Beyond Physical Memory

Swaping Mechanism

- 메모리 계층에 하나의 레벨이 더 필요함
- 물리적 메모리보다 큰 곳
- 운영체제가 현재 잘 안쓰는 Address Space를 넣어 둘 곳 -> 주로 하드 디스크

Swap Space

- 운영체제는 최우선적으로 프로세스를 실행 할 때 코드/데이타를 메모리에 넣어야함
- Swap Space를 추가함으로써, 운영체제가 동시에 실행하는 여러개의 프로세스를 위한 큰 가상 메모리가 있는 것 처럼 보이게 할 수 있음
- 디스크에서 페이지 앞/뒤로 이동하기 위해 어느정도의 공간 확보
- Page 크기의 단위로 Swap Space를 기억

Present Bit

- PTE(Page Table Entry)에 저장돼있는 비트, Disk에서/Disk로 Page Swap을 하기위해 시스템 윗부분에 machinery 추가
- 1 : 페이지가 물리적 메모리에 존재한다, 0 : 메모리에는 없고, 디스크에

Page Replacement Policy

- 메모리가 가득차서 새로운 페이지를 들이기 위해서 존재하는 페이지를 쫓아내야 할 때
- 어떤 페이지를 쫓아낼지
- 실제로 Replacement가 발생할 때(Swap daemon / Page daemon의 역할 - Background thread) : 메모리 전체가 가득차면, 페이지 하나를 Replace -> 비현실적임 -> 프리 메모리가 Low Watermark보다 적을 시, 페이지들을 내보내서 High Watermark 위까지 프리 메모리로 만듬

Page Fault

- 물리적 메모리에 존재하지 않는 페이지에 접근을 시도하는 것
- 디스크에 있는 페이지를 swap해서 메모리로 들고와야함

Swaping Policies

- 메모리의 용량이 유한하기 때문에, 공간이 부족하면 OS가 몇개의 페이지를 쫓아내야됨
- 운영체제의 Replacement Policy에 따라 그 방법이 달라짐

Cache Management

- Cache Miss를 최소화 하는 것이 목적
- AMAT(Average Memory Access Time) = (P_hit * T_m) + (P_miss * T_d) -> P_hit : hit 확률, T_m : 메모리 접근 비용, P_miss : miss 확률, T_d : 디스크 접근 비용

Optimal Replacement Policy

- 전체적으로 가장 적은 Miss
- 제일 나중에 접근 될 페이지를 없앰
- 가장 완벽한 방법으로, 실현 가능성은 없고 비교용임

FIFO Policy

- 페이지가 들어오면 큐에 들어감
- 먼저 들어온 페이지가 먼저 나감
- 구현하기 쉽지만, 해당 페이지들의 중요성을 인지하지 못함
- Belady's Anomaly : 캐시 크기가 커지면 hit rate가 무조건 올라갈것같지만 FIFO의 경우 아닐 때가 있음

Random Policy

- 랜덤 페이지를 골라서 내보냄
- 그냥 말그대로 랜덤임
- Hit rate도 말그대로 랜덤임

Using History

- Recency : 최근에 접근 된 페이지가 다시 접근 될 확률이 높다 -> LRU
- Frequency : 많이 접근 된 페이지는 내보내선 안된다 -> LFU
- 과거 이력을 사용하기 위해서는 메모리 레퍼런스를 할 때마다 추가 작업을 해야함 -> 하드웨어 지원으로 해결

LRU(Least Recently Used)

- 사용한지 가장 오래 된 친구를 내보냄
- 하드웨어 지원 : Use Bit - 페이지가 레퍼런스 되면 use bit가 1로 변경, 하드웨어는 이 비트를 이후 안건드림 (운영체제의 역할)

Clock Algorithm

- 모든 페이지가 원형 형태로 존재
- 시곗바늘이 특정 페이지를 가리키면서 시작
- use bit가 0인 페이지를 찾을 때 까지 시곗바늘 돌아감 -> 1을 만나면 use bit를 0으로 만들고, 0을 만나면 페이지 내보냄
- Dirty Page를 고려하는 경우(Modified Bit = Dirty Bit) -> 수정된 페이지는 디스크로 돌아가서 바뀐정보를 저장해야함 -> (Reference, Dirty) 쌍으로 우선순위 결정 : (0,0) > (0,1) > (1,0) > (1,1)
- N'th Chance Version : 페이지별로 N번의 Chance를 부여 -> N번만큼 걸리면 내보냄
- 하드웨어가 지원하는 Dirty Bit가 필요한가
    - 모든 페이지들을 ReadOnly로 불러옴
    - 쓰기동작을 시도할시 OS에게 트랩을 넘겨서 Dirty Bit를 Set해서 Read-Write모드로 바꿈
    - 페이지가 디스크에서 돌아오면 Read-Only로 바꿈
- 하드웨어가 지원하는 Use Bit가 필요한가
    - 모든 페이지를 불가능으로 바꾼다(메모리 포함)
    - Invalid 페이지 읽기를 시도 할 경우 OS에게 트랩을 넘김
    - Use/Modified Bit를 셋하고, 페이지를 read-write 모드로 바꿈
    - 클락 핸드가 지나갈때 Use/Modified Bit/Valid 비트를 리셋함

Second Chance List

- 메모리를 두 파트로 나눔(Active List - RW, SC List - Invalid)
    - Active List에 있는 페이지들은 최고속도로 접근
    - 아닌 경우에는 Page Fault 발생
    - Overflow 페이지(리스트의 끝자락)을 SC리스트의 첫부분으로 옮기고 Invalid 표시
    - 찾는 페이지가 SC리스트에 있는 경우 활성리스트로 옮기고, RW 표시

