7단원. 가상 메모리 (최빈출 단원)

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

Q1. 가상 메모리란 무엇이고 어떤 이점을 주나요? ★★★

답안 프로세스의 논리 주소 공간과 물리 메모리를 분리하여, **프로세스 전체가 메모리에 없어도 실행**할 수 있게 하는 기법입니다. 이점은 ① 물리 메모리보다 큰 프로그램 실행 가능, ② 각 프로세스가 필요한 부분만 메모리에 올라가므로 더 많은 프로세스를 동시에 수용(멀티프로그래밍 정도 향상), ③ 프로세스 간 완전한 주소 공간 격리와 보호, ④ 라이브러리 공유·COW 같은 메모리 공유 최적화, ⑤ 프로그래머가 메모리 크기를 신경 쓰지 않는 프로그래밍 편의입니다. 구현은 요구 페이징이 표준입니다.

관련 개념 논리/물리 분리, 요구 페이징, 지역성(성립 근거), 주소 공간 격리

Q2. 요구 페이징(Demand Paging)과 페이지 폴트 처리 과정을 설명하세요. ★★★

답안 요구 페이징은 페이지를 미리 올리지 않고 **실제로 접근할 때** 메모리에 적재하는 방식입니다. 접근한 페이지가 메모리에 없으면(페이지 테이블 유효 비트 0) MMU가 **페이지 폴트** 트랩을 발생시키고, 커널이 ① 유효한 접근인지 검사(아니면 SIGSEGV) → ② 빈 프레임 확보(없으면 페이지 교체) → ③ 디스크에서 해당 페이지를 읽어 프레임에 적재 → ④ 페이지 테이블 갱신(유효 비트 1) → ⑤ 폴트를 일으킨 명령을 재실행합니다. 디스크 I/O가 포함되므로 페이지 폴트는 정상 메모리 접근보다 수만~수십만 배 느려, 폴트율이 성능을 좌우합니다.

관련 개념 유효/무효 비트, 트랩, 명령 재실행, 실효 접근 시간(EAT) 계산, major/minor fault

Q3. 페이지 교체 알고리즘 FIFO, OPT, LRU를 비교 설명하세요. ★★★

답안 FIFO는 가장 먼저 들어온 페이지를 교체합니다. 구현이 큐로 간단하지만 오래된 페이지가 자주 쓰이는 페이지일 수 있고, 벨레이디의 모순이 발생합니다. **OPT(최적)**는 앞으로 가장 오랫동안 사용되지 않을 페이지를 교체하며 폴트가 이론적 최소지만 미래를 알아야 해서 구현 불가능하고, 다른 알고리즘의 평가 기준으로 씁니다. **LRU**는 가장 오랫동안 사용되지 않은 페이지를 교체합니다. 과거로 미래(지역성)를 근사해 OPT에 가까운 성능을 내지만, 엄밀한 구현(카운터나 스택)은 모든 접근마다 갱신이 필요해 비싸므로 실제 OS는 참조 비트 기반 근사 LRU(clock)를 씁니다. 참조 문자열로 폴트 수를 세는 계산 문제가 단골입니다.

관련 개념 참조 문자열 계산, 벨레이디의 모순(FIFO), clock(second chance), LRU 구현 비용

Q4. 벨레이디의 모순(Belady's Anomaly)이란 무엇인가요? ★★

답안 프레임 수를 **늘렸는데** 페이지 폴트가 오히려 **증가**하는 직관에 반하는 현상으로, FIFO에서 발생합니다(예: 참조열 1,2,3,4,1,2,5,1,2,3,4,5는 프레임 3개일 때 9회, 4개일 때 10회 폴트). LRU나 OPT 같은 **스택 알고리즘**(프레임 n개일 때

메모리 내용이 n+1개일 때 내용의 부분집합이 되는 성질)에서는 발생하지 않습니다. FIFO는 페이지의 사용 여부와 무관하게 교체하기 때문에 이런 역전이 생깁니다.

