Memory Management (2/2)

Page Replacement

메모리 과다 할당 상태 (Over-allocating of memory)

멀티 프로그래밍 시스템에서 메모리 내에 위치한 유저 프로세스의 수가 증가함에 따라 발생

모든 유저 프로세스가 사용하는 페이지 수보다 물리 메모리의 프레임 수가 적은 상황

해결 방법

Page fault 처리에 Page replacement를 추가한다.

Page Replacement

물리 메모리에 위치한 페이지를 디스크에 저장하고, 요구된 페이지가 해당 프레임을 할당 받도록 하는 방법

현재 사용되지 않는 메모리를 디스크로 내보내는 표시를 하고,

Page Fault with Page Replacement

1. 디스크에서 요구된 페이지의 위치를 찾는다
2. 물리 메모리에서 Free frame을 찾는다

   Free frame이 있다면 사용한다. ② 없다면, 페이지 교체 알고리즘을 사용하여 교체할 프레임(victim frame)을 선택한다. ③ 교체할 프레임을 디스크에 저장하고, 페이지 테이블, 프레임 테이블을 변경한다.
3. 요구된 페이지를 2에서 선택된 Free frame으로읽어 들이고 해당 페이지 테이블, 프레임 테이블을 적절하게 변경한다
4. 유저 프로세스를 재 시작 한다

Basic Page Replacement 도식

page out을 할 필요는 없지만 page table업데이트는 해야함

더이상 text를 담고 있지 않기에 invalid하다고 해줘야 함

그리고 새롭게 할당된 물리메모리에 대하여 page table업데이트를 해줘야 함

stack → page out → swap

질문 : 교수님 특정 사용자를 차단하려면 SYN 패킷의 필드를 확인하고 작업을 수행해줘야 할텐데 여기에 CPU를 사용 할 것 같습니다. 그렇다면 DDoS 공격은 네트워크카드에서 해결해야하나요?

답변 : 네트워크 카드에서, 커널의 네트워크 스택에서 하는 경우도 있다.

Yes. 네트워크 카드만이 only는 아니다.

질문 : 교수님 text page가 프로그램에 존재하기 때문에 pageout 하지 않아도 된다는게 이해가 잘 되지 않습니다ㅠㅠ

답변 : 프로세스의 메모리 이미지가 T-D-S이다.

→ a.out이 executable이다. 이것으로부터 T-D-S로 복사가 된다.

T의 특성은 Readonly이다. 이 말은 값이 달라지지 않는다는 의미이다. 사용자가 바꾸려고하려면 MMU가 fault를 뱉음.

null pointer에 접근한다는 것은 0번지에 접근한다는 것인데, 이는 T에 접근한다는 의미이다.

0번지는 Read/Write도 다 안되는 T인 것이다.

T는 다른 페이지에게 값을 넘겨줘도 다시 값을 읽을 수가 있다.

(변경된 값을 반영하기 위해 storage에 저장한다)

프로세스가 필요한 working set 제공해주면 paging fault가 최소화

질문 : working set의 수만큼 page를 할당해주면 page fault를 최소화 해줄 수 있는데, process가 실행이 되기 전에 working set의 수를 어느 정도 예측할 수 있는건가요?

답변 :

이는 사실 연구해야하는 문제이다.

머신러닝 같은 알고리즘이 계속 도는 경우가 많은데 이런경우에는

프로그램을 한번 돌리고 locality를 예측을 해 놓는 것이다.

미리 돌려보고 필요한 메모리를 제공하는 것이다.

NO. 알 수 없다.

(그러나 특수한 경우에서는 가능하지도 않을까...?)

좋은 질문 :

아직 잘 이해가 안됐습니다 ㅠㅠ... Swap Space가 디스크에 존재한다고 하면 stack fault가 일어나면 디스크를 거쳐야하는 것 아닌가요?

답변 : 이론적으로는 그러한데, 이론적으로 그렇지 않다.

page out이 된것이 아니라면 읽어올 것이 없다.

질문 :

해당 스택이 page out이 됐었는지 여부는 어떻게 알 수 있나요?

