Design Project 1

Buffer To be Safe(BTS)

20200130 김유준, 20200180 김하린

Introduction

절체절명의 해누리호로 인해 임의로 손상이 가는 RAM에 대응하는 가상의 메모리 디자인이 필요하다. 큰 틀에서 우리의 디자인은 손상이 갈 메모리를 Disk swap 없이 미리 빠르게 복제할 수 있도록 Buffer를 관리한다. 이런 메모리 관리 Scheme을 Buffer to be safe(BTS)라고 부른다.
우리는 3번의 design iteration을 통해 3가지 버전의 BTS를 구현했다.

- Buffer To be Safe: Buffering at the Last using Double-sized buffer (BTS-BLD)
- Buffer To be Safe: Buffering at the Beginning using Double-sized buffer (BTS-BBD)
- BTS-BBD2

이름은 Buffer의 크기와 Buffer를 해주는 timing을 말해준다.

Buffer To be Safe(BTS)

---------- Iteration 1: BTS-BLD ----------

Observation 1

double-sized buffer before new cycle

- 흑점 폭발의 영향을 받게 될 프레임의 데이터를 교란되기 이전에 메모리의 안전한 곳으로 옮겨놓아야 한다. 또한, 교란되고 있는 곳의 메모리에 접근하려는 시도를 모두 차단해야한다.

Idea 1

- 30tick 이 끝나기 직전에 모든 process들의 가동을 멈추고 page table의 frame number를 필요시 안전한 frame number(다음 주기 때 흑점 폭발의 영향을 받지 않는 frame number)로 바꿔준다. 그리고 page의 frame number가 바뀐 경우에는, 바뀌기 이전의 frame으로부터 바뀐 이후의 frame으로 메모리를 복제한다.
- 이 때 swap-in/swap-out 빈도를 줄이기 위해 총 2000 frame 중 28개의 frame은 의도적으로 아무 process도 사용하지 않고 위에 언급한 memory 복제 및 교란중인 frame 관리에 사용하도록 한다.

Development 1

Assumptions

- 흑점 폭발에 의한 영향 사이의 주기를 sunspot cycle(혹은 cycle)이라고 한다. 교란되는 frame은 sunspot cycle이 시작할 때 바뀌게 된다.
- OS의 timer_ticks()기준으로 timer_ticks()%30가 만족되는 순간과 sunspot cycle이 시작하는 시점이 일치한다고 가정한다. 실제의 상황이라면 sunspot cycle의 시작이 랜덤한 N번째 tick이나 tick과 tick 사이에서도 일어날 수 있으나, booting과정에서 교란 프레임의 변화를 관찰함으로써 sunspot cycle의 시작과 우리의 operation cycle을 충분히 일치시킬 수 있다.
- i번째 tick(i=0,..., 30)은 이번 sunspot cycle에서 i번째 tick을 이야기한다. 엄밀히는 30 tick이 되는 순간에 다음 sunspot cycle이 시작되었다고 본다.
- refresh가 30 tick이 되기 전에 전부 끝난다고 가정한다.
- Booting 시 현재 교란되고 있는 프레임들의 bitmap bit(아래 Description 참고)은 1로 설정된다고 가정하자. 직접 구현했다면 좋았겠지만, 그러지는 않았다.
- disturbed_frame_indices_at(tick t) 을 부를 때 교란중인 메모리에 접근하지 않아 폭발이 일어나지 않는다고 가정한다. (이 가정은 모든 Iteration에 해당된다.)

Description

- 앞서 언급했듯이, 2000개의 frame 중 28개의 frame은 메모리 복제를 위해 비워졌거나 교란 프레임들을 관리하는 buffer frame으로 사용할 것이다. buffer frame들을 앞으로 '비워진 프레임' 혹은 'empty frame'이라는 용어로 부른다.
- 비워진 28개의 frame은 Kernel page에 EMPTY_F으로 저장한다. 이 frame list와 모든 프로세스들의 page table은 29 tick 에 timer_interrput()를 통해 refresh()가 불리며 업데이트 된다.
- refresh() 가 불리기 전 까지 EMPTY_F은 이번 sunspot cycle과 지난 sunspot cycle에서 frame_in_danger()인 frame을 모두 포함하고 있다. refresh()가 불린 이후에는 frame_in_danger()인 frame_will_be_in_danger()인 frame 들이 모두 EMPTY_F에 포함된다.
  refresh()는 현재 sunspot cycle이 끝나기 직전에 사용되고 있는 frame들 중 앞으로 교란될 위치에 있는 frame들을 EMPTY_F의 안전한 frame들 중 하나에 복제한다.

- timer_handler 에서 매 cycle의 29 ticks 마다 refresh()를 하고, bitmap의 bit data를 수정한다. bit data에 관해서서는 다음 문단의 설명을 참고한다.
- Physical frame들을 사용하지 않기 위해서는 palloc의 수정이 필요하다. Pool은 memory pool을 관리하기 위한 구조체인데, bitmap을 field로 가지고 있어 어떤 page가 allocate되어있는지를 알 수있다. Palloc에서는 kernel pool과 user pool을 구분해서 관리한다. 별다른 말 없이 bitmap이라고 사용된 용어는 user pool이 가지고 있는 bitmap을 일컫는다.
  refresh()전에 EMPTY_F 프레임 중 다음 cycle에서는 교란되지 않는 frame의 user_pool bitmap을 0(unallocated)으로 바꿔주고, refresh()이후 EMPTY_F 프레임 중 다음 cycle에서 교란되는 frame의 user_pool bitmap을 1(allocated)로 바꿔준다.
  자세한 과정정은 아래 figure, pseudo code, validation을 참고한다.

