TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – GJ (박현우)
uc820@naver.com

목차

Chapter 3 태스크 관리

7-2 실시간 태스크 스케줄링

1) CPU를 어떤 태스크가 사용하도록 하나?

⇒ 가장 급한 태스크가 먼저 수행하도록!

우선 순위가 높다.

2) 어떤 기준으로 태스크를 고르나?

⇒ task_struct 및에 policy, prio, rt_priority

스케줄링 정책

- 일반태스크 = sched_NORMAL

⇓

CFS 스케줄링 → Vruntime을 계산한다.

- SCHED_RR은 우선 순위를 고려한다면 동일한 타임 슬라이스를 갖지 않는다.

3) 가장 높은 태스크를 어떻게 찾나?

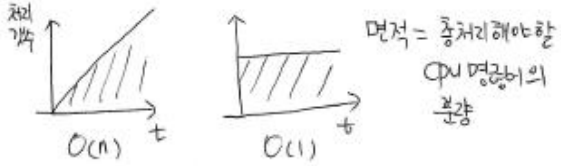
⇒ 모든 태스크는 이중 연결 리스트에 연결 되어 있으므로 이를 활용하면 모든 태스크에 접근 가능

List_head

but, 스케줄링 시간이 O(n) 이다.

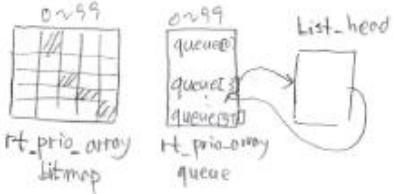
⇒ 태스크가 많아지면 소요시간을 예측 X

So, Hash 나 Map으로 자료구조를 바꿈. O(1)



∴ 갯수가 무식하게 많을 땐 O(1)이 좋다.

4) 태스크 접근 속도의 단점극복은?



⇒ 우선순위에 해당하는 비트를 1로 set 하고 해당 태스크를 queue에 삽입한다. 비트연산 후 나온 비트 1을 가지고 큐에서 해당 태스크를 바로 찾으면 스케줄링 시간이 O(1)이 된다.

5) Deadline 정책이란?

⇒ deadline이 가장 가까운 태스크를 스케줄링 한다

ex) 모니터 그리기 $\frac{1}{30}$초 = 0.033초

실제 걸리는 시간 = 0.01초
0.023초
0.02초 (context switch)

⇒ deadline을 이용하여 Rbtree에 정렬 됨

- 삽입과 삭제가 빈번서 유리

⇒ DEADLINE 스케줄링 기법에서 사용하는 자료구조는 struct rt_rq, struct dl_rq에 존재한다.

FIFO, RR

정적우선순위 (kernel DEADLINE)

7-3 일반 태스크 스케줄링 (CFS)

completely fair scheduler

1) 어떤 스케줄링이 완벽하게 공정한 것일까?

⇒ A와 B의 CPU 사용시간이 1:1로 항상 같아야 한다.

가상 시간 (구동시간 X)

ex)

우선순위 ↑	우선 순위 ↓
가상 0.1초	= 가상 0.1초
실제시간 0.2초	≠ 실제시간 0.05초

if) 시간 단위가 너무 길면 태스크의 반응성 ↓
반대로 너무 짧으면 문맥교환 비용 ↑

cpu_context_save

2) 위 상황을 위해 도입한 'Vruntime'이란?

⇒ 우선순위를 고려, 일반 태스크의 실제 우선 순위 100~139; 사용자 수준 -20~19 나중에 커널이 (priority+120)으로 변환

1024

*Vruntime += physical runtime × (weight / weight_curr)

rate

즉, 가중치↑ 시간이 빨리 찬다 → Vruntime↓

우선 순위가 낮다

∴ 우선순위 문제와 비용문제 해결!!

3) 스케줄링이 되는 태스크를 어떻게 효율적으로 찾나?

⇒ 가장 작은 Vruntime 값을 가지는 태스크가 가장 과거에 CPU를 사용함.

↳ tree 구조 가장 왼쪽에 있는 게 값이 가장 작다.