답변 :

PTE flag에 있다.

page out되면, 그 페이지는 invalid해줘야한다.

anonymous?

질문 :

process 실행 시 시간의 흐름에 따라 page fault가 0이 되는 현상에 대하여 조금 더 설명해주실 수 있으실까요? process 실행 중 모든 text page가 동시에 memory에 존재하게 될 가능성은 매우 희박할 것 같아 text page에 대한 page fault가 주기적으로 발생할 것으로 예상되어 질문드립니다.

답변 :

working set에서 프로세스가 돌고 있을 때, locality

Page Replacement 고려사항

각각의 유저 프로세스에게 어떻게 프레임을 분배해 줄 것인가?

Frame-allocation algorithm

어떤 프로세스는 메모리를 조금 쓰고 어떤 것은 많이 쓴다.

fairness의 문제.

페이지 교체가 필요할 때 어떻게 교체할 페이지를 고를 것인가?

Page-replacement algorithm

victim을 어떻게 고를 것인가?

위 알고리즘들은 모두 페이지 교체에 의한 I/O 작업 수행 횟수를 최대한 줄이려는 목적을 가지고 있으며, 적합한 알고리즘의 사용은 시스템의 성능을 크게 좌우하는 요소이다.

I/O 작업은 매우 큰 비용을 사용하기 때문

I/O 작업은 ms 단위이다. 원래는 ns단위이다.

질문 : data는 어떤 식으로 page out이 되는지 궁금합니다.

답변 : 과제!

Page Replacement Algorithms

어떤 페이지 교체 알고리즘을 사용할 것인가?

가장 낮은 Page-fault 발생 빈도를 가진 알고리즘

즉, 가장 낮은 I/O 작업 횟수를 요구하는 알고리즘

→ I/O를 최소화 시키는 것과 Page-fault 발생 빈도를 최소화 시키는 것과 동일

여러 알고리즘들을 비교하기 위해 아래 환경을 가정

세 개의 프레임이 할당됨

Page-fault 발생 빈도는 프레임의 개수와 반비례함

페이지를 참조하는 순서 – 20번

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

한 번 참조된 페이지는 페이지 교체가 일어나기 전에는 물리 메모리에 위치한다

이 경우에는 Page-fault가 발생하지 않음

페이지 7의 이경우에는 page-fault가 발생하지 않음 에서 '이 경우'가 어떤 경우인가요?

→ 메모리에 있으니까 당연히 pf가 발생하지 않는다.

Optimal algorithm

앞으로 가장 오랫동안 사용되지 않을 페이지부터 먼저 교체함 → 이것이 optimal일 것이다.

모든 알고리즘 중 가장 낮은 page-fault 발생 빈도를 가지는 이론상 최적의 알고리즘

앞으로 어떤 페이지가 사용될지를 미리 알 수 없다 – 실제 구현 불가능

→ 미래를 알아야 구현가능

다만 왜 이것이 중요한가? upper bound가 되기 때문이다.

→ 얼마나 더 개선할 수 있는지를 알 수 있기 때문이다.

처음에 pf가 나는 것은 어쩔 수 없다는 것이다.

접근한 페이지에 다시 접근할때의 PF를 줄일 수 없느냐 있느냐의 문제인 것이다.

optimal은 7, 0, 1중에서 가장 멀리 있는 녀석이 누구냐는 것을 묻는 것이다

그런식으로 하면 page fault가 9번 난다.

FIFO algorithm

구현이 쉽고 대부분의 lower bound가 되어준다.

가장 먼저 들어온 것을 교체한다.

SCR algorithm

Second Chance Replacement
FIFO기법의 단점을 보완하는 기법
오랫동안 주 기억 장치에 존재하던 페이지들 중에서 자주 사용되는 페이지의 교체를 방지하기 위해 고안됨
FIFO 큐를 만들고 사용하되, 참조 비트를 두어 페이지를 관리한다

참조 비트(Reference Bit)

최초로 프레임에 로드 될 때와 페이지가 참조되었을 때마다 1로 세팅 된다

일정 주기마다 다시 0으로 리셋 된다

참조비트를 두어서 최근에 접근되었는가를 알게한다.