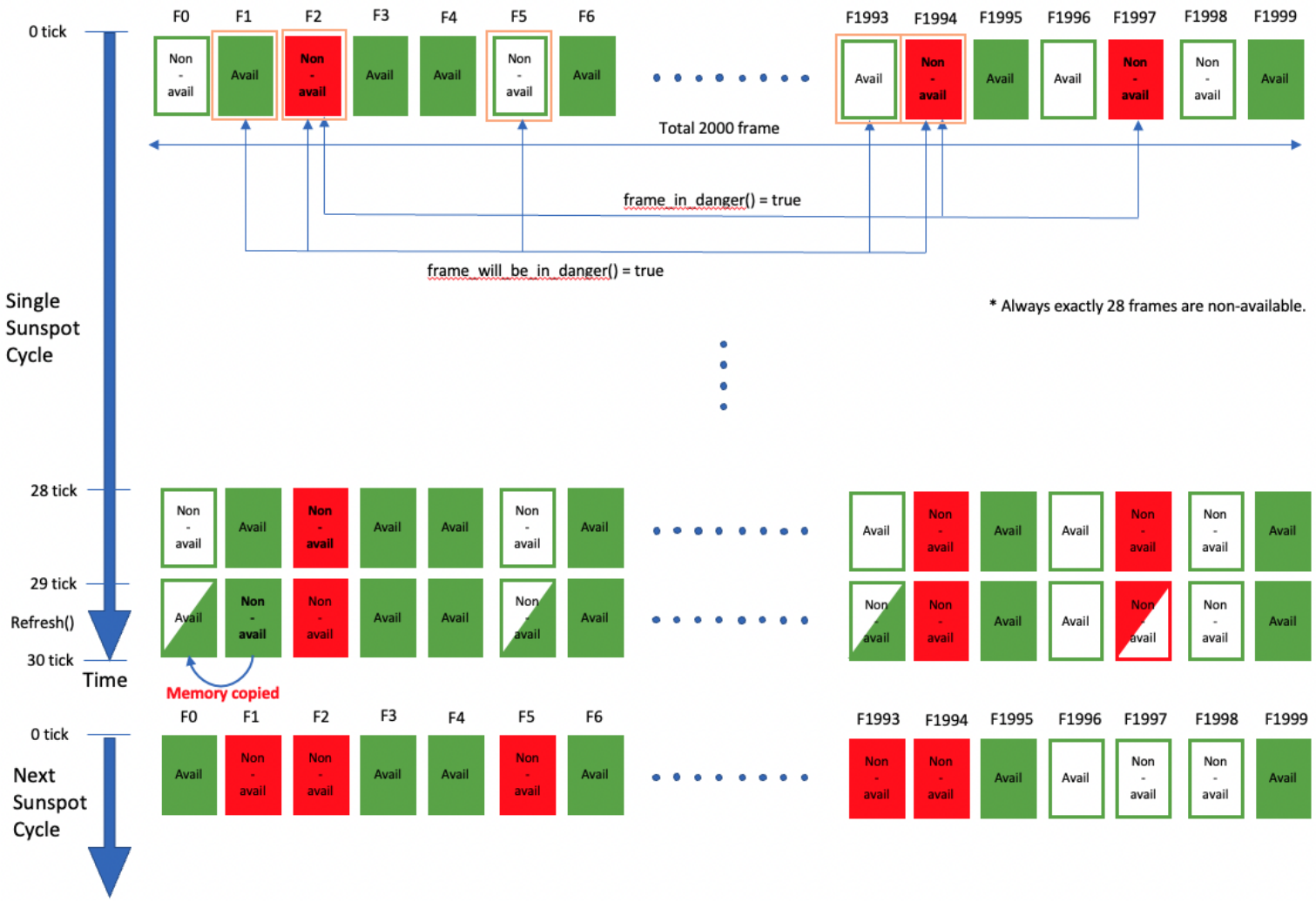


Figure1. Diagram for refresh() in Development1

- Available - 사용할 수 있는 프레임 (= EMPTY_F에 속하지 않는 프레임)
- Non-available - 사용할 수 없는 프레임 (= EMPTY_F에 속한 프레임)
- Filled Box - 사용되고 있는 프레임 (bitmap의 bit이 1. i.e. Allocated)
- Hollow Box - 사용하고 있지 않은 프레임 (bitmap의 bit이 0. i.e. Unallocated)

- Red Box - 흑점의 영향을 받는 프레임
- Green Box - 흑점의 영향을 받지 않는 프레임

위 그림은 현재 디자인에서 가능한 예시 상황을 나타내고 있다. 기존 OS와 달리 29 tick에서 refresh()와 bitmap bit 수정이 일어난다. 29 tick ~ 30 tick 사이에서 일어나는 각 프레임별 변화를 살펴보면 다음과 같다.

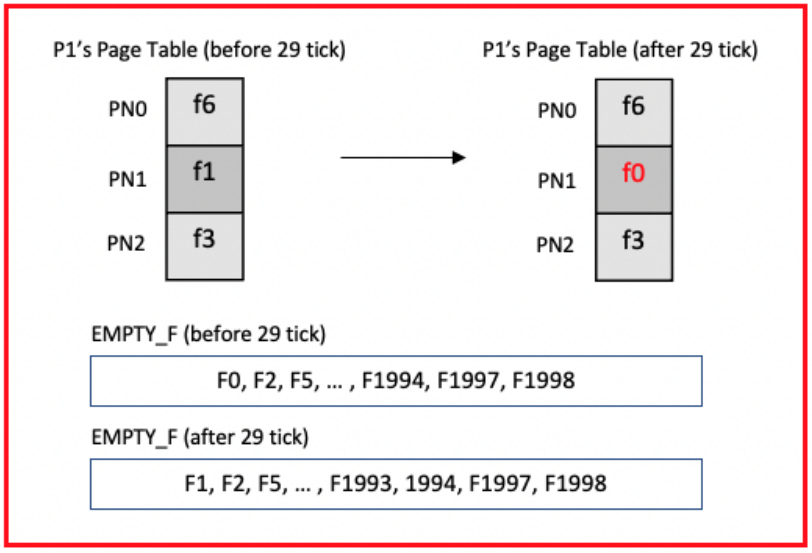
F1 frame : F1 프레임은 현재 사용중(bitmap bit=1)이며 흑점의 영향을 받지 않지만, 다음 주기에서는 교란될 프레임이다. 그러므로 bitmap의 bit은 1로 유지시킨다. refresh()가 불리면, F1은 다음주기에서 사용될 수 없기에 emtpy_safe_frame()을 통해 데이터를 옮길 새로운 안전한 프레임을 부여받는다. 이를 반영시키기 위해 page table에서 F1을 갖고 있던 pte의 값은 F0으로 수정되고, 새로운 프레임으로 모든 데이터가 복제된다. F1은 더 이상 사용할 수 없으므로 EMPTY_F에 추가된다.

