handling Page Fault

إذا النظام أجا بدّه ينفذ شي أو صفحة مش موجودة بالذاكرة اي بصير عنا هو الـ Page fault فبصير عنا Trap يعني تعثر للـ OS.

⇐ أول شغلة بعملها الـ OS بتأكد إذا الريفرنس شه موجود (مش موجودة الصفحة أصلاً أم موجودة بس مش في الميموري.

⇐ بس يلاقيها .بجيبها وبحطها في free frame عن طريق ال scheduled disk operation
وبحطها V.

① looks at another table to decide if its invalid. reference or just not in memory
② Find free frame.
③ Swap page into frame via scheduled disk operation.
④ set Validation bit to V.
⑤ Restart the instruction that caused the page fault.

* Aspects of Demand Paging

- Extreme case - start process with no pages in memory.

⇐ هاي حالة خاصة، هي إنه لما البرنامج يبلش مش رح يكون في ولا page تنفذه في الميموري يعني كل أمر يتم تنفيذه عبارة عن page-fault. لحد ما يجيب جميع الصفحات ويبلش ينفذه بشكل طبيعي. هاي الحالة اسمها Pure Demand Paging

* Hardware support needed for demand paging:
  ① Page table with validation bit
  ② Secondary memory
  ③ Instruction restart (لما التعليمات تعمل مشاكل بنعملها restart)

* Free-Frame List

بكون في قائمة بأرقام ال frames الفاضية والي مستعدين يستقبلوا صفحات جديدة أو ريت إيفنت

free-frame list : Pool of free frames for satisfying such requests.

وبالعادة ال OS بخصص frames فاضية مخصوصة للتعامل مع حالات ال Page-fault
⇐ هاي الحالة اسمها Zero-fill-on-demand.

* Stages in Demand Paging

① Trap to the OS.
② Save the user registers & process state.
③ Determine that the interrupt was a page fault.
④ check the page reference was legal and determine the location
⑤ Issue a read from the disk to a free frame:
  ⓐ wait in a que until the read request is serviced.
  ⓑ wait for the device seek / latency time
⑥ while waiting, allocate the CPU to some other user
⑦ receive an interrupt from I/O
⑧ save the registers and process state for the other user.
⑨ Determine that the interrupt was from the disk
⑩ correct page tables
⑪ wait for the CPU to be allocated to this process
⑫ Restore registers, process state, new page table then resume the interrupted instruction.

* Performance of Demand Paging

- Three major activities
  ① service the interrupt
  ② Read the page
  ③ Restart the process

Page Fault Rate $0 \leq p \leq 1.0$
$\Rightarrow p = 0$ (no page faults)
$\Rightarrow p = 1$ (every reference is a fault)

Effective Access Time

$EAT = \underbrace{(1-p) \times \text{memory access}}_{\text{no page fault}} + p(\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$

ex Memory access time = 200 ns
Average page fault service time = 8 ms

$$EAT = (1-p) \times 200 + p\ (8\ msec)$$
$$= (1-p)\,200 + p\,(8 \times 10^6)$$
$$= 200 + 7\,999\,800\,p$$

If $p = 0.001 \Rightarrow EAT = 8200$ ns

* we should minimize the number of page faults.

* Copy-on-write
هي التقنية بتسمح لبرنامجين الاشتراك في نفس الصفحات بدون ما تعمل نسخة لكل واحد فيهم لحد ما يحاول واحد فيهم يعدّل على النسخة، وهنا بتعمل نسخة خاصة فيه يعدّل عليها.
COW allows both parent and child processes to intially share the same pages in memory.

- Vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent.
  - ⇒ Designed to have child call exec()
  - ⇒ very efficient

* What happens if There is no Free Frames?
في الحالة هي بتتم Page replacement عن طريق خوارزمية معينة.
Page replacement: find some page in memory, but not really in use, page it out.
⇐ عليه اختيار الخوارزمية بتعتمد على أقل خوارزمية بعمل Page fault هو اي بستخدمه.