4) 너무 자주 스케줄링이 발생하지 않나?

⇒ 빈번한 스케줄링을 막기 위해 스케줄링 간 최소 지연 시간이 있다.

5) 스케줄러는 언제 호출되나?

⇒ 2가지 ① nice_System_call (직접 호출)

② flags (need_resched)

thread_info

|| set

6) 철수 99개 태스크, 영희 1개 태스크 CPU 사용이 불공정하지는 않나?

⇒ CFS는 그룹 스케줄링 정책을 지원함.

9 태스크와 시그널

- 시그널은 태스크에게 비동기적 사건 발생을 알림.

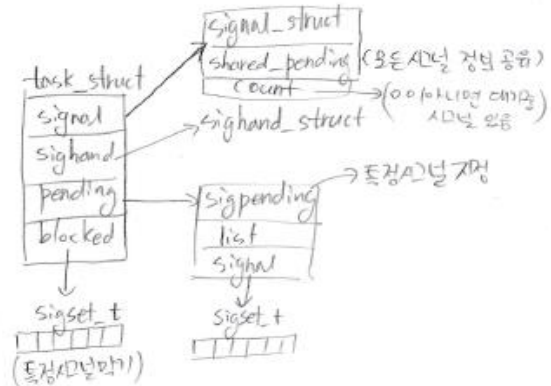
1) 태스크가 시그널을 원활히 처리하려면?

⇒ ① 다른 태스크에게 시그널을 보낼 수 있어야함.

② 시그널이 오면 수신할 수 있어야 함. (task_struct에 signal, pending 변수)

③ 시그널이 오면 시그널 처리 함수를 지정할 수 있어야함.

(sys_signal(), task_struct 및 sighand변수)



2) 인터럽트와 트랩 / 시그널 간의 차이점은?

⇒ 커널에게 알림 / 태스크에게 알림

system call / ctrl+C

Chaper 4 메모리 관리

1 메모리 관리 기법과 가상메모리

1) 물리 메모리의 한계를 극복하기 위해 만들어 진 것은?

⇒ 가상 메모리 (virtual memory)

2) 얼마 만큼의 가상주소 공간을 제공해야 할까?

⇒ 32비트 cpu 경우 2^{32} (4GB)
64비트 cpu 경우 2^{64} (16EB)

* 한 가지 주의할 점
⇒ 물리적으로 4GB의 메모리를 모두 사용자 태스크한테 제공하는 것은 아님.

* 추가내용
⇒ ① 메모리 배치정책이 불필요
(물리 메모리에 직접 안해도 됨.)
② 태스크의 빠른 생성이 가능하다.
(task_struct 복사하면 됨.)
③ 태스크간 메모리 공유/보호가 손쉽다.
(운영체제)

Compiler → 가상메모리 —(sys_exec)→ 물리메모리

2 물리 메모리 관리 자료구조

• 리눅스는 시스템에 존재하는 전체 물리 메모리에 대한 정보를 가지고 있어야 한다.
When? 부팅시에 ~

1) 한정된 용량의 메모리를 효율적으로 쓰려면?

⇒ paging, buddy, 가상메모리, SMP

2) 모든 CPU가 메모리와 입출력 버스 등을 공유하는 SMP 구조에는 무엇이 있고 문제점은?
(Symmetric Multiprocessing)

⇒ 문제점: 여러 cpu가 메모리 공유로 인해 병목현상이 발생 (발열 등.)

그렇기 때문에, NUMA 가 생겨났고 기존 시스템은 UMA라 함.

2-1 Node

1) 리눅스에서 접근 속도가 같은 메모리 집합을 무엇이라 하나?

⇒ 뱅크(bank) = Node
(물리) (가상)

ex) UMA구조는 한 개의 뱅크 = Node 1개
NUMA 구조는 복수 개의 뱅크 = Node 여러개
→ pg List_data (접근)
→ contig_page_data (접근)

* 무엇으로 접근하든, 결국 pg_data_t 구조체로 표현

1) 해당 Node에 속한 실제 물리 메모리 양
(node_present_pages)
2) 물리메모리의 메모리 맵의 위치
(node_start_pfn)
3) Zone 구조체를 담기위한 배열과 갯수)
(node_zones) (nr_zones)

* Cache 활용 ⇒ '성능 향상'
⇒ 물리 메모리 할당 요청되면, 요청한 태스크가 수행되고 있는 cpu와 가까운 노드에 메모리 할당.