F0 frame : F0 프레임은 EMPTY_F에 속한 사용할 수 없는 프레임 중 하나였다. 그러나, F1이 사용할 수 없게 되면서 이주하게 될 프레임으로 F0이 선택되었다. F0은 F1으로부터 데이터가 복제되고, bitmap의 bit이 1로 설정된다. 이제는 사용이 가능하고 사용중인 프레임으로 변하면서 EMPTY_F에서 F0이 제거된다.

F5 frame : F5 프레임은 EMPTY_F에 속한 사용할 수 없는 프레임 중 하나였다. 그러나, 다음 주기에 흑점의 영향을 받게 된다. 다음 주기에서 다른 프로세스가 사용할 수 없도록 만들기 위해 bitmap의 bit을 1로 설정한다.

F1993 frame : F1993 프레임은 사용할 수 있지만 사용하고 있지 않던 프레임이다. 그러나, 다음 주기부터는 사용할 수 없게 되기 때문에, bitmap의 bit이 1로 설정된다. 그리고 EMPTY_F 리스트 중 다음 주기에서도 안전할 프레임 중 하나를 제거하고 F1993을 추가한다.

F1997 frame : F1997 프레임은 흑점의 영향을 받고 있지만, 다음 주기에서는 흑점의 영향을 받지 않는다. EMPTY_F의 총 프레임 개수를 28개로 유지돼야 하기에 별 다른 조치를 취하지



않는다.

Figure2. Diagram for change on PT and EMPTY_F after 29th tick

Before 29th tick

P1이란 프로세스의 page table 에 있는 pte 중 하나가 F1을 갖고 있는 상황이다.
아래는 EMPTY_F 리스트에 속해 있는 프레임들이다.

After 29th tick

PN1의 프레임이 F1에서 F0으로 바뀌었다. EMPTY_F에 있던 F0이 제거되고 F1이 추가되었다.
F1993의 교란이 예정되면서 F1993이 EMPTY_F에 추가되었다. (그림에는 표현되어 있지 않지만, 추가될 때 다음 주기에 안전한 다른 프레임 하나를 제거하므로 프레임 수는 28개로 쭉 유지된다.)

[pseudocode]

```
int tick_current()          // 현재의 tick을 알 수 있다.
global struct frame * EMPTY_F[28]// 커널이 갖고 있는 28개의 empty frame
list

// paddr의 struct page 포인터를 리턴한다.
struct page * paddr_get_page(void * paddr)

struct frame {
    void *kva;
    struct page *page;
}

// 특정 시점(t)에 교란되는 프레임 리턴
```

```
int[] disturved_frame_indices_at(tick t)
// 특정 시점(t)에 교란되는 프레임 포인터 리스트 리턴
struct frame ** disturved_frame_at(tick t)

// F가 현재 주기에서 흑점의 영향을 받는지를 리턴
bool frame_in_danger(struct frame F)
    d = disturved_frame_at(tick_current()).includes(F)
    return d

 / F가 다음 주기에서 흑점의 영향을 받는지를 리턴
bool frame_will_be_in_danger(struct frame F)
    d = disturved_frame_at(tick_current()+30).includes(F)
    return d

// 커널이 저장하고 있는 28개의 empty 프레임 번호 중에 현재 흑점의 영향을 받지
않고, 다음 주기에서도 흑점의 영향을 받지 않는 프레임 번호를 하나 리턴한다.
struct frame * empty_safe_frame() {
    for F in EMPTY_F {
        if (!frame_in_danger(F) && !(frame_will_be_in_danger(F))
          return F
    }
    return NULL
}

// 다음 주기에서 흑점의 영향을 받는 프레임 번호의 경우 empty frame 번호 중에
하나를 선택해서 교체하고 메모리를 새로운 프레임으로 복제한다.
bool replace_with_empty_frame(uint64_t *pte, void *va, void *aux) {
      struct page * old_P = paddr_get_page(pte -> paddr)
      struct frame * old_F = old_P -> frame
      if (frame_will_be_in_danger(old_F)) {
            struct frame * new_F = empty_safe_frame()
            ASSERT(new_F != NULL)
            // EMPTY_F : new_F -> old_F
            EMPTY_F.remove(new_F)
            EMPTY_F.add(old_F)
            old_P -> frame = new_F
            // replace pte's paddr value in to new frame's paddr
            pte -> paddr = vtop(new_F -> kva)

            // set user pool bitmap for new_F to 1
            set_f_user_pool(new_F, true)

            // Copy data from old frame if it was allocated
            if get_f_user_pool(old_F) {
                memcpy(new_F -> kva, old_F ->kva)
            }
      }
```

```
	return true;
}


// thread가 갖고 있는 모든 pte에 대해서 replace_with_empty_frame()을
실행시킨다.
void refresh_mapping(thread T) {
	pml4_for_each(T->pml4, replace_with_empty_frame ,NULL)
}

// 모든 유저 프로세스에게 refresh_mapping()을 실행시킨다.
void refresh() {
	// block interruption while refreshing mapping
	old_intr_status = intr_disable()
	for all user_processes P:
		refresh_mapping(P)
	set_intr_status(old_intr_status)
}

// 수정된 timer interrupt handler
timer_interrupt (struct intr_frame *args UNUSED) {
    ticks++
    thread_tick()
    if (ticks%30==29)
        for F in EMPTY_F:
            if (!frame_will_be_in_danger(F)) {
                set_f_user_pool(F, false) // Set bitmap for F as
unallocated
            }
        refresh()
        for F in EMPTY_F:
            if (frame_will_be_in_danger(F)) {
                set_f_user_pool(F, true) // Set bitmap for F as
allocated(although it will not contain any meaningful data) so that no
user program can access
            }
        tlb_flush()     // flush TLB since PT mapping has been changed.
        timer.sleep(1ms)
    else
        thread_wakeup(ticks)
}

// 아래 함수들은 timer_interrupt에서 교란중인 frame이 allocate되지 않도록
하는데 사용됨

/* A memory pool. */
struct pool {
```

```
    struct lock lock;                   /* Mutual exclusion. */
    struct bitmap *used_map;            /* Bitmap of free pages. */
    uint8_t *base;                      /* Base of pool. */
}

/* Atomically sets the bit numbered IDX in B to VALUE. */
void bitmap_set (struct bitmap *b, size_t idx, bool value)

/* Returns the value of the bit numbered IDX in B. */
bool bitmap_test (const struct bitmap *b, size_t idx)


/* Set bitmap bit corresponding to frame F in user pool as according to
value*/
void set_f_user_pool (frame *f, bool value){
    int phys_addr = vtop(f -> kva)
    int bit_idx = (phys_addr - user_pool -> base) / PGSIZE
    bitmap_set(user_pool -> used_map, bit_idx, value)
}
/* Get bitmap bit corresponding to frame F in user pool */
bool get_f_user_pool (frame *f){
    int phys_addr = vtop(f -> kva)
    int bit_idx = (phys_addr - user_pool -> base) / PGSIZE
    return bitmap_test(user_pool -> used_map, bit_idx)
}
```

