안녕하세요 김민석 학생,

내가 지금에서야 한국에 돌아와서 시차적응을 끝내느라

이번 주에는 오피스 아워를 열지 못했습니다.

답변은 아래 달았어요.

공부하느라 고생이 많습니다!

------ 원본 메시지 ------
보낸 사람 "김민석" < wsms8646@gist.ac.kr>
To "박건혁" < maharaga@gist.ac.kr>
날짜 2022-05-27 오후 4:36:28
제목 [학사 김민석]main memory/virtual memory 관련 질문

안녕하세요 박건혁 교수님
저는 교수님의 운영체제를 수강중인 수강생 김민석이라고 합니다.
8-9 강을 복습하던 도중 질문들이 있었으나 이번 주 TA 아워가 없는듯하여 메일로 질문을 드립니다.

1. 8 장 54 페이지에서 inner page 가 10 개의 bit 로 표시되는 부분에 49 페이지에서 이야기한 'Each entry is 4 bytes'가 여전히 적용이 되기 때문인가요?
그리고 저는 49 페이지를 볼때 이 말을 physical address 가 32-bit 로 표현되기 때문에 각 entry 가 4byte 이고 그렇기에 inner page 의 물리적 크기를 page size 인 4KB 에 맞추기 위해 inner page 의 entry 갯수를 2^10 개로 하여 2^10*4=2^12=4KB 로 만들었다고 이해했는데 이 이해도 맞을까요?
마지막으로 지금까지의 내용이 다 맞다고 하면 이 예시에서는 logical address space 는 64-bit 지만 physical address space 는 32-bit 라고 생각을 하는데 만약 64-bit physical address space 라면 address 구성이 '41 | 9 | 12'가 될거 같습니다. 맞을까요?

-> 일단 49 페이지에서 Page Table entry 가 4byte 인 이유는 32 비트 아키텍쳐에서의 한 클럭 내에 접근 가능한 최대 데이터 크기가 4byte 이기 때문입니다.
그러니 학생의 이해가 일단 맞고, inner page 의 크기를 page 사이즈에 맞도록 디자인한 것에 가깝습니다만 굳이 그래야 할 이유는 없습니다.
다만 page 사이즈에 맞춤으로써 최적화가 되었다고 볼 수 있겠네요.
그리고 64 비트 아키텍쳐에서 page table 의 한 entry 가 8 바이트가 되도록 시스템을 구성하는 것이 당연하겠지만,
본 예제에서는 page table entry 가 4 바이트라고 가정하고 논리를 펴나갔기 때문에 inner page bit 가 10bit 가 되었습니다.
만일 학생의 생각대로 page entry size 를 8byte 로 설정한다면 43 | 9 | 12 가 되겠지요.

2. 8 장 58 페이지의 inverted page table 에서 hash table 기법을 이용하면 효율을 늘릴 수 있다는데 감이 잘 안옵니다. 조금만 더 자세히 설명해주실 수 있을까요?

-> Inverted Page Table 의 가장 큰 장점은 메모리를 최소로 사용한다는 점입니다.
물리적인 메모리의 0 번 프레임부터 MAX-1 프레임까지
  프로세스 | 페이지 숫자
이렇게 저장한 테이블을 참조하기 때문에
시스템을 통틀어 메모리 크기 / 페이지 크기 = 프레임 길이 만큼의 페이지 테이블을 유지하면 됩니다.
예를 들어 모두 4KB 페이지로 나눈 8GB 메모리 시스템이 있다고 하면
  2M, 즉 2 백만 개의 Inverted Page Table 을 *단 하나만* 유지하고 있으면 됩니다.

이전의 two-level page table 같은 경우 각 개별 프로세스 별로 페이지 테이블을 따로 유지해야 합니다.

그럼 여기서 문제가 발생하게 되는데, Process ID 101 번의 37 번째 페이지를 접근하기 위해서는 Inverted Page Table 전체를 검색해야 합니다.
즉 O(n)의 Search Problem 을 풀어야 한다는 이야기지요.
Two-level page table 은 각각의 프로세스가 개별적으로 페이지 테이블을 유지하고 있고,
각 페이지 테이블의 index 가 페이지 숫자와 같으므로 Search 할 필요 없이
memory access cycle 2 번이면 메모리 접근이 됩니다만,
위 예제에서 Inverted Page Table 은 최악의 경우 2 백만번의 Memory access + 1 번을 해야 데이터를 읽어올 수 있게 됩니다.
여기에 Hash Function 을 써서 해당 프레임을 최대한 빨리 찾게 해주면
그 속도가 급격하게 빨라지게 되겠죠?
본문 내용은 그 이야기입니다.