* Page Replacement

⇐ بتم منع الـ over-allocation في الميموري عن طريق تعديل الـ page-fault service routine الي بنستخدمها في الـ page-replacement

⇐ بتم استخدام بت معين عشان نعرف نسخ الصفحة المعدّلة من اللي مش معدّلة.

- Use **modify (dirty) bit** to reduce overhead of page transfers
  - only modified pages are written to disk
- Large virtual memory can be provided on a smaller physical memory.

* Basic Page replacement

① Find the location of desired page — ① البحث عن موقع الصفحة المطلوبة من الديسك

② Find free frame: — ② يلاقي frame فاضي
- there is a free frame? use it — إذا لقاه بنحط الـ page فيه
- No? select a victim — ما لقاه بطلّع واحد
- write victim frame to disk if dirty — بتم نقل محتويات الضحية للديسك

③ Bring the desired page — ③ بنجيب الصفحة الجديدة للـ frame

④ restart the instruction. — ④ بنعمل restart

* Page and Frame Replacement Algorithms

⇐ بنحتاج الـ FRAs عشان نحدد عدد الـ frames المعطية لكل بروسس بحيث ما نعطي كل بروسس frames أكثر مما بحتاج، وبنحدد الي frames بنقدر نبدّلها.

⇐ بنحتاج كمان الـ PRs

الفكرة إنه احنا بنقيم ونختار بناءً على أقل وحدة بتسبب page-fault في البداية لما أول ما توصل البروسس أو أثناء العملية.

Evaluating algorithm by running it on (reference string)

↲ أرقام الـ Pages

* Graph of Page fault Vs. The number of frames

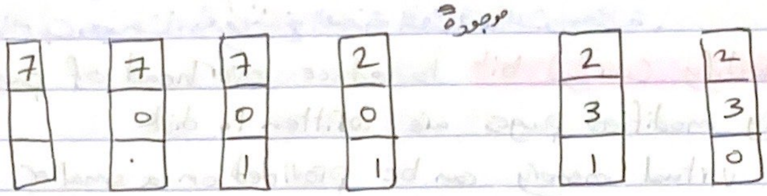
⇐ كل ما زاد عدد الـ frames بتقل نسبة الـ Page fault.

* First-In-First-Out (FIFO) Algorithm

أول بيج دخلت هي أول وحدة بتطلع

7 0 1 2 0 3 0

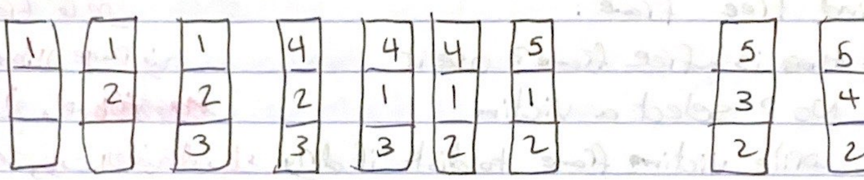
موجودة



ex Reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frame

1 2 3 4 1 2 5 1 2 3 4 5



9 page faults

* adding more frames can cause more page faults
  ↳ Belady's Anomaly

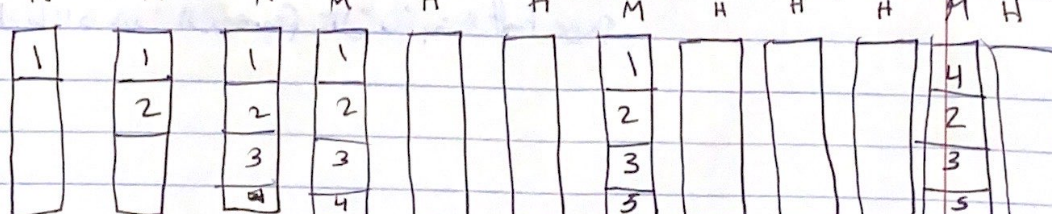
← زيادة عدد الـ frames لا يعني بالضرورة انخفاض نسبة الـ page fault