Validation 1

Can empty_safe_frame() return NULL?

- replace_with_empty_frame()에서 empty_safe_frame()은 frame_will_be_in_danger(F)를 만족해야 불린다. refresh_mapping()에서 loop로 도는 F 중 frame_will_be_in_danger(F)를 만족하는 F가 주어졌다고 하자.
- F는 EMPTY_F에 포함되지 않는다(EMPTY_F의 관리 scheme에 의해). 만약 empty_safe_frame()이 NULL을 return하려면 EMPTY_F이 모든 frame_in_danger()와 frame_will_be_in_danger()인 frame을(둘을 합쳐 총 28개) 포함하고 있어야 한다. F도 frame_will_be_in_dange(F) 를 만족하기 때문에 EMPTY_F이 F를 포함하고 있어 모순이다.

Can newly allocated user memory cause an explosion?

- 프로그램이 실행되는 중 palloc을 통해 새로운 memory를 allocate받을 수 있다. 이 때, user_pool에 있는 bitmap에서 연속하게 unallocated된 page를 찾아 allocate해준다. refresh()를 하고 나면 frame_will_be_in_danger()인 frame은 모두 EMPTY_F에 추가되고, 이 frame들이 palloc에게 선택받지 못하도록 bit를 1로 설정된다. bit를 1로 만들면, frame이 실제로는 어느 page에도 allocate되지 않았지만

palloc_get_page()등의 call을 할 때는 해당 frame이 이미 allocate 된 것으로 표시되어 선택하지 않는다. 따라서 해누리호의 폭발을 막을 수 있다.

- 다만 29tick에서 bit를 0으로 set하기 위한 !frame_will_be_in_danger()조건은 frame_in_danger()를 체크해주지 못하기 때문에 frame_in_danger()인 frame의 bit가 0으로 set 될 수 있다. 때문에 29-30 tick사이에 새로운 Allocation이 생기면 교란중인 frame이 allocate 될 수 있고 이로 인해 해누리호의 폭발 가능성이 생겼다. 이를 해결하고자, 29 tick에서 timer_interrupt가 끝나고 나서는 한 tick동안 sleep을 할 수 있도록 설정을 해 주었다. 다음 sunspot cycle에서는 bit를 0으로 바꾸어 주었던 frame들이 모두 안전하게 사용될 수 있다.

Does this algorithm cause a memory leak?

- 다음 clock cycle에 일어날 교란으로 인해 bitmap의 bit가 0에서 1로 바뀐 경우 교란이 끝날 때 bitmap이 다시 0으로 바뀐다. 코드 상으로 이렇게 1로 set된 bit는 다음 sunspot cycle의 refresh() 이전에 EMPTY_F의 frame중 frame_will_be_in_danger()가 아니라면 다시 0으로 설정된된다. 두 refresh() call 사이에서 EMPTY_F는 바뀌지 않으며, 특정 frame이 EMPTY_F에 추가된 직후와 제거되기 직전에 각각 frame_will_be_in_danger()와 !frame_will_be_in_danger()를 만족하여 bit가 1로 한번, 0으로 한번 설정된다. 때문에 Memory leak가 없음을 증명할 수 있다. 이는 figure 1에서 29 와tick의 F5와 F1997의 상황에 대응된다.
- 다만 Figure1의 F0 및 F1같은 상황처럼 이미 Allocated되어 있는 memory가 교란이 예정됐다면 메모리를 복제해 주어야한다. 이 때는 F0을 allocate시켜주고 이곳에 F1의 메모리를 복제한다. F1은 EMPTY_F에 추가된다. User program은 바뀐 PTE를 통해 나중에 F1대신 F0을 free하게 되고, F1은 더이상 교란되지 않을 때 EMPTY_F에서 bitmap이 0으로 수정된다. 즉 memory leak가 일어나지 않는다.

How is EMPTY_F managed?

- EMPTY_F는 non-available한 frame의 list로 OS가 booting될 때 현재 교란되는 프레임을 포함한 28개의 임의의 frame으로 initialize되고, replace_with_empty_frame()을 통해서만 수정한다. replace_with_empty_frame() 에서는 항상 remove와 동시에 add가 발생하기 때문에 프레임은 28개로 늘 유지된다.