3. 8 장 59 페이지에서 logical address 가 pid+p+d 로 구성되어 있는데 physical address 가 32-bit 라고 했을 때 앞의 3 field 가 합쳐져서 logical address 도 32-bit 인가요 아니면 p+d 가 32-bit 이고 거기에 pid 가 붙어서 32-bit 를 넘는 logical address 가 되는건가요?(이 질문은 1 번째 질문에서 한 physical address 의 크기와 logical address 의 크기는 다를 수 있다는 것이 맞다고 가정한 후에 하는 질문입니다.)

-> 이 예제에서는 pid 부분이 additional bits 가 될겁니다.
각 프로세스 입장에서 logical address 는 무조건 0 부터 시작해서 maximum 까지 배정되어야 하는 겁니다.
물론, 아직까지는 8 장이니 maximum 까지는 아니고 해당 프로세스가 가진 최대 주소값까지 지정'할 수 있는' 32bit 논리 주소 + pid 가 추가되는 식입니다.
4. 8-9 강 전체적으로 계속 궁금했었는데 여기에서 저는 항상 main memory 에 올라가는 것은 process 이지 program 이 아니라고 생각했습니다. 다만 9 강 파트 1 4 페이지에도 'partially-loaded program'이라고 되어있고 수업시간에 process 가 아니라 program 이라고 강조도 하셨는데 왜 program 인지 궁금합니다

-> 학생 말대로 메인 메모리에는 process 가 올라가야 하는 것이 맞습니다.
그리고 process 는 program 이 'executed' 되어 메모리에 'loaded'된 상태를 말합니다.
8 장에서는 이 구분이 명확했는데, 아직 가상메모리가 적용되기 전이라
program 이 '완전히 load'되어야 process 가 되어 실행이 가능해지기 때문입니다.
다만 8 장에서도 실행을 위해 항상 프로그램 모두가 메모리에 load 될 필요는 없는 경우가 있었습니다.
Dynamic Linked Library 의 경우 실행 시기에 아직 메모리에 load 되지 않았으니,
DLL 을 포함하는 프로세스는 어떻게 보면 Partially-loaded Program 이라고 볼 수 있고
DLL 이 모두 메모리 loaded 된 프로세스는 그제서야 Completely-loaded Program = Process 라고 볼 수 있지요.

사실 프로그램과 프로세스를 분리하는 이유는,
프로그램은 단순한 헥사코드의 집합이고 이걸 그대로 메모리에 올리면 제대로 실행이 불가능하기 때문입니다.
(그런 시스템이 존재하기는 합니다. ATMEGA 같은 소형 시스템은 Absolute Binding 으로 헥사코드를 그대로 메모리에 올려 실행합니다.)
내가 수업 중에 Program 을 강조하는 경우는, 아직 메모리에 Completely loaded 되지 않은 측면을 강조할 때 그렇습니다.

5. 9강 파트 1 20 페이지에서 demand paging 이 처음에 모든 process 를 main memory 에 올려놓고 쓰는 것보다 느리다고 했는데 10 페이지에서 demand paging 은 faster response 를 가진다고 한것과 상반되는거 같습니다.(다만 이 부분은 demand paging 이 처음부터 다 memory 에 올리는 시간+특정 page 에 접근하는 시간에 비하면 response 가 빠르다고 말하신 것 같은데 맞을까요?)

-> 첫 문장에서 '느리다'라고 말한 건, 프로세스가 실행 중일 때의 연산 속도를 말한 겁니다.
단순히 생각해서 프로세스의 모든 페이지가 메모리에 올라와있으면 page fault 가 발생하지 않을 것임에 반해,
demand paging 을 사용하는 경우 page fault 가 발생하지 않을 도리가 없습니다.
따라서 단순 실행 속도만 따질 때 전자보다 후자가 거의 몇십만배 가량 느립니다 (라고는 해도 페이지 당 miliseconds 단위입니다.)

다만 후자의 10p 에서 faster response 라는 말은, 처음 프로그램을 실행한 이후
해당 프로그램이 response 를 주는 시간 자체는 더 빠르다는 말입니다.
어찌되었건 demand paging 이든 메모리에 데이터를 올리는 시간이든 한 페이지를 메모리에 로딩하는 차이는 몇백 나노초 정도로 크지 않습니다.