    - 찾는 페이지가 SC리스트에 없는 경우 해당 페이지를 활성리스트의 첫부분으로 옮기고 SC리스트의 LRU Victim 페이지를 쫓아냄
    - USE BIT를 사용하지 않음

Free List

- Demand Paging에 사용하기 위한 Free Page들의 집합을 가지고 있음
    - 여기서 Free List는 백그라운드에서 Clock 알고리즘등을 통해 채움
    - Dirty 페이지들은 리스트에 들어오면 디스크에 쓰기 시작
    - 페이지가 다시 사용되기 전 필요해지면, 쓰는 페이지들 쪽으로 다시 이동
- Page Fault 발생 시 반응 속도가 훨씬 빨라짐

소프트웨어 TLB에 Use Bit가 필요한가?

- 하드웨어 쪽 TLB에서는 Use Bit 사용 // TLB Entry가 대체되면 소프트웨어가 페이지 테이블에 Use Bit 복사
- 소프트웨어 TLB는 엔트리들을 FIFO 리스트로 보관, TLB에 없는 페이지들은 Second-Chance List에 있음 (LRU 방식)

Core Map

- 페이지 테이블은 VPN -> PFN 제공
- 반대 쪽 매핑이 필요한가? : 그렇다
    - 클락 알고리즘은 페이지 프레임 기준으로 굴러감 (물리적 페이지를 공유하는 경우에는 Virtual Page가 여러개일수도있음)
    - 모든 PTE를 invalidate 하기 전에는 페이지를 다시 디스크로 넣을 수 없음

Workload Examples

- Locality가 없는 워크로드의 경우 : 캐시가 워크로드를 수용하기 충분하다면, 어떤 policy도 rate 차이가 없음
- 80-20 워크로드 : Locality 존재 - 80퍼의 레퍼런스는 페이지의 20퍼를 가리키고, 20퍼의 레퍼런스는 페이지의 80퍼를 가리킴 -> Hot pages(자주 접근되는 페이지)는 LRU가 hit rate 높음 -> Clock의 경우 LRU보단 조금 낮지만, 다른 방법들보단 나음
- Looping Sequential : 첫 50 페이지는 Unique하게, 그 이후는 loop인 경우 -> 첫 50까지는 FIFO/LRU는 0의 히트레이트, 이후는 100의 히트레이트, RAND는 점진적으로 증가

Page Selection Policy

- 운영체제는 언제 페이지를 메모리로 가져올지 정해야함
- 그 방법과 옵션이 다양함
- Demand Paging : 실제로 접근 됐을때만 로딩해옴(로딩타임 감소, 메모리 절약)
- Prefetching : 페이지가 곧 사용될 것을 예측해(Guess) 미리 메모리로 가져옴
- Clustering : 대기하고 있는 쓰기 작업 몇개를 모아서, 한번에 디스크로 쓰기 -> 하나하나 하는것보다 효율적

Thrashing

- 메모리가 oversubscribe되어서 물리적 메모리를 초과해 더이상의 프로세스 실행 요청을 받을 수 없을 때
- 프로세스의 부분을 실행하지 않기로 하고, 줄어든 프로세스의 부분만을 실행
- Page Fault Rate가 지나치게 높아서, 각각의 프로세스가 충분한 frame을 보장받지 못할때 -> 실행보다 페이징이 더 오래 걸림

- Working Set : 프로세스가 얼마만큼의 메모리를 필요로 하는지 추측 - WS(t) = t△~t 사이에 레퍼런스된 페이지의 집합 -> Thrashing 현상을 탐지, 새로운 프로세스가 시작 될 때 확인 -> 현재 프로세스들의 Frame Demand의 총합(D) > 시스템에서 사용가능한 프레임의 수(m) 일시 Thrashing 발생 -> D>m 일시 프로세스 몇개를 유예해버리고, D<m일시 새로운 프로세스 실행 -> 최적의 멀티프로그래밍을 하되 Thrashing은 방지
- 각각의 프로세스는 최소 필요 페이지수가 있음 -> 이를 지켜줘야 함
    - Global Replacement : 다른 프로세스로부터 Frame을 받아올 수 있음
    - Local Replacement : 내부에서 해야됨

Allocation

- Equal Allocation : 모든 프로세스가 동일한 수의 프레임을 할당받음
- Proportional Allocation : 프로세스의 사이즈에 따라 할당
    - Allocation = size of process / size of all process * total number of frames
- Priority Allocation : Proportional이지만, 사이즈를 쓰지 않고 우선순위를 기준으로 계산
- Adaptive Allocation? : page-fault rate가 너무 작으면 frame을 잃고, 너무 크면 받음

VAX/VMS Virtual Memory System

- 메모리 관리 하드웨어 : 32비트 Address Space, 512B page (VPN = 23bit, Offset = 9bit)
- 두개의 세그먼트, 페이지 : 세그먼트 P0 = 코드/힙, P1 = 스택 -> 세그먼트별로 Page Table 저장
- 하드웨어가 TLB를 관리
- 유저와 커널이 같은 Address Space에 있음
- Page Replacement : 레퍼런스 비트가 없음 / Segmented FIFO(Software Management) 사용 -> 각 프로세스가 RSS(Resident Set Size)를 가짐 -> FIFO방식으로 RSS의 페이지들을 관리 -> 글로벌 리스트에는 2번 기회
  Page Clustering