속도저하 요인 분석

- Frame 28개를 사용하지 EMPTY_F를 위해 사용해서 Swap-in/out 빈도가 약간 증가한다.
- refresh()를 매 sunspot cycle마다 해줘야 한다.
- 매 sunspot-cycle마다 한 tick 시간만큼을 sleep해야한다.

Synchronization

- refresh()를 intr_diable()을 통해 한번에 실행 될 수 있게 해주었다.

디자인 비교

- Timing: Sunspot cycle시작할 때 vs 끝날 때 / EMPTY_F를를 어떻게 수정할까?

refresh()를 부르는 시점에서 어떤 frame들을 EMPTY_F에서 넣고 뺄지, 그리고 그에 따라 어떤 frame들의 data를 복사할지 고려했다. 현재 cycle에서 교란되고 있는 frame을 sunspot cycle의 시작(ticks%30 == 0)에서 EMPTY_F에 넣어주는 방식과 다음 cycle에서 교란되고 있는 frame을 sunspot cycle의 끝(ticks%30 == 29)에서 수정하는 방식을 비교했다. 전자의 방식에 따르면 Sunspot cycle의 시작에서 refresh()기 불리면서 EMPTY_F과 해당되는 frame의 data를 접근하게 되면 이미 교란된 frame에 접근하여 해누리호가 폭발 할 수 있다. 때문에 다음 sunspot cycle에서 교란될 frame들을 미리 예측하는 후자의 방식이 채택되었다.

- 왜 최소 28개의 frame을 비워야 하는가?

우선 매 순간 14개의 frame이 교란되는 상태라고 가정하자. 교란되는 frame은 사용할 수 없기 때문에 14개의 frame을 항상 비워두어야 한다. 이 14개를 제외한 다른 frame들을 모두 사용하고 있는 상태에서, 다음 sunspot cycle에서 교란될 또 다른 14개 frame의 데이터를 옮기려면, 메모리에 빈 공간이 없어 swap-in/out이 발생한다. 교란될 14개 frame 모두의 swap-in/out이 발생하면 sunspot cycle당 28개의 I/O call이 발생한다. synchronization을 위해 refresh()를 실행 할 때 intr_diable()을 해주었기 때문에 sunspot cycle당 28 tick 동안 아무것도 아무것도 할 수 없게 된다. 즉, CPU resource를 굉장히 많이 낭비하게 된다. 이러한 이유로 비워둘 frame의 개수를 28개로 두어 교란에 인한 swap-in/out 가능성을 제거했다.

Observation, again

- 기존 디자인은 한 주기 내에서 29번째 tick 때 refresh()를 한다. 하지만 refresh()가 가정과 달리 한 tick 안에 끝나지 않으면 데이터 손실 뿐만 아니라 교란중인 frame을 건드려 폭발 위험이 존재한다. 이 위험을 제거하기 위해서 refresh() 시기를 앞당긴다. 0번째 tick에서 다음 sunspot cycle에서 교란 될 frame들을 이용해 refresh()를 해주면 [Iteration1-validation- 디자인 비교 - timing]에서 언급한 위험과 refresh가 한 tick안에 끝나지 않아 생길 수 있는 위험을 모두 제거할 수 있다.
- Palloc을 할 때 교란되지 않는 frame만을 주기 위해서 timer.sleep(1ms)를 해 주었는데, 이로 인한 CPU waste를 해결하고자 한다.
- 28 - 29 tick 사이에 disk read/write를 하게 된다면, refresh()로 pte가 바뀐 것을 제대로 반영하지 못할 수 있는 문제가 있어 이를 해결해야 한다. 28 tick부터 모든 read/write call을 멈추는 것도 하나의 해결책이다.

---------- Iteration 2 : BTS-BBD ----------

Idea 2

- refresh() timing : 29번째 tick -> 0번째 tick 으로 변경한다.
- 대신 refresh() 전후로 bitmap을 수정할 frame들이 바뀐다.
- non-available history를 통해 한 tick sleep하는 것을 막을 수 있다.
    - set_f_user_pool()를 통해 bitmap을 수정하는 작업 역시 allocate/ free라고 부른다.
    - will_be_in_danger()인 frame의 bitmap bit을 직접 설정하여 allocation을 관리해 주는 구현을 고민했다. 이렇게 하기 위해서는 지난 cycle에서 in_danger() 였으면서 이번 cycle기준으로 !will_be_in_danger()인 frame을 free해주고, will_be_in_danger()인 frame을 allocate 해준다. 그러려면 지난 cycle에서 in_danger() 였던 frame list를 history로 가지고 있어야 한다.
    - 반면에 아래 구현은 is_in_danger()를 사용해 history를 가지고 있지 않아도 잘 작동하도록 구현했다.

Development 2

Assumptions

- Iteration-1과 동일하게 sunspot cycle이 매번 tick%30==0이 되는 순간 영향을 주며, 영향을 받은 직후 timer interrupt가 발생하여 spot cycle의 시작과 timer interrupt 사이에 user program이 실행되지 않는다고 가정한다.
- Booting시 OS에서 현재 교란중인 프레임과 다음 주기에서 교란될 프렘임의 bitmap의 bit을 1로 설정했다고 가정한다.

