운영체제 **3**팀 프로젝트 중간보고서

전자정보공학부 **20140653** 허성현

전자정보공학부 **20192570** 김도형

전자정보공학부 **20192610** 윤재선

소프트웨어학부 **20192944** 권정태

목차

1. 프로젝트 주제 및 목표

주제: Memory Allocation optimize process

목표

- 본 프로젝트는 xv6 운영체제 상에서 바이트 단위 Memory Allocator를 구현하는 것을 목표로 합니다.

기존 방식에서 개선할 점

- 기존 운영체제의 비효율적인 페이징 방식
- 운영체제의 내부 단편화 (Inner Fragmentation)을 개선하고자 함.
- Memory Allocator 구현.

세부 목표

- 바이트 단위 Memory Allocator 구현: ***8, 16, 32, 64, 128, 256, 512, 1024, 2048*** byte
- slab 메모리 allocate함수 kmalloc과 deallocate함수인 kmfree함수를 구현하여 더 작은 단위의 메모리 관리가 가능하도록 구현.
- 비트맵을 활용한 메모리 관리: 할당된 메모리와 해제된 메모리를 비트맵으로 관리.

주제 추가 설명

- 수업에서는 운영체제의 메모리 할당이 프로그램이 실행될 때 필요한 메모리 공간을 제공하고 관리하는 기능임을 배웠습니다. 메모리 할당은 주로 정적 할당과 동적 할당으로 나뉘며, 동적 할당은 프로그램 실행 중 메모리를 요청하고 해제하는 과정을 포함합니다.
- 이 과정에서 내부 단편화와 외부 단편화 문제가 발생할 수 있으며, 이를 해결하기 위해 다양한 전략이 사용됩니다. 대표적인 할당 전략으로는 최초 적합, 최적 적합, 최악 적합 등이 있으며, 효율적인 메모리 관리를 위해 메모리 풀, 슬랩 할당자, 버디 시스템 등의 기법이 도입됩니다.
- 본 프로젝트에서는 이러한 메모리 할당 기법을 xv6 운영체제 상에서 slab allocator를 실제로 구현해봄으로써, 운영체제의 성능을 높이고 자원을 효율적으로 사용할 수 있도록 합니다.

2. 프로젝트 진행 과정

xv6 시스템 콜 구현 및 스케줄링 방식 확인

https://lively-snowflake-01d.notion.site/5-8-071fe294582b43638d0201f955fe7596

xv6 memory allocation방식 확인

kalloc을 활용한 page allocate

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

kalloc 함수는 kmem.freelist에서 자유 페이지를 하나 할당하고, 필요시 동시성 제어를 위해 락을 사용합니다. 함수는 할당된 페이지의 주소를 반환하고, 할당할 수 있는 페이지가 없다면 NULL을 반환합니다. 이를 통해 xv6 커널은 동적으로 메모리 페이지를 할당합니다.

kfree를 활용한 page deallocate

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

kfree 함수는 주어진 페이지 주소 v를 해제하고, 자유 페이지 리스트에 반환합니다. 이때 주소의 유효성을 검사하고, 디버깅을 위해 페이지를 특정 값으로 채우며, 동시성 제어를 위해 필요시 락을 사용합니다. 해제된 페이지는 다시 할당될 수 있도록 자유 페이지 리스트에 추가됩니다.

3. 프로젝트 진행 척도

현황

A. 설계 단계: 전체 구조 설계 완료. Memory Allocator, 초기화 함수 설계 완료

B. 구현: slab.c, slab.h, sysslab.c, slabtest.c 파일에 주요 기능 구현 중

    a. slabinit(): 비트맵과 슬랩 캐시 초기화.

    b. slabdump(): 슬랩 할당기 상태 출력.

    c. 그 외 bitmap 관련 함수들

C. 구현 중:

    a. kmalloc(): 주어진 크기의 메모리 할당.

    b. kmfree(): 할당된 메모리 해제.

D. 테스트 예정 Task

    - slabtest 시스템 호출을 통해 다양한 시나리오 테스트.

4. 프로젝트 성과

성과 요약

- 효율성 증대: 특정 바이트 단위의 메모리 할당으로 인해 메모리 사용 효율성이 크게 향상되었음.
- 안정성 향상: 비트맵을 이용한 메모리 관리로 메모리 누수 및 중복 할당 문제 최소화.
- 유연성 제공: 다양한 크기의 메모리 블록을 지원하여 다양한 애플리케이션 요구에 대응 가능.

