- Pages : split logical memory into pages of the same si[ze]
  ↳ stored (power of two)
- Frames : split Physical memory into pages of the same size

*) Each page is stored in a frame.

*) A process has 1 or more pages of memory.

* If we use page we don't need a contiguous memory block we can save one by one

Page table : A page table (an array) gives the memory address (frame) of each page of memory.

logical memory ⟨ Page number :
Page offset :

* technically cpu uses X2 times to find

cpu → page table → physical ~~table~~ memory

X2 times

So we use a Table lookaside Buffer.

TLB ← implemented in hardware to use associative memory.
↳ many if statements

TLB hit → an appropriate TLB entry was found
⇒ the associated frame is returned.

TLB miss → no TLB entry is found
⇒ the frame number is returned from the page table in memory.

TLB performance.

ε = TLB Look-up time.

Memory access cycle : 50us

Probability of a TLB hit : a

$$\underbrace{\varepsilon + a.50}_{\text{one access}} + \underbrace{(1-a).100}_{\text{Probability of } 1-a}$$

one access
= probability of a

Probability of 1-a
two access

ex) $\varepsilon + a.50 + (1-a)100$
$= \varepsilon + a.50 + 100 - 100a$
$= \varepsilon + 100 - a.50$

Phantom memory

Moar bits

dirty bits

Valid bits

$$\text{Processor} \xrightarrow{\text{logical address}} \boxed{\text{MMU}} \xrightarrow{\text{Physical address}} \boxed{\text{memory}}$$

Paging
Segmentation

logical address $\xrightarrow{\text{Mapping}}$ Physical address = binding.

paging system 에서는 빈 프레임에 어떤 페이지든 적재 할 수 있어 메모리를 효율적으로 사용 가능하다. 또한 외부 단편화가 생성되지 않는다.

disadvantage

1) 한 프로세스를 많은 페이지로 나누어서 메인 메모리의 여러 군데에 분산 적재하면 운영체제의 페이지 관리 부담이 크다.

2) 또 프레임 단위로 적재 함으로 어떤 프로세스의 공간이 필요한 페이지 크기와 맞지 않으면 마지막 페이지에 할당된 프레임이 완전히 차지 않아 내부 단편화가 생길 수도 있다.

3) page 크기가 너무 작으면 page table을 유지 하는데 부담을 줄 수 있다. 페이지 테이블을 유지하는 부담은 페이지 크기를 증가 시켜서 줄일 수 있다.

page fault handling Algorithm.

1) First in First out

* Belady's anomaly → more memory spaces to process but it gets more page faults.

number of page frame ↑
number of page fault ↑
FIFO 's problem.

∴ First - in - First - out is not the best

but in the general case, if number of frames ↑ then number of page faults ↓

then Let's beat the Belady's anomaly.
How? → Let's Look at the future.
= optimal replacement. = theorical, we can't see the future but the lower bound that we can know.

제일 나중에 사용할 frame을 victim으로 고르고 마지막에는 가장 위칸을 고르면 된다.

Convoy effect = ~~slowing down the whole operating system because of few slow process~~
In non-preemptive cpu scheduling if long working processes has the cpu all others process should wait for that process.

future is least recently used, which we can't see the
the past working set, we predict from

using Clock.

↳ 헷갈린다.

need to loop over n frames of memory to find the least recently used.

↳ So use priority queue.

Second chance algorithm.
Instead of a clock per page, have a single bit
a reference bit for page

① reference bit 1 이면
② 0으로 바꾸고 ㄷ
③ 다음 page table로
④ 그 다음 0을 찾으면 그게 victim.

Valid bit, reference bit.

P 50 + (1−P)

So the best one when we can't see the future is least recently used, we predict from the past working set.

using Clock.

↳ 헷갈린다.

need to loop over n frames of memory to find the least recently used.

↳ So use priority queue.

Second chance algorithm.
Instead of a clock per page, have a single bit
a reference bit for page

① reference bit 1 이면
② 0 으로 바꾸고
③ 다음 page table로
④ 그 다음 0을 찾으면 그게 victim.

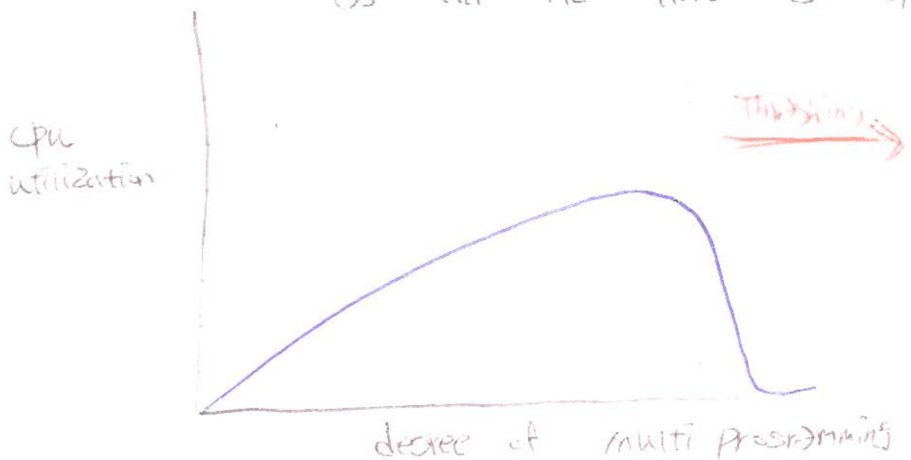
valid bit, reference bit.