Description

- timer_interrupt에서는 기존의 refresh()함수를 그대로 사용하고 refresh의 타이밍과 bitmap의 bit 설정 대상이 바뀐다. 기존에는 frame_will_be_in_danger()인 프레임들의 bit이 1로 설정되었다면, 바뀐 디자인에서는 frame_in_danger()인 프레임들의 bit을 1로 설정한다.
- Refresh 및 bit 수정 작업을 해줄 frame을 그대로 두고 timing만 0 tick으로 바꾸게 된다면, 기존에 frame_in_danger()인 frame이 allocate될 수 있어 timer.sleep(1ms)을 해준 부분을 timer.sleep(30ms)로 바꾸어야 한다. 이렇게 되면 OS가 제대로 작동할 수 없다. 대신, 이번 sunpot cycle에서 교란되는 frame들의 bit를 수정하여 새롭게 allocate되지 않도록 하고, 지난 sunspot cycle에서 교란되었지만 이번 sunspot cycle에서는 안전한 frame들을 unallocate해준다.
- refresh()가 일어나기 1 tick 전 부터 disk read/write를 하는 것이 어떤 문제를 야기하는지 생각해 보았다. Read/write를 하는데 한 tick이 소요되기 때문에 중간에 refresh()를 하면서 physical memory address가 바뀔 수 있다. 하지만 refresh를 하기 전에 읽은 data는 refresh()내에서 다른 프레임으로 복제되고, refresh 이후에 읽는

data는 PTE가 바뀌었기 때문에 알맞은 주소로 마저 읽어 알고리즘의 별다른 수정이 없이 정상적으로 코드가 작동한다.

Figure 3. Diagram for implementation in Development 2

Figure3은 현재 디자인에서 가능한 예시 상황을 나타내고 있다. 한 가지 유의할 점은 next sunspot cycle을 나타난 행은 refresh()와 bit modification 이전의 상황을 담고 있다. Iteration-1과 달리 0 tick에서 refresh()와 bitmap bit 수정이 일어난다. 0 tick ~ 1 tick 사이에서 0 tick의 timer interrupt로 인해 일어나는 각 프레임별 변화를 살펴보면 다음과 같다.

F0 frame : F0 프레임은 지난 cycle에서는 EMPTY_F에 속한 사용할 수 없는 프레임 중 하나였다. 현재 cycle에 들어서면서는 교란되고 있다. 교란되게 되면서, bitmap의 bit이 1로 설정된다.

F2 frame : F2 프레임은 지난 주기에서 교란되었으나, 이번 주기에 들어서는 교란되지 않는다. 그러나, 28개의 프레임을 비워두기로 했으므로, 별 다른 조치를 취하지 않는다.

F3, F1996 frame : 지난 주기에서 F3 fram은 EMPTY_F에 속해 있던 사용할 수 없는 프레임이었고, F1996 프레임은 사용가능한 프레임이었다. 하지만, 현재 주기에 들어서면서

F1996 프레임이 다음 주기에서 교란될 프레임으로 정해지면서, EMPTY_F에 속한 프레임 중 다음 주기에서도 안전한 프레임인 F3 프레임을 제거하고 F1996 프레임을 새롭게 추가하였다. 이제 F3 프레임은 사용 가능한 프레임이고, F1996 프레임은 EMPTY_F에 속한 사용 불가능한 프레임이 된다.

F5 frame : F5 프레임은 이전 주기에서는 EMPTY_F에 속한 사용 불가능한 프레임이었고, 이번 주기와 다음 주기 모두 교란될 프레임이다. 사용할 수 없으므로 bitmap의 bit이 1로 설정된다.

F6 frame : F6 프레임은 사용할 수 있었고, 사용 중이었던 프레임이다. 그러나, 다음 주기에 교란될 것이 확정되면서 데이터를 옮기게 되었다. 데이터를 옮길 프레임으로 EMPTY_F에 속한 프레임들 중 현재와 다음주기에 안전한 프레임 중 하나인 F1994 프레임이 선택되었고, 메모리가 복제되었다. F6 프레임은 EMPTY_F에 추가되어 더 이상 사용할 수 없게 된다.

F1994 frame : F1994 프레임은 지난주기까지는 교란되었지만, 이번 주기와 다음 주기 모두 안전하여 F6 프레임의 이주 프레임으로 결정되었다. F6의 메모리가 복제되고, EMPTY_F에서 삭제된다.

F1997 frame : F1997 프레임은 지난 주기와 이번 주기 모두 교란되고 있다. 따라서 건드리지 않는다.

F1998 frame : F1998 프레임은 이번주기에 와서 교란되고 있다. 더 이상 사용할 수 없으므로 bitmap의 bit을 1로 설정한다.

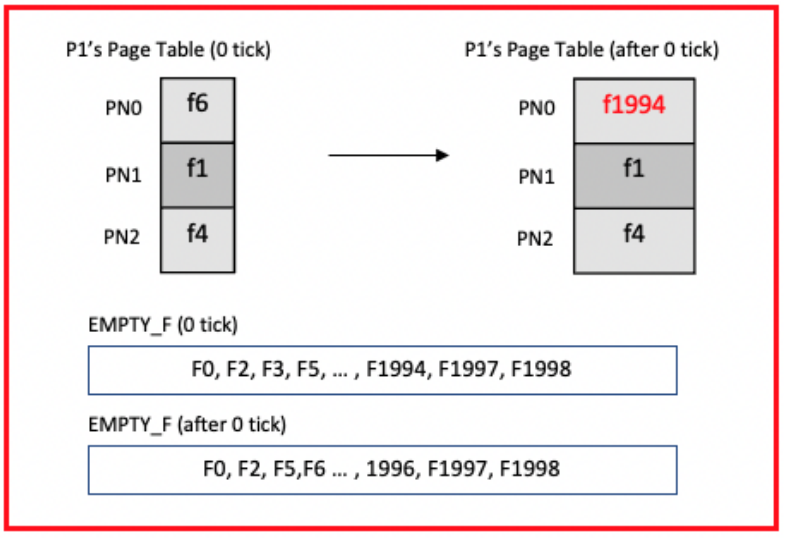


Figure 4. Diagram for change on PT and EMPTY_F after 0th tick

Start of the 0th tick

P1이란 프로세스의 page table 에 있는 pte 중 하나가 F6을 갖고 있는 상황이다.
아래는 EMPTY_F 리스트에 속해 있는 프레임들이다.

After 0th tick

PN1의 프레임이 F6에서 F1994로 바뀐다. 이와 함께 EMPTY_F에 있던 F1994가 제거되고 F6이 추가되었다. F1996의 교란이 예정되면서 F1996이 EMPTY_F에 추가되었고, 그 대신에 안전한 F3이 제거되었다.