* Optimal Algorithm

← نطلع البيج الي مش رح نستخدمها لفترة طويلة (طبعاً ما بنقدر نتنبأ بالمستقبل ونعرف شو الي رح ينطلب وشو الي لأ لهيك تطبيق هاد الالغوريتم مستحيل)، بس احنا بنستخدمها عشان نقارن بينها وبين الألغوريتمز الثانية

ex 4-frame example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	2	3	4	1	2	(5)	1	2	3	4	5
M	M	M	M	H	H	M	H	H	H	M	H



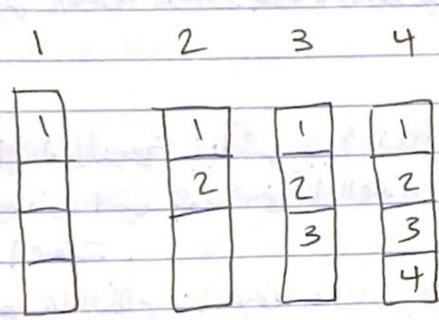
Page-fault = 6 page faults

. used for measuring how well your algorithm performs.

* Least Recently Used (LRU) Algorithm)

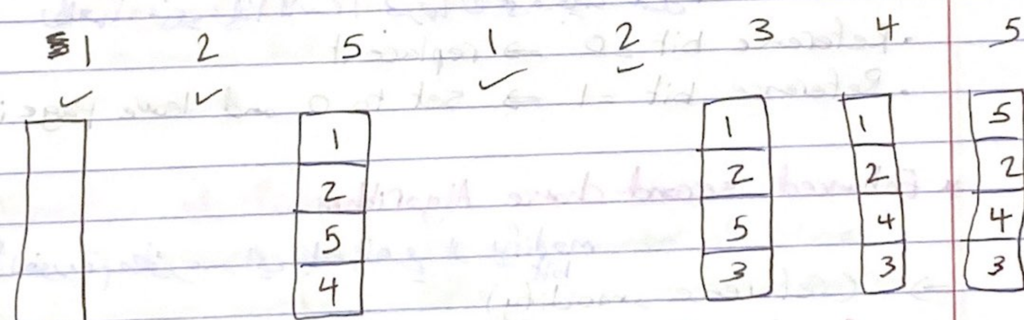
← بتبدل الصفحة الي صارلنا زمان ما استخدمناها (أقل وحدة استخداماً)

ex 4-frame example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Better than FIFO

← هاي الطريقة هي الأكثر استخداماً نظراً لأدائها الجيد.



Page-faults = 8 page faults

⇒ LRU & Optimal don't suffer from Belady's anomaly.

. How to Implement?

① Counter implementation

في هاي الطريقة بنحط عداد لكل صفحة (في جدول خاص يعني) بحط قيمة الساعة في الوقت الي استخدمنا فيها أو طلبنا هاي الصفحة، ولما بدنا نطلع بنطلع الي عندها أقل رقم (يعني أقدم وقت)

② Stack Implementation

بنحفظ أرقام الصفحات في stack ، الصفحة الأقدم بتكون تحت أما الأحدث استخداماً بتكون فوق، يعني لما نستخدم صفحة بنطلعها ونحطها فوق

⇐ هاي الطريقة سريعة بس صعبة بدها ٦ بوينترز يتغيروا كل مرة

* LRU Approximation algorithm

بكون في خانة reference bit بدل على إذا تم طلب هاي الصفحة أو لا، مبدئياً بس الصفحة تكون موجودة لأول مرة بتكون القيمة صفر، لما تنطلب تصير ١

* Second-chance algorithm

هاي الخوارزمية بتبلش بمبدأ الـ FIFO وبكون معها كمان reference bit

⇐ أول اشي بتكون صفر لكل الصفحات، بتم تبديلها لـ ١ لما يتم استخدام هاي الصفحة.

⇐ لما النظام يختار صفحة عشان يبدلها بنفحص الـ reference bit، إذا كانت صفر بنطلعها دغري، إذا كانت ١ بنحولها لصفر وبنخليها تكمل.