$P\,50 + (1-P)$

if localities (working set) > available
means page faults are frequent

then low cpu usage as time is spent swapping.

Thrashing = almost no useful work is getting done,
as all the time is spent swapping



Prefetching = CPU가 앞으로 수행될 명령어들을 미리 인출하여 CPU 내부의 큐에 넣어 둠으로써 수행속도를 향상 시키는 법.

we can't see the future but we can tell to the computer what is going to be happen
= Hint

hint = what data might be accessed next.

Paddy

Small page size	Large page size
~~More TLB hit rate~~	Large localities
More TLB miss rate	More Thrashing
	reduce TLB miss rate

2) ~~Page table~~

- Multi programming → 한정된 메모리를 여러 프로세스가 함께 사용.
- 메모리 관리 → Process들을 위해 메모리를 할당하고 제거하며 보호하는 활동

fetch policy → deciding when process going to allocate on memory from disk.
언제 메모리에 적재?

Requirement fetch
Prediction fetch

Placement policy → first in first allocate
디스크에서 반입한 프로세스 / Best fit
메모리를 어느 위치에 저장 할 것인가? / worst fit.

Replacement policy → First in First out
메모리가 충분하지 않을 때 현재 / Least Recently used
적재된 메모리 중 어떤걸 제거?

Memory address

logical address $\xrightarrow{\text{MMU}}$ physical address.

= 하드웨어

Address binding = logical address를 physical address로 mapping 하는 작업.

binding 종류.

1) Compiler가 원시 코드를 컴파일 하여 목적 파일 생성
2) Linker가 목적 파일을 라이브러리 파일이나 다른 목적 파일에 결합.
3) loader가 메모리에 프로그램 배치.

Swapping : 중기 스케줄러, 즉 디스크에 저장된 프로세스를 메모리로 옮기고, 메모리에 적재된 프로세스는 큐에서 대기.

Swappin - in
swapping - out

Memory allocation Method

연속

· 크기가 다른 프로세스에 같은 메모리 양을 배치해서 메모리가 낭비된다.
Internal, external fragmentation 발생
↳ Compaction을 사용하여 external fragmentation 해결 → 하지만 실행시간 더 걸림.

↓

First Fit
best Fit
worst Fit

) 연속적재

fragmentation Problem

비연속

프로세스 하나가 물리적 공간의 여러 공간에 분산되어 올라 가는 것.

고정 : Paging

비고정 : Segmentation

Solution 1)

메모리 통합 = Coalescing → 다른 인접한 빈 공간이 비어 있으면 점검하여 하나로 합친다. 하지만 메모리 전반에 흩어진 빈 공간 모으는 힘들다.

(인접한 공간 통합)

Solution 2)

메모리 압축 = Compaction → 메모리 내용을 적절히 움직여 사용 가능 공간을 큰 블록 하나로 만드는 것.

(어디든 가능)

단점 = 시스템 자원 소모가 크고 압축하는 동안 시스템은 모든 일을 중지해야 하기 때문에 실시간 시스템에 심각한 문제가 된다.

Buddy System ⇒ 연속 할당 시스템에서 단편화 문제를 해결하기 위해 큰 버퍼를 계속 2등분 하는 것

Paging → 프로세스를 크기가 동일한 page로 나누고, 메모리 또한 frame로 나누어 page를 frame에 담는것.

paging 의 장점 → external fragmentation X
empty frame 에 어떤 frame이든 적재 가능해서 메모리를 효율적으로 사용 가능.

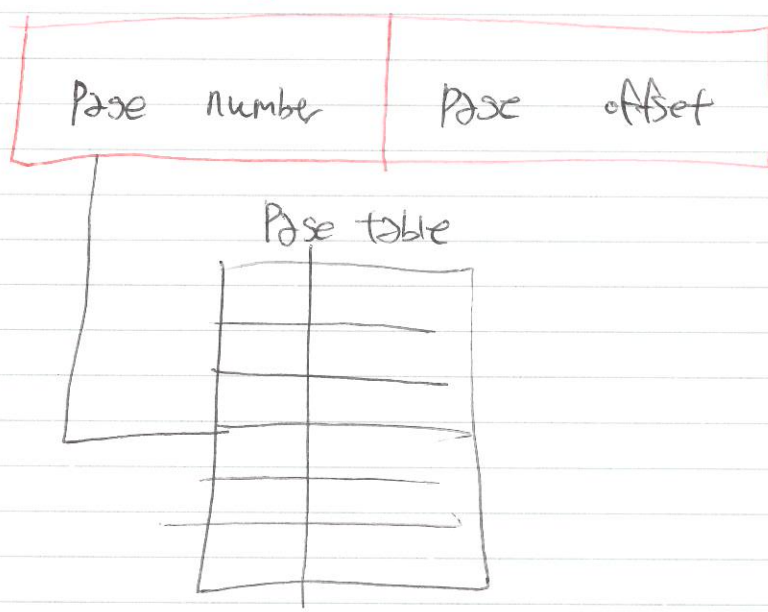
→ 하지만 page를 메인 메모리의 여러 위치에 분산 적재 함으로 page 관리 부담이 크다.
Internal fragmentation 현상 발생

logical address = page index + page offset

page table = page index 와 physical address의 frame number를 포함한다,

★ page size = Always 2^n logical address → page number 와 offset으로 쉽게 변환하기 위해서.

logical address



-7-