[Pseudocode]

```
timer_interrupt (struct intr_frame *args UNUSED) {
    ticks++
    thread_tick()
    if (ticks%30==0) {
        for F in EMPTY_F:
            if (!frame_in_danger(F)) {
                set_f_user_pool(F, false) // Set bitmap for F as
unallocated
            }
        refresh()
        for F in EMPTY_F:
            if (frame_in_danger(F)) {
                set_f_user_pool(F, true) // Set bitmap for F as
allocated(although it will not contain any meaningful data) so that no
user program can access
            }
        TLB.flush() }
    thread_wakeup(ticks)
}
```

Validation 2

폭발 가능성이 없음

- frame_in_danger()인 frame은 user program에게 새로 allocate되지 않고, user program에게 allocate되었었다면 지난 cycle의 refresh()에서 이미 다른 메모리로 data가 옮겨 갔을 것이다. frame_in_danger()인 frame에 대해 set_f_user_pool()을 실행하는 것은 폭발과 연관이 없다.

메모리 Leak가 없음

- refresh() 전후로 대칭적인 구조와 두 연속한 cycle의 refresh() 사이에 EMPTY_F가 변하지 않는다는 점에서 refresh()후에 set_f_user_pool()로 allocate된 frame은 더이상 danger가 아니게 되었을 때 refresh()전에 set_f_user_pool()로 unallocate된다. refresh() 안에서 memcpy가 일어날 때 green non-avilable로 정보가

복사되는데, green non-available은 항상 hallow하다. 그러면 원래 정보가 있던 green available filled는 green non-available hallow가 되고 새로 정보가 옮겨간 green non-available hallow는 green available filled가 되어 filled인 frame의 수는 변하지 않고, 이 filled는 나중에 user program에서 free() call이 불리면 unallocate 된다.

성능이 향상됨

- CPU 관점
  한 tick동안 CPU가 sleep하는 코드가 사라졌기 때문에 performance가 향상되었다.
- Memory관점
  결국 매 시점 2000개 중 최대 28개의 frame만 추가적으로 allocate된 것처럼 bit가 수정되기 때문에 page fault로 인해 swap-in/out을 해야돼서 추가적으로 걸리는 시간이 유의미하게 늘어나지 않는다.

Observation, again

- 아뿔싸, EMPTY_F에 있는 안전한 non-avail 프레임들이 다른 유저 프로세스에 의해 사용되지 못하도록 막는 부분이 없다는 것을 깨달았다! 교란되는 frame들은 refresh 전후로 bitmap을 수정해 주지만, EMPTY_F안에 있는 frame들 중 교란이 일어나고 있지 않은 frame들도 user process에게 allocate되지 않도록 함께 관리를 해주어야 한다.

---------- Iteration 3: BTS-BBD2 ----------

Idea 3

- 한 가지 중요한 원칙이 지켜지지 않고 있었다. EMPTY_F에 속해있는 frame들은 전부 user program이 palloc_get_page()를 부를 때 allocate되지 않아야 한다. 하지만 우리는 교란중인 frame들만 user program에 allocate되지 않도록 관리를 하고 있었다.
- EMPTY_F에 add를 할 때 해당 frame의 user_pool의 bitmap의 bit를 1로 수정해주고 remove할 때 bit를 0으로 만들어준다면 EMPTY_F의 frame들은 user program에 의해 allocate되지 않는다. add,remove 대신에 add_empty_f_and_bit_modify(), remove_empty_f_and_bit_modify()를 사용해서 EMPTY_F 프레임들의 bitmap을 수정하자.
- OS를 Booting하여 EMPTY_F를 만들때도 add_empty_f_and_bit_modify()를 사용한다.

Development and Validation 3

bit 관리

- add_empty_f_and_bit_modify(), remove_empty_f_and_bit_modify()는 각각 EMPTY_F를에 add, remove를 함과 동시에 해당 frame의 bit을 1, 0으로 만들어준다. 따라서 EMPTY_F를 이 함수들 만을 이용해서 수정한다면 user process가 EMPTY_F안의 frame을 allocate할 수 없고, memory leak 또한 없다.
- EMPTY_F는 교란중인 frame을 전부 포함하고 있기 때문에 이 scheme을 사용하게 되면 refresh() 전후로 is_in_danger()여부로 bitmap을 수정하는 작업을 따로 할 필요가 없어진다.

[Pseudocode]

```
* Iteration-2와 비교했을 때 replace_with_empty_frame()와
timer_interrupt()가 수정되었다.

void add_empty_f_and_bit_modify(frame F) {
    EMPTY_F.add(F)
    set_f_user_pool(F, true)
}

void remove_empty_f_and_bit_modify(frame F) {
    EMPTY_F.remove(F)
    set_f_user_pool(F, false)
}

bool replace_with_empty_frame(uint64_t *pte, void *va, void *aux) {
      struct page * old_P = paddr_get_page(pte -> paddr)
      struct frame * old_F = old_P -> frame
      if (frame_will_be_in_danger(old_F)) {
            struct frame * new_F = empty_safe_frame()
            ASSERT(new_F != NULL)
            // EMPTY_F : new_F -> old_F
            bool was_allocated = get_f_user_pool(old_F)
            remove_empty_f_and_bit_modify(new_F)
            add_empty_f_and_bit_modify(old_F)
            old_P -> frame = new_F
            // replace pte's paddr value in to new frame's paddr
            pte -> paddr = vtop(new_F -> kva)

            if (was_allocated) {
                set_f_user_pool(new_F, true)
                memcpy(new_F -> kva, old_F ->kva)
            }
      }
      return true;
}
```

```
// 수정된 timer interrupt handler
timer_interrupt (struct intr_frame *args UNUSED) {
    ticks++
    thread_tick()
    if (ticks%30==0)
        refresh()
        TLB.flush() }
    thread_wakeup(ticks)
}
```