제거 대상으로 선택된 페이지의 참조 비트가 1로 세팅 된 경우, 최근에 사용된 페이지므로 제거하는 대신 참조 비트만 0으로 리셋

→ 경고를 한번주는 것이다.

→ working set에 들어갔느냐의 여부를 따지는 것이다.

→ 교수님 질문 : 왜 counter를 두지 않는가? 여러번 체크하면 좋지 않나?

→ 답변 : 비트를 얼마마다 한번씩 시켜야하나? → every memory access마다!

→ 그때마다 counter와 bit의 연산 시간이 어떻게 다른가?

→ 그 연산 시간가 있다. bit는 1clock, counter는 최소 2~3clock인데 이것을 모든 메모리에 다 해줘야한다는 것이다.

→ 결과적으로 성능을 떨어뜨리는데 큰 기여를 한다.

→ 운영체제의 최고 절대 미덕은 성능이다. 구조적 아름다움이 아니다.

Clock algorithm

counter를 시계처럼 돌리고, 두개의 포인터가 돌아간다.

Hand는 큐를 따라 1칸씩 이동한다

Hand가 가리키는 페이지의 참조 비트가 1이라면 0으로 리셋

pageout은 hand를 따라가면서 pageout시킴

head는 전체 물리메모리에 대해서 돌아가고, 참조된 것을 주기적으로 reset시킴

threahold 이하로 떨어지면

threahold 1, threahold 2가 있어야 한다.

threahold 2이하가되면 hand가 더 빨리 도는 것이다.

질문 : 일정 주기마다 참조 비트가 0으로 리셋 된다고 했는데 그러면 카운터가 어디선가 결국 필요한거 아닌가요?

답변 : 여기서는 참조 비트를 사용한다.

질문 : Clock Algorithm은 자체적으로 여유공간을 두는 알고리즘 같은데, 그렇다면 여유 공간이 없을 때 보다 page fault는 조금 더 발생할 수 있겠지만, page replacement가 최대한 발생하지 않도록 하는 알고리즘이라고 이해하면 될까요?

답변 : yes 페이지 replacment를 먼저해서 replacment를 줄이는 방법이다.

질문 : clock aglorithm에서 참조비트가 1이 되는 것은 SCR처럼 page에 access할때인가요?

답변 : yes

질문 : SCR algorithm에서 reference bit이 0으로 리셋되는 것은 모든 page entry 에 대한 것인가요? 아니면 각각의 entry마다 일정시간이 지나면 1->0으로의 전환이 있는 것일까요?

답변 : 다 스캔을 하고 바꾸는 것이다.

질문 : 교수님 아직 paging과 메모리 구조(data, text, stack)의 관계가 이해가 잘 되지 않습니다ㅠㅠ paging이 단순히 주소공간을 4KB 동일한 크기로 나누어서 물리 메모리에 일부만 올린다고 이해를 했었습니다.

그렇다면 paging은 메모리 구조(data, text, stack)에 상관없이 4KB씩 잘려서 관리되는 건지 궁금합니다. 그렇가면 이전에 얘기했던 text page는 4KB씩 나누는데 해당되는 부분이 text인 페이지를 말씀하시는 건가요?

LFU algorithm

- Least Frequently Used
- 사용 빈도가 가장 적은 페이지를 교체하는 기법
- 지금까지 가장 적게 참조된 페이지가 교체 대상으로 선택된다
- 일단 프로그램 실행 초기에 많이 사용된 페이지는 그 후로 사용되지 않더라도 프레임을 계속 차지하는 문제가 있음

이 알고리즘은 시간에 대한 개념이 필요하다

시간적인 문제를 제대로 반영하지 못한다는 단점이 있다.