다만 몇 기가바이트 단위의 거대 프로그램이 되면, 전체 프로그램을 메모리에 다 로딩하는 데에 걸리는 시간은 유의미하게 커지게 됩니다.
여기서 demand paging 으로 초기 메뉴까지 우선 메모리에 로딩하게 되면 고작 몇 초 안팎으로 사용자에게 초기 메뉴까지는 보여줄 수 있게 되는 셈이죠.
그런 의미에서의 faster response 라는 이야깁니다.

앞의 slow/fast 는 real-time running performance 를, 후자의 slow/fast 는 execution 에 따른 first response 의 의미입니다.

6. 9강 파트 1 25 페이지에서 page replacement 는 over-allocation 을 막으면서 page-fault 를 최소한으로 발생시키기 위한 것이라고 하셨는데 over-allocation 을 막는 이유는 앞의 COW 에서 이야기한 free-frame pool 이 성능향상에 도움을 주기 때문에 free-frame pool 을 유지하기 위함이라고 봐도 될까요?

-> 그렇게 보아도 무방합니다. free page 의 존재 여부에 따라 전체 시스템의 performance 가 달라지니까요.

7. 9강 파트 1 39 페이지에서 enhanced second-chance algorithm 은 dirty bit==0 인 page 는 just discard 하면 되기에 page transfer 이 적어서 victim page 로 선정하기 좋은 조건이라고 하셨는데 만약 dirty bit==0 이어도 anonymous memory 면 어떻게 하나요?(이 알고리즘은 victim page 선정일 뿐이어서 replacement 과정에서 page transfer 2 번을 할거라고 예상하고 있습니다.)

-> 해당 알고리즘은 딱히 anonymous memory 여부를 따지지 않습니다. 그 부분까지 고려하는 건 enhanced second chance algorithm 을 변형한 것이 되지요.
다만 생각해볼 때, anonymous memory 라면 빈번히 수정이 일어날 것이고 대부분은 dirty 상태가 유지될 것으로 봅니다.

8. 9강 파트 2 2 페이지에서 process 가 minimum 이하의 frame 을 할당받은 상황을 가정하고 있는데 이 경우 어떤 instruction 에서 할당받은 frame 수보다 많은 frame 을 요구하면 항상 page fault 가 나고 page fault 딜링 이후 instruction restart 로 인해 계속 같은 instruction 에서 page fault 가 되며 task 진행이 불가할거 같은데 맞나요?

-> instruction restart 를 하기 전에 지속적으로 메모리를 요구하게 될테니 거기서 프로세스가 멈추게 되겠지요.

멈춘다기 보다는 열심히 하드디스크와 메모리를 혹사시키는 stall 상태가 될 겁니다.

9. 9 강 파트 2 11 페이지의 working-set window 에 대한 예시로 10,000instruction 이라는 예시가 있는데 working-set window 는 fixed number of page reference 라고 되어 있어 10,000 reference 가 되야될거 같은데 맞나요?(instruction 마다 필요한 page reference 갯수가 다를 수 있다고 이전에 이야기 하셔서 위의 예시는 fixed number of 'page reference'로 보기 힘들거 같습니다)

-> 정확합니다. reference 로 봐야 할겁니다.
저 부분은 수정하겠습니다. 고마워요 :)

10. 9 강 파트 2 12 페이지에서 working-set window 가 10,00 인데 왜 15,000 개의 reference 를 체크하는 것인지 모르겠습니다.

-> 저 내용에서 파란색 비트 둘이 (memory-bit) 및 working-set window 가 가리키는 부분이고, 빨간색 비트는 현재 시간에서의 dirty bit 입니다.
10000 reference 의 기록을 가지고 있고, 매 5000 번의 reference 마다 업데이트 되는 식입니다.
본 예제의 경우 10001~15000 까지의 working set 을 트래킹하는 셈이 되겠네요.

이전의 설명과 연동하자면,
Current working scope 가 1~5000 reference 까지고,
저 1 시점 이전의 10000 개의 reference 들을 working set 으로 가지고 있는 겁니다.

11. 9 강 파트 2 15 페이지에서 5 번의 memory access 마다 process 에 할당된 frame 수를 바꾼다는 것이 맞나요? 그리고 deallocation priority 가 victim frame 선정에 쓰이는 priority 가 맞나요?

-> 맞습니다. 1, 6, 11, 16 번째 access 마다 각 프로세스에게 배정된 FRAME 을 바꾸는 겁니다.
Deallocation priority 역시 학생이 말한 게 맞습니다.

12. 9 강 파트 2 17 페이지 memory-mapped file 에서 memory-mapped file 은 kernel 영역 main memory 에 올라간다고 했는데 그럼 user process 가 kernel 영역의 main memory 를 건드리는 일이 되는데 문제가 없는건가요?(mapping 만 user process 의 virtual address space 에서 kernel 의 main memory 영역에 되고 실제 접근은 system call(mmap 같은)으로 하기때문에 mode change 를 하여 문제가 없다는 생각을 했는데 이게 맞을까요?)