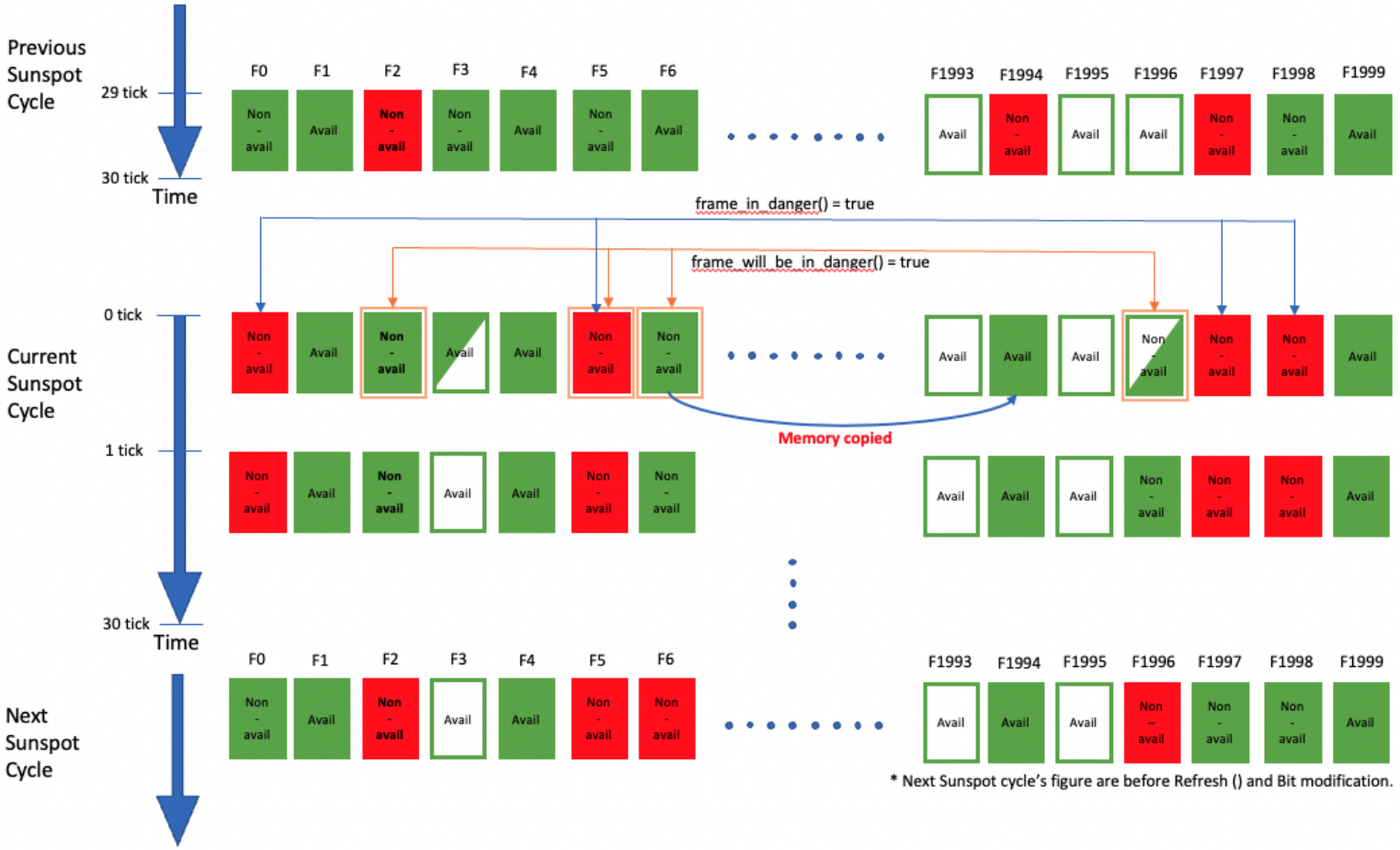


Figure 5. Diagram for implementation in Development 3

위 그림은 Iteration-2와 똑같은 상황이지만 바뀐 bitmap bit manipulation을 반영하고 있다. 기존 교란되지 않고 있던 Non-available 프레임들은 bitmap bit=0(hollow block)으로 설정되어 있었지만, 현재는 모두 1로 설정되어 있다(filled block).

Frame 3 : 이전 주기에서는 Non-available이었으나, F1996이 EMPTY_F로 추가되면서 EMPTY_F에서부터 제거되는 프레임이다. 제거되면서 다시 bitmap의 bit이 0으로 변경된다.

나머지 설명은 Iteration-2와 같다. EMPTY_F와 PT의 변화도 Figure 4를 참조하면 된다.

Conclusion

여러 Design iteration을 거치면서 기존 디자인의 문제점을 개선해 나갔다. Iteration2에서는 기존에 매 cycle마다 한 tick만큼 쉬어야 했던 부분을 해결했고, Iteration3에서는 EMPTY_F가 allocate되지 않도록 해야하는 문제를 완전히 해결했다. 각 step의 validation을 통해 폭발이 일어나지 않음과 memory leak/loss가 없음을 입증했다. 마지막으로 교란되는 frame에 의한 disk swap-in/out을 막기 위해서는 최소 28개의 frame을 비워야 한다는 점을 증명함으로 memory관리적인 측면에서 Design이 optimal함을 증명했다.

다만, timer tick이 30의 배수가 되는 순간과 sunspot cycle이 시기적으로 일치한다는 가정은, OS booting을 할 때 교란되는 frame을 관찰하여 sunspot cycle이 일어난 직후에 tick이 0부터 시작하는 design을 추가해 주더라도 완전히 해결되지 않는다. 이 Design을 추가하면 매 cycle의 29 tick과 30 tick 사이에 흑점폭발에 인한 교란이 생기게 되는데, 29 tick에서 30 tick 사이에 아무것도 하지 않도록 design을 해주면 흑점 폭발이 30 tick(혹은 다음 cycle의 0 tick)에서 일어난 것과 동일한 결과를 가져오기 때문에 우리의 Design이 잘 작동하게 된다.