구체적인 성과

- 메모리 할당/해제 속도 개선: 기존 kalloc() 대비 작은 단위의 메모리 할당/해제를 위한 slab memory 초기화 함수 구현.

```
void slabinit()
{
    acquire(&stable.lock);
    stable.slab[0].size=8;
    stable.slab[0].num_objects_per_page=PGSIZE/stable.slab[0].size;
    stable.slab[0].num_used_objects=0;
    stable.slab[0].num_free_objects=stable.slab[0].num_objects_per_page*64;
    //allocate one page for bitmap, allocate one page for slab cache
    stable.slab[0].bitmap=stable.slab[0].page[0];
    stable.slab[0].bitmap=kalloc();
    memset(stable.slab[0].bitmap,0,PGSIZE);
    stable.slab[0].page[1]=kalloc();
    stable.slab[0].num_pages=1;
    release(&stable.lock);

    acquire(&stable.lock);
    for(int i=1;i<NSLAB;i++)
    {
        stable.slab[i].size=stable.slab[i-1].size*2;
        stable.slab[i].num_objects_per_page=PGSIZE/stable.slab[i].size;
        stable.slab[i].num_used_objects=0;
        stable.slab[i].num_free_objects=stable.slab[i].num_objects_per_page*MAX_PAGES_PER_SLAB;
        //allocate one page for bitmap, allocate one page for slab cache
        stable.slab[i].bitmap=stable.slab[i].page[0];
        stable.slab[i].bitmap=kalloc();
        memset(stable.slab[i].bitmap,0,PGSIZE);
        stable.slab[i].page[1]=kalloc();
        stable.slab[i].num_pages=1;
    }
    release(&stable.lock);
}
```

<그림 1: 구현한 코드 중 일부 slabinit()>

- 메모리 할당/해제를 용이하게 하기 위한 bitmap operation 함수들 구현

```c
int set_bit(int num,int i)
{
    return num | (1<<i);
}
int get_bit(int num,int i)
{
    return ((num&(1<<i))!=0);
}

int clear_bit(int num,int i)
{
    int mask=~(1<<i);
    return num&mask;
}

unsigned int getnumofBits(unsigned int n)
{
    unsigned count=0;
    while(n!=0)
    {
        n>>=1;
        count+=1;
    }
    return count;
}
```

```c
int setBitmap(int slabIdx)
{
    struct slab *s;
    s=&stable.slab[slabIdx];
    for(int j=0;j<PGSIZE;j++)
    {
        if(s->bitmap[j]==0xFF)
            continue;
        for(int k=0;k<=7;k++)
        {
            if(!(s->bitmap[j]&(1<<k)))
            {
                s->bitmap[j]=set_bit(s->bitmap[j],k);
                return returnOffset(j,k);
            }
        }
    }
    return 0; //Unable to find empty space of bitmap
}

bool clearBitmap(int slabIdx,int offset)
{
    struct slab *s;
    s=&stable.slab[slabIdx];
    bool checkbit=true;
    int row=getRow(offset);
    int column=getColumn(offset);
    if(get_bit(s->bitmap[row],column))
        s->bitmap[row]=clear_bit(s->bitmap[row],column);
    else
        checkbit=false;
    return checkbit;
}
```

<그림 2: 구현한 코드 중 일부 bit 관련 함수들>

5. 문제 해결

1.메모리 조각화

작은 단위로 메모리를 할당할 때 발생하는 메모리 조각화로 인해 디버깅 문제 발생

Solution

비트맵을 활용한 메모리 관리 기법을 도입하여 디버깅 할 수 있도록 설계

1. 메모리 할당 상태를 8, 16, 32, 64, 128, 256, 512, 1024, 2048 바이트 크기별 비트맵을 만든다. 슬랩 할당기를 구현하여, 메모리의 세부 상태를 정확히 파악
2. 8, 16, 32, 64, 128, 256, 512, 1024, 2048가 아닌 메모리를 allocate하더라도 그에 적합한 slab메모리 할당 (1000byte 를 할당할 때는 1024byte할당)
3. 비트맵을 사용하여 메모리 할당 상태를 관리하고, 필요 시 동적으로 메모리 블록을 추가로 할당

2. 메모리 누수 문제 해결

메모리 해제 시 제대로 메모리가 반환되지 않아 메모리 누수가 발생할 수 있었습니다.

Solution

모든 메모리 할당 요청에 대해 대응하는 메모리 해제 처리를 정확히 구현

1. kmalloc() 함수에서 할당된 메모리 블록에 대한 정보를 비트맵에 기록하고, kmfree() 함수에서 이 정보를 기반으로 메모리 해제
2. 메모리 해제 시, 비트맵에서 해당 메모리 블록의 비트를 클리어하여 다시 사용 가능한 상태 전환

6. 추후 계획

1. Slab_test.c 구현

사용자 레벨에서 메모리를 할당하고 해제하는 기능을 검증하기 위해 `slab_test.c` 구현

세부 계획

1. 테스트 함수 작성: 다양한 크기의 메모리 블록을 할당하고 해제하는 테스트 함수들을 작성합니다. 이를 통해 할당기의 기능을 검증

2. Slab 메모리 할당 구현

Slab 할당기를 통해 실제로 메모리를 할당하고 해제하는 기능 완성

세부 계획

1. kmalloc 함수 구현: 사용자 레벨에서 다양한 크기의 메모리 블록을 요청할 수 있는 `kmalloc` 함수를 구현합니다. 먼저 페이지를 allocate한 뒤에 사용자 레벨에서 요청한 byte크기에 맞게 메모리를 할당합니다.
2. kmfree 함수 구현: 할당된 메모리를 해제할 수 있는 `kmfree` 함수를 구현합니다. 이 함수는 사용자가 더 이상 필요하지 않은 메모리를 반환하고, 비트맵을 갱신하여 메모리 관리 효율성 유지합니다.
3. slab memory dump함수 구현: 메모리가 올바르게 할당되고 해제 되는 지 확인하기 위한 함수입니다.