2-2 Zone

1) Zone은 무엇인가?

⇒ node의 (메모리) 일부분을 따로 관리할 수 있도록 만든 자료구조가 Zone이다.

2) 노드에 존재하는 물리메모리 중 16MB이하 부분은 특별한 관리를 받는다. 뭐라 부르나?

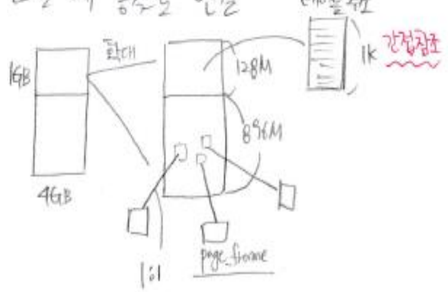
⇒ ZONE_DMA (video, network, sound) - device driver
Alsa, V4L2

그렇다면, 16MB이상은?

⇒ ZONE_NORMAL

3) 리눅스의 가상주소공간과 물리 메모리 공간을 1:1로 연결한다면 아무리 해도 1GB이상 접근이 안된다. 수 GB 사용하면 너무 치명적 약점이지 않나?

⇒ 물리메모리 1GB이상이면 896MB까지는 커널의 가상주소공간과 1:1 연결, 나머지 부분은 필요할 때 동적으로 연결



∴ 이러한 방식을 ZONE_HIGHMEN이라고함.

4) Zone에 속한 물리 메모리 관리는 어떻게?

⇒ ① ZONE 구조체를 사용한다.
(물리메모리 시작주소, 크기, 버디 할당자의 free_area)
② watermark 와 VM_stat 사용 통해서 빈 공간을 만듦
→ (스왑조정후 메모리해제)
→ • 현재 사용중인 page
• 현재 비어있는 page
③ 프로세스가 Zone에 요청 했으나 page가 부족하면 wait_queue에 넣고 해싱해서 wait_table 변수가 가리킴.

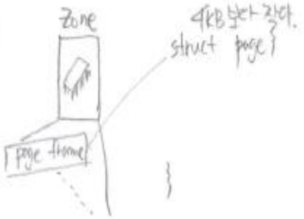
2-3 page frame (부팅시)

1) 물리 메모리의 최소 단위는?

⇒ page frame → page 구조체에 의해 관리됨.

2) 리눅스가 시스템 내의 모든 물리 메모리에 접근 하려면?

⇒ 모든 페이지 프레임(4KB)당 하나씩 page 구조체가 생긴다. 이는 부팅시 구축되어 물리 메모리에 특정위치에 저장됨.
→ node_mem_map을 통해 접근이가능 (0번 노드 = bank 1개)



[3] Buddy와 Slab (알고리즘)

1) 시스템 내의 물리 메모리가 아닌, 리눅스가 가진 물리 메모리는 어떻게 할당 또는 해제 하나?

⇒ 1Byte 단위의 할당은 메타데이터가 너무 방대해진다. 그래서 물리메모리 최소 단위인 4KB로 결정함.

But, 4KB보다 작다면 내부 단편화 발생! 그래서, *슬랩 할당자를 도입.

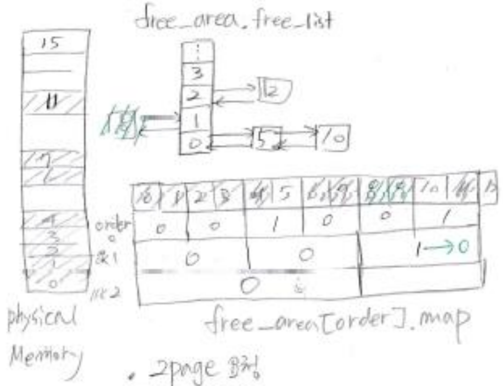
반면에, 4KB보다 크다면 4MB($2^n \times 4KB$) 방식으로 할당.
⇓
*버디 할당자

[3-1] Buddy Allocator

1) 버디 할당자는 어디에 있나?