NRU algorithm

- Not Recently Used
- 최근에 사용하지 않은 페이지를 교체하는 기법 페이지마다

  → 문제는 최근을 어떤 식으로 나타내느냐이다.
- 참조 비트와 변형 비트를 두어 관리한다
- 참조 비트(Reference Bit)

  Reference → read/접근을 했다.
  - 최초로 프레임에 로드 될 때와 페이지가 참조되었을 때마다 1로 세팅 된다
  - 일정 주기마다 다시 0으로 리셋 된다
- 변형비트(Modified Bit)

  Modified → 변경이 되었다
  - 최초로 프레임에 로드 될 때는 0
  - 페이지의 내용이 바뀔 때 1로 세팅 된다
- 페이지 교체가 필요한 시점에 다음 순서대로 교체 대상으로 삼는다
  - 참조 0, 변형 0
  - 참조 1, 변형 0
  - 참조 0, 변형 1
  - 참조 1, 변형 1

  참조와 변형의 중요도는 차이가 있다.

  참조와 변형을 반영하는 역할은 MMU가 한다.

LRU algorithm

- 가장 오랜 시간 참조되지 않은 페이지부터 먼저 교체함
  - 페이지 사용의 지역성을 고려하여 optimal algorithm과 유사하며 실제 구현 가능한 알고리즘

    → locality를 가정한다.

    이 알고리즘은 좋은 성능을 낸다.

    → 이 말은 가정이 맞았다는 것이다

    → 그 가정은 locality가 있다는 것인데, 실제로 locality가 존재한다.
  - 구현 방법
    - Counter의 사용 : 참조된 시간을 기록
    - Queue의 사용 : 한 번 사용한 페이지를 큐의 가장 위로 이동시킨다
      - 가장 위의 페이지 : 가장 최근에 사용된 페이지
      - 가장 아래의 페이지 : 가장 오래 전에 사용된 페이지

    r, w을 쓸 수도 있다.

    → 다만 그러려면 주기적으로 reset을 시켜줘야 한다.

    → reset을 안시키면 최근인지 아닌지를 알 수가 없다.

NRU 나 LRU나 가장 오래전에 참조된 페이지를 교체하는 것 같은데 정확히 어떤 차이가 있는건가요...? 구현 상의 차이인가요??

NRU는 접근을 안한 것 (bit를 이용해서 최근에 사용하지 않은 프로세스를 approximation한다.)

LRU은 큐를 사용해서 맨 마지막에 있는 것을 뺀다

queue를 통해서접근하면 두번의 memory write가 있다는 말이 무슨 뜻인가요?

A. 큐의 노드를 이동시킬 때 pointer를 두번 변경해야 한다는 말이다.

Swapping

- Page out으로도 메모리 부족을 해결하지 못할 경우 필요한 기법

  → free memory라는 global 변수가 있다. → 가용 메모리를 나타낸다.

  이 값이 커지지 않거나 오히려 떨어지면 page out으로는 해결이 안되는 것이다.

swap threadhold ≥ free memory이면 Swap out을 해야 한다.

이렇게 되면 sleep을 하고 있는 프로세스(작동을 하지 않는 proccess) 같은 것들을 찾아서 swap해버린다.

- Swap out 대상이 된 **프로세스 전체**를 secondary storage로 보낸다

  학생질문

  프로세스 전체가 secondary storage로 내려갈 경우 어느정도의 속도 저하가 생기나요?

  → 많이 생긴다

  swapping이라는 동작을 가지고는 있으되, 사용이 되지 않도록 해야 한다.
- 이렇게 page out이나 swapping에 사용되는 secondary storage (backing store)를 swap 영역이라 한다

Page out은 page를 하나씩 빼는 것이다.

Swapping 도식

어떤 프로세스를 swapping하나?

victom proccess

page replacement와 비슷한 방법으로 선택을 한다.

Contiguous Memory Allocation – NO VM

imbeded device를 사용하는 경우는 VM을 안쓴다.

OS는 상황에 따라 다르게 설계되어야 한다.

지금도 필요한 경우가 있다. 스마트 공장같은 것

<1> Single Partition Allocation

- 가장 단순하게 메모리를 사용하는 방법
- user program 영역을 한 번에 오직 1개의 user program만 사용하도록 한다

<2> Multiple Partition Allocation

- <1>의 방법에 multiprogramming 개념을 추가하여 user program 영역을 여러 개의 user program이 사용하도록 한다

  100KB → 50KB, 20KB, 30KB로 미리 쪼개 놓는다.