- Reference bit = 0 ⇒ replace it
- Reference bit = 1 ⇒ set to 0 and leave page in memory

* Enhanced second-chance Algorithm

ألغوريتم مطوّر عن الي قبله بانه في الـ modify bit

⇒ (reference, modify)

(0,0) neither recently used nor modified (أفضل خيار نطلعه)

(0,1) not recently used but modified (خيار منيح بس لازم نعمل write بالبداية)

(1,0) recently used but clean (غالباً يتم طلبه كمان شوي)

(1,1) recently used & modified (أسوأ خيار للتبديل)

* Counting Algorithms

عداد لكل صفحة فيه عدد قديش مرة تم طلب هاي الصفحة.

① Least Frequently Used (LFU) Algorithms

Replace page with smallest count.

② Most Frequently Used (MFU) Algorithm.

page with smallest count was probably just brought in and has yet to be used.

* Page-Buffering Algorithms

← فكرته إنه لازم دايماً يكون في frames فاضية (يعني ما ننزنقوا بس وقت الـ page-fault، إنه لازم يختاروا الضحية والبديل، وما يكون الوقت المناسب يطلعوها برا، ولازم يكون في قائمة بأرقام الـ modified pages، وطبعاً قبل ما يطلعوها لازم يراقبوا إذا تم استخدامها كمان مرة يعودوا على ضحية ثانية.

1. Keep a pool of free frames, always.
2. Keep list of modified pages.
3. Keep free frame contents intact and note what is in them.

* Applications and page replacement

جميع التطبيقات لا تتعامل مع فكرة الـ page replacement يكون عندها تنبؤات بس أحسن التطبيقات منهم من هالناحية هي الداتابيس.

Raw disk mode: OS can give direct access to the disk, getting out of the applications.

- ↳ Bypasses buffering
- ↳ locking

* Allocation of Frames

- Each process needs minimum number of frames.
- Maximum number = total frames in the system.

. Two major allocation schemes

- ↳ Fixed Allocation
- ↳ Priority Allocation

* Fixed Allocation

. معناها بنقسم عدد الـ frames على عدد الـ processes بالتساوي، يعني لو عنا 100 frames، 5 processes، كل process بتاخذ 20 frames.

* Proportional Allocation (بتتوزع بناءً على حجم البروسس)

. Dynamic as degree of multiprogramming, process size change

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

s_i = size of process p

$S = \sum s_i$

m = total number of frames

* Global Vs. Local Allocation

* الـ Global البروسس بتقدر تاخذ frame من أي محل بـ ها، إيام حتى لو مش إلها بس هاد الإشي بأثر على وقت التنفيذ مع هيك الإنتاجية بتزيد.

* الـ local البروسس بتوخذش غير من الـ frames تاعونا وهاد الإشي عكسه يأدي لاستغلال غير جيد للذاكرة.

* Reclaiming Pages

* Non-Uniform Memory Access (NUMA)

- speed of access to memory varies.
- optimal performance ⇒ "close to" CPU
- solved by Solaries by creating Igroups

عشان يراقبوا علاقة الـ CPU بالميموري وهكذا

* Thrashing

If process doesn't have enough pages, the page fault rate is very high.

Thrashing: A process is busy swapping pages in and out. (الـ pages الي بتم تخزينها)

⇐ ليش هاد الإشي بصير؟ لأنه عدد الـ pages الي البروسس معينة أكبر من حجم الميموري فالبروسس بتضلها تبدل بين الـ pages

when

$\sum$ size of locality > total memory size

* Working-set Model

$\Delta \equiv$ working-set window = a fixed number of page references

- if Δ too small $\Rightarrow$ will not encompass entire locality
- if Δ too large $\Rightarrow$ will encompass several localities
- if $\Delta = \infty$ $\Rightarrow$ will encompass entire program.

$D = \sum WSS_i$ = total demand frams

- if $D > m \Rightarrow$ Thrashing