⇒ zone 구조체에 free_area[]로 구축된다.

2) 버디 할당자의 동작 원리는?



• 2page 요청

[3-2] Lazy Buddy

1) 왜 Lazy Buddy가 등장 했나?

⇒ 페이지 프레임을 할당/해제를 반복하면 큰 페이지를 쪼개서 할당하고 해제를 하면 다시 합쳐야 한다. 이러하면 굉장히 많은 오버헤드가 발생한다.
(slab)

So, free area에서 비트맵 포인터가 nr_free로 바뀜.
(사용 중이지 않은 페이지 프레임 개수)

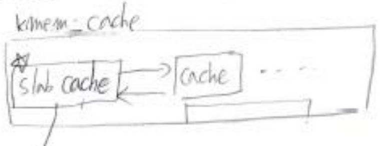
2) 그렇다면 버디가 어떻게 동작하나?

⇒ zone 마다 있는 watermark 값과 사용가능한 페이지 수를 비교하고 메모리 부족시 ... free_pages() 함수로 메모리 반납.

[3-3] Slab Allocator

1) 4KB(페이지 프레임)보다 작은 메모리를 요구 한다면??

⇒ 4KB의 공간을 미리 만들고 32byte 공간 씩 할당을 해주고 자주 사용되면 cache 형식으로 가지고 있는다.



제일먼저 할당 받아야만 kmem_cache를 받아 다양한 캐시를 생성할 수 있다.

2) 외부 인터페이스 함수

vmalloc() ← *kmalloc() / kfree() ⇒ 물리적으로 연속!!
속도↓ 공간이 큼
device ↓ 버디
↳ 메모리 순차배치
(Cache) Locality 속도빠름

[4] 가상 메모리 관리 기법

Sticky bit 10bit / 10bit / 12bit 간접참조

1) 가상 주소 공간을 어떻게 할당/해지 하나?

⇒ 태스크는 자신의 고유한 가상 메모리를 갖음
커널이 → mm_struct로 관리해야함.
즉, text, data, heap, stack 정보를 알아야 함

2) mm_struct 자료구조가 관리하는 정보는?

① vm_area_struct로 세그먼트를 관리

② 효율적인 관리를 위해 RB트리의 시작 mm_rb와 최근 접근한 vm_area_struct를 가리키는 mmap_cache 변수가 존재

③ 주소변환을 위한 페이지 디렉토리의 시작주소 PGD 관리
(성능 극대화)

④ 가상메모리구조에 대한 변수
(28byte)

3) vm_area_struct 자료구조를 구체적으로 본다면?

⇒ ① 세그먼트의 시작, 끝 주소 (vm_start, end), 접근제어 플래그 (vm_flags) 변수를 갖는다

② 실제 실행파일 위치 정보 vm_file, vm_offset 변수
↓
pgoff → 기계어가 어디까지 읽었는지.

So, 이 변수들은 페이지 폴트가 발생할 때 어떤 파일을 읽어야 할지를 결정

[5] 가상메모리와 물리 메모리의 연결 및 변환

1) 가상메모리와 물리 메모리를 연결시키려면?

⇒ ELF 포맷의 헤더를 읽어 물리 메모리 올려 놓을지를 결정 후, 가상주소에 연결할지는 정해진 rule에 따른다.

* 디스크에서 실행 파일 내용을 모두 읽어 물리 메모리에 올리는 게 아님
⇓
Demand on paging

NPTL의 이점

1. 커널 전용과 유저 전용이 1:1로 매칭.

2. Kernel Stack을 활용함.

3. 용량이 너무 크면 용량을 제한.

기존에는 커널 전용과 유저 전용이 1:N로 매칭
용량이 너무 크면, 캐쉬가 깨짐.