<3> No Partition

- 각 프로그램이 필요에 따라 전체 user program 영역을 사용
- 이 경우, page/swap out시에 garbage collection이 필요하다

Memory Allocation Problem

- Program을 물리 메모리에 load할 때, 비어 있는 메모리 공간이 여러 개 있을 때, 그 중에서 어떤 것을 선택할 것이지를 결정하는 것

  비어 있는 공간 중 어디에 placement할 지를 정하는 것이다.

- First-fit

  맨 처음 빈 공간에 넣는 방법

- Best-fit

  자투리 공간이 가장 적게 나는 곳에 넣는 방법

- Worst-fit

  자투리 공간이 가장 많이 나는 곳에 넣는 방법

  → 사실 자투리 공간은 버려지는 공간이 아니다.

  → 그렇기에 때에 따라서는 Worst-fit이 가장 좋을 수도 있다.

Fragmentation

- External Fragmentation
  - 프로그램에게 할당 후, 남은 메모리의 조각이 새로운 할당 요청에 충분하지 않은 경우
    - Paging은 external fragmentation을 해결하기 위한 방법

    프로그램 단위로 물리메모리에 할당한 이후 자투리 공간이 쓰이지 않는 경우

프로그램도 page로 쪼개고, 물리메모리도 page로 쪼개어서 page to page로 할당되기 때문에 External Fragmentation는 존재하지 않는다.

- Internal Fragmentation
  - 할당된 단위 내부에 메모리가 사용되지 않는 경우
    - Paging에서 page frame은 4KB로 고정 되었지만 요청한 물리 메모리 영역이 3998B인 경우 2B의 internal fragmentation이 발생하게 된다.

  프로그램과 물리메모리가 각각 4KB로 할당 되었는데,

  프로그램이 2B만을 사용한다면, 나머지 4094B는 낭비되는 것이다.

  페이지의 크기가 커질수록 Internal Fragmentation의 크기도 커진다.

Fragmentation 예

Protection과 Relocation

교수님 질문

proccess의 abstraction은 뭘 제공하는가?

1. 스케줄링의 단위
2. 메모리 주소에 대한 protection

어떻게 protection를 제공하는가?

프로세스가 PT를 하나씩 할당을 받아서 이 같은 protection을 제공한다.

protection은 MMU가 제공한다.

protection은 PT와 PT를 관리하는 MMU의 동작을 통해서 제공된다.

MMU가 없는 상황에서는 어떻게 하나?

VM이 없다는 것은 MMU가 없다는 것인데, 이런 경우에는 protection을 어떻게 제공하는가?

- Protection

    - Contiguous memory allocation 방법을 사용할 때, OS의 메모리 영역과 User program의 메모리 영역은 서로 구분되어야 함
    - 서로의 영역을 침범하지 못하도록 보호를 해야 함

- Relocation
    - User program은 재배치 가능한 주소로 표현됨
    - 재배치 가능한 주소를 이용하여 프로그램이 어느 위치에 load되더라도 쉽게 code의 주소를 결정할 수 있어야 함
- Protection과 relocation을 위한 hardware의 지원
    - Limit register와 relocation register를 이용하여 구현

Protection과 Relocation (cont.)

- Limit register: 참조가 허용되는 주소의 최대값
    - Limit register의 값을 비교하여 참조하는 주소가 허용되는 영역인지를 판별

  register에서는 간단한 값 비교가 가능하다.

  Limit register를 이용하여 어떠한 값 이하로 내려 갈 것을 정하는 것이 가능하다.

  이 조건을 만족하지 않으면 trap을 발생시키는 것이다.

- Relocation register: 프로그램이 차지하는 주소 영역 중 첫 번째 주소
    - 재배치 가능한 주소를 통해서 실제 물리 메모리의 주소로 참조 가능하게 함

1:08:25