관련 개념 스택 알고리즘, 부분집합 성질, FIFO의 한계

Q5. 스래싱(Thrashing)의 원인과 해결책을 설명하세요. ★★★

답안 스래싱은 프로세스들이 실행보다 **페이지 폴트 처리(스왑 인/아웃)에 대부분의 시간을 쓰는** 상태입니다. 원인은 멀티프로그래밍 정도가 과해 각 프로세스가 필요한 최소 프레임(워킹셋)조차 확보하지 못하는 것입니다. 악순환이 특징적입니다: 폴트 증가 → CPU 이용률 하락 → OS가 이용률을 높이려 프로세스를 더 투입 → 프레임 부족 심화 → 폴트 폭증. 해결은 ① 워킹셋 모델로 각 프로세스의 필요 프레임을 보장하고 부족하면 일부 프로세스를 중단(스왑 아웃), ② 페이지 폴트 빈도(PFF)가 상한을 넘으면 프레임 추가·하한 미만이면 회수, ③ 근본적으로 메모리 증설 또는 워크로드 축소입니다. 운영 관점에서는 vmstat의 si/so 지속 발생과 CPU 이용률 하락 조합이 스래싱 신호입니다.

관련 개념 워킹셋 모델, PFF, 멀티프로그래밍 정도, vmstat 진단

Q6. 워킹셋(Working Set) 모델을 설명하세요. ★★

답안 워킹셋은 최근 시간 창 Δ 동안 프로세스가 참조한 페이지들의 집합으로, 지역성에 근거해 '지금 이 프로세스가 실제로 필요로 하는 메모리'를 나타냅니다. OS는 각 프로세스의 워킹셋 크기 합이 가용 프레임을 넘지 않도록 멀티프로그래밍 정도를 조절하고, 넘으면 일부 프로세스 전체를 중단시켜 스래싱을 예방합니다. Δ가 너무 작으면 지역성을 다 못 담고, 너무 크면 여러 지역성이 섞입니다. 실무에서 컨테이너 메모리 limit 설정이나 JVM 힙 사이징도 '워킹셋을 물리 메모리에 담는다'는 같은 사고방식입니다.

관련 개념 지역성 창 Δ, 프레임 할당, 스래싱 예방, RSS(Resident Set Size)

Q7. 프레임 할당 방식(균등/비례/전역/지역 교체)을 설명하세요. ★

답안 **균등 할당**은 모든 프로세스에 같은 수의 프레임을, **비례 할당**은 프로세스 크기(또는 우선순위)에 비례해 배분합니다. 교체 범위 기준으로 **전역 교체**는 다른 프로세스의 프레임도 빼앗을 수 있어 메모리 활용이 유연하지만 프로세스의 폴트율이 자신이 통제할 수 없는 외부 요인에 좌우됩니다. **지역 교체**는 자기 프레임 안에서만 교체해 성능이 예측 가능하지만 유휴 프레임 활용이 안 됩니다. 리눅스는 기본적으로 전역 교체이며, cgroup 메모리 제한이 사실상 지역 교체 경계를 만듭니다.

관련 개념 전역 vs 지역 교체, cgroup 메모리 제한, 최소 프레임 수

Q8. 페이지 크기가 크거나 작을 때의 트레이드오프는? ★★

답안 작은 페이지: 내부 단편화가 적고 필요한 부분만 정밀하게 적재하지만, 페이지 수가 많아져 페이지 테이블이 커지고 TLB 커버리지가 줄며 폴트 횟수가 늘어납니다. **큰 페이지**: 페이지 테이블이 작고 TLB 히트율이 높아지며 디스크 I/O가 효율적(한 번에 많이)이지만 내부 단편화가 커집니다. 표준은 4KB이고, 메모리를 수십 GB 쓰는 DB·가상화 서버에서는

TLB 미스 감소를 위해 2MB/1GB **huge page**를 명시적으로 씁니다. 리눅스의 THP(Transparent Huge Pages)는 자동화 기능이지만 지연 스파이크를 유발할 수 있어 DB 벤더들은 비활성화를 권장하기도 합니다.

