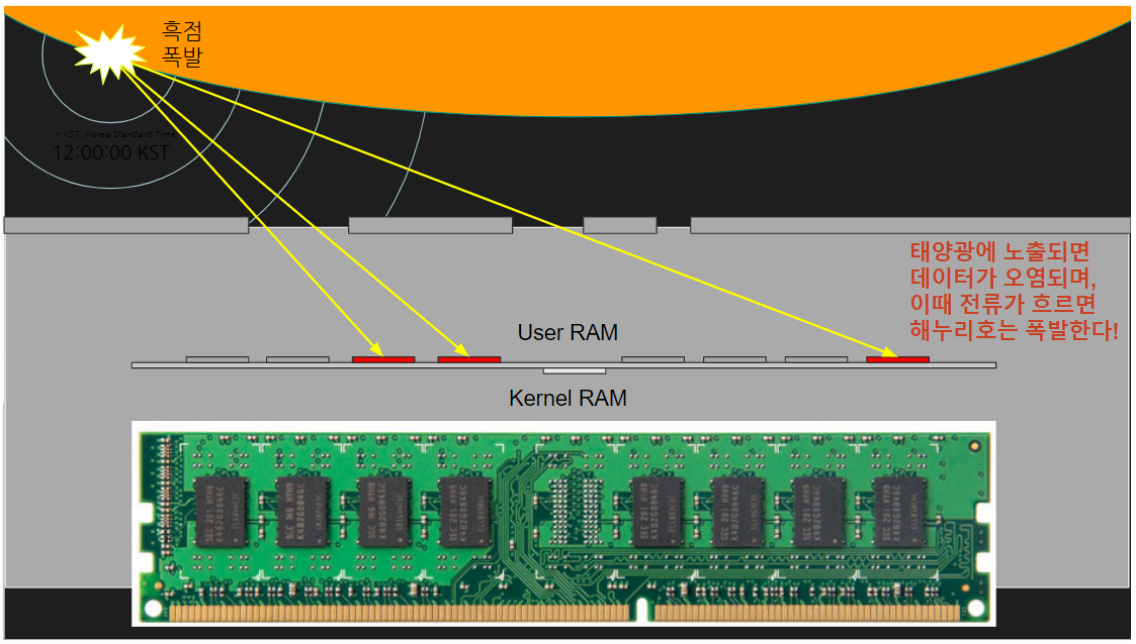
탐사선 해누리호 긴급 패치 방안

20180093 김보연
20180789 이해준

Overview

탐사선 해누리호와 우주 쓰레기 충돌로 인해 해누리호의 제어 컴퓨터 영역에 구멍이 발생했다. 해당 구멍으로 인해 흑점 폭발이 발생할 시 커널 영역을 제외한 RAM의 일정 부분에 태양풍이 도달하게 되었다. 태양풍이 도달한 메모리 영역은 교란이 발생한다. 흑점 폭발의 영향을 받는 RAM의 영역은 랜덤하게 변화하며, 해당 RAM 영역의 데이터를 로드할 경우 탐사선이 폭발할 수 있다. 또한, 태양풍의 영향을 벗어난 이후에도 해당 RAM영역의 데이터는 오염된 Junk값으로 변하여 사용할 수 없다.



[Fig 1] 태양풍에 노출된 해누리호 RAM 모식도