-> 정확합니다. 실질적으로 memory mapped file 의 사용 예제를 보면, 어쨌건 '파일'이니 모두 system call 로 접근하게 되므로,
유저 입장에서 직접적인 접근은 배제되어 있습니다.

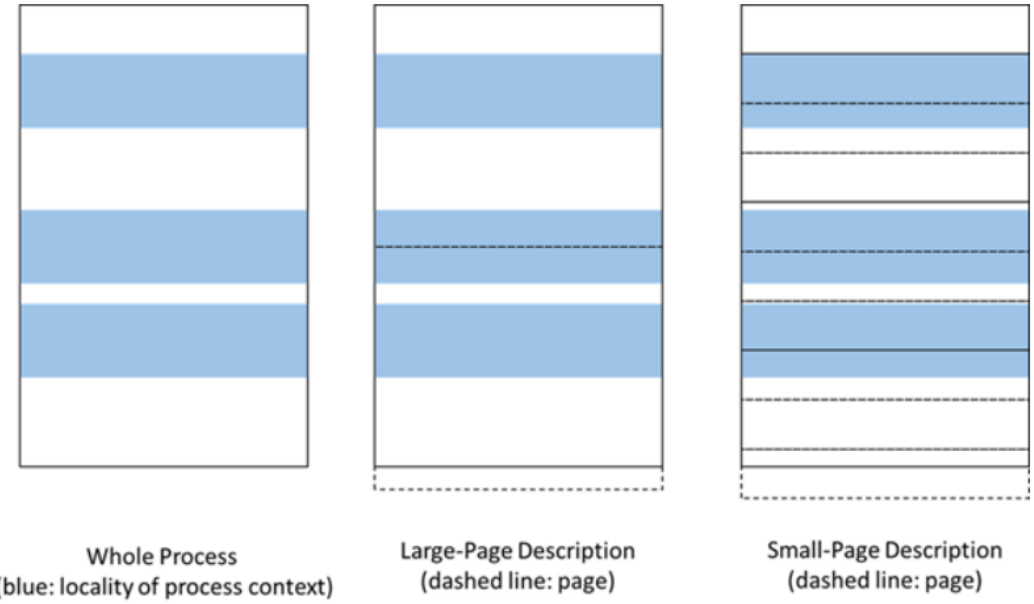
13. 9 강 파트 2 30 페이지에서 page size 가 커지면 spatial locality 가 줄어든다고 하셨는데 왜 그런지 모르겠습니다. 사실 ㅓ는 locality 는 source code 에서 정해지는거지 page size 에 따라 달라질거 같지 않습니다.

-> 아마도 locality 에 대한 정의가 달라서 그렇게 이해하는 것 같습니다.
당연히 locality 는 근본적으로 소스 코드 레벨에서 결정되는 것은 맞고,
해당 슬라이드는 run-time locality fitting 이 엄밀하게 말하면 더 정확합니다.

예를 들어서 page size 1MB 라고 가정하고, 실질적으로 한 프로세스가 이 중 100KB 만큼만 접근한다고
가정해봅시다.
일종의 for-loop 을 사용한 반복계산 이후 결과 출력하고 종료되는 프로세스가 이런 것과 비슷하겠네요.

그러면 한 페이지의 10%만큼만 지속적으로 활용되고, 이는 '메모리 공간 상 한 페이지의 일부만 집중적으로 활용'한 셈이 됩니다.
이를 Low Spatial Locality 라고 말한 게 해당 슬라이드입니다.
또한 Temporal 한 입장에서 봤을 때 해당 페이지는 집중적으로 reference 가 되니 temporal locality 는 높다고 판단할 수 있습니다.

이 로컬리티는 Resolution 과 밀접한 관련이 있습니다.
다시 말해 page size 를 줄이면 spatial resolution 이 증가하게 되고, 각 프로세스가 가지는 run-time locality 에 유연하게 대처하여
낭비되는 공간이 최소화되는 것을 돕는 반면 빈번한 page fault, I/O 오버헤드같은 것들을 가져오게 되는 것이죠.
Page size 를 늘리면 spatial resolution 이 감소하게 되니, 원하는 locality 에 피팅하기가 힘들게 됩니다.
대충 아래같은 그림인데...



도움이 됐으면 좋겠습니다.

긴 질문글 읽어주셔서 감사합니다.
수강생 김민석 올림