관련 개념 TLB 커버리지, huge page, THP와 DB 튜닝, 내부 단편화

Q9. mmap이란 무엇이고 read/write와 어떻게 다른가요? ★★

답안 mmap은 파일이나 익명 메모리를 프로세스 주소 공간에 **매핑**하는 시스템 콜입니다. 매핑 후에는 포인터로 메모리를 읽고 쓰듯 파일에 접근하며, 실제 적재는 페이지 폴트를 통해 요구 페이징으로 이뤄집니다. read/write는 커널 버퍼(페이지 캐시)에서 사용자 버퍼로 **복사**가 일어나고 매번 시스템 콜이 필요한 반면, mmap은 페이지 캐시를 직접 매핑해 복사와 시스템 콜을 줄입니다. 대용량 파일 임의 접근, 프로세스 간 공유 메모리(MAP_SHARED), 라이브러리 로딩에 쓰입니다. 다만 폴트 비용과 TLB 영향이 있어 순차 대량 읽기에서는 read가 나을 수 있습니다.

관련 개념 페이지 캐시, zero-copy, MAP_SHARED/PRIVATE, 라이브러리 로딩(ld.so)

Q10. 리눅스 OOM Killer는 언제, 어떻게 동작하나요? (운영 연계) ★★★

답안 커널이 페이지 회수·스왑으로도 메모리를 확보할 수 없는 극한 상황에서, **oom_score가 가장 높은 프로세스를 SIGKILL로 강제 종료**해 시스템을 살리는 최후 수단입니다. 점수는 메모리 사용량 위주로 계산되며, oom_score_adj로 프로세스별 가중치를 조정할 수 있습니다(-1000이면 제외). 이 배경에는 리눅스의 **메모리 오버커밋** 정책이 있습니다. malloc 시점에는 가상 주소만 주고 실제 페이지는 첫 접근 때 할당하므로, 약속한 메모리가 실제로 모자라는 순간이 올 수 있는 것입니다. 운영 시 dmesg에서 OOM 로그를 확인하고, 컨테이너 환경에서는 cgroup 메모리 limit 초과 시 컨테이너 단위 OOM kill이 발생합니다(쿠버네티스 OOMKilled 상태).

관련 개념 오버커밋(vm.overcommit_memory), oom_score_adj, dmesg, K8s OOMKilled, 메모리 limit

---

Q11. 페이지 캐시란 무엇이고, free 명령의 available 메모리와 어떤 관계인가요? (운영 연계) ★★★

답안 페이지 캐시는 디스크에서 읽은 파일 데이터를 메모리에 캐싱하는 커널 기능으로, 같은 데이터의 재접근을 디스크 I/O 없이 처리합니다. 리눅스는 남는 메모리를 최대한 페이지 캐시로 활용하므로 free 명령에서 free(완전 미사용)가 작아 보여도 문제가 아닙니다. 캐시는 메모리가 필요하면 즉시 회수 가능하기 때문입니다. 그래서 실질 가용 메모리는 **available** 컬럼(free + 회수 가능한 캐시)을 봐야 합니다. "리눅스 서버 메모리가 꽉 찼어요"라는 신고의 상당수가 페이지 캐시를 오해한 것으로, 인프라 면접의 단골 실무 질문입니다. 쓰기 캐시(dirty page)는 주기적으로 디스크에 반영(writeback)됩니다.

관련 개념 buff/cache, available, dirty page와 writeback, drop_caches, vm.dirty_ratio

---

Q12. 세그멘테이션 폴트(SIGSEGV)는 왜 발생하며, 페이지 폴트와 무엇이 다른가요? ★★

답안 **페이지 폴트**는 접근한 페이지가 메모리에 없을 때 발생하는 정상적 이벤트로, 커널이 페이지를 적재해 투명하게 해결합니다. **세그멘테이션 폴트**는 프로세스가 자신에게 **허용되지 않은** 주소(매핑되지 않은 영역, 읽기 전용 페이지에 쓰기 등)에 접근했을 때 커널이 SIGSEGV 시그널을 보내 프로세스를 종료시키는 오류입니다. 즉 페이지 폴트 처리 과정에서 '유효하지 않은 접근'으로 판정되면 SIGSEGV가 됩니다. 원인은 널 포인터 역참조, 해제된 메모리 사용, 배열 범위 초과, 스택 오버플로우 등이며, 코어 덤프와 gdb로 디버깅합니다.

관련 개념 유효성 검사, 널 포인터, 코어 덤프, minor/major/invalid fault 구분

Q13. 요구 페이징 환경에서 실효 접근 시간(EAT)을 계산해보세요. ★★

답안 EAT = (1−p) × 메모리 접근 시간 + p × 페이지 폴트 처리 시간 (p는 폴트 확률). 예: 메모리 접근 200ns, 폴트 처리 8ms(=8,000,000ns), p=0.001이면 EAT = 0.999×200 + 0.001×8,000,000 ≈ 8,200ns로 메모리 접근보다 **40배** 느려집니다. 성능 저하를 10% 이내로 유지하려면 220 ≥ 200 + p×8,000,000에서 p ≤ 0.0000025, 즉 40만 번 접근에 1회 미만의 폴트만 허용됩니다. 페이지 폴트가 얼마나 치명적인지, 왜 교체 알고리즘과 스래싱 방지가 중요한지를 수치로 보여주는 문제입니다.

관련 개념 EAT 공식, 폴트 확률의 민감도, 디스크 vs 메모리 속도 격차

Q14. clock(Second Chance) 알고리즘을 설명하세요. ★★

답안 LRU를 저비용으로 근사하는 실용 알고리즘입니다. 프레임들을 원형 큐로 놓고 포인터(시곗바늘)가 순회하며, 각 페이지의 **참조 비트**를 검사합니다. 비트가 1이면(최근 사용됨) 0으로 바꾸고 한 번의 기회를 더 주며 다음으로 넘어가고, 0이면(한 바퀴 동안 미사용) 교체 대상으로 선택합니다. 하드웨어가 접근 시 참조 비트를 자동으로 1로 세팅해주므로 접근마다 소프트웨어 갱신이 필요한 엄밀한 LRU보다 훨씬 쌉니다. 변형으로 참조 비트+수정 비트를 함께 보는 개선판(NRU 계열)이 있으며, 수정된 페이지는 디스크에 써야 해서 교체 비용이 크므로 우선순위를 낮춥니다.

관련 개념 참조 비트, 수정(dirty) 비트, 원형 큐, LRU 근사

Q15. 컨테이너/클라우드 환경에서 가상 메모리 관련해 주의할 점은? (인프라 연계) ★★

답안 ① **cgroup 메모리 limit**: 컨테이너가 limit을 초과하면 호스트 메모리가 남아도 OOM kill됩니다. JVM 등 런타임이 컨테이너 limit을 인식하도록 설정해야 합니다(최신 JVM은 자동 인식). ② **request/limit 설계**: 쿠버네티스에서 request 대비 limit이 크면 노드 메모리 오버커밋으로 이웃 워크로드 영향(noisy neighbor)이 생길 수 있습니다. ③ **스왑**: 쿠버네티스는 전통적으로 스왑 비활성화를 요구해 왔으므로, 메모리 부족이 곧 OOM으로 직결됩니다. ④ **페이지 캐시 공유**: 같은 노드 컨테이너들이 페이지 캐시를 공유하므로 메모리 계측(RSS vs working set)이 혼동되기 쉽습니다. 모니터링은 container_memory_working_set_bytes 기준으로 보는 것이 표준입니다.

관련 개념 cgroup v2, K8s requests/limits, OOMKilled, working set 메트릭, JVM 컨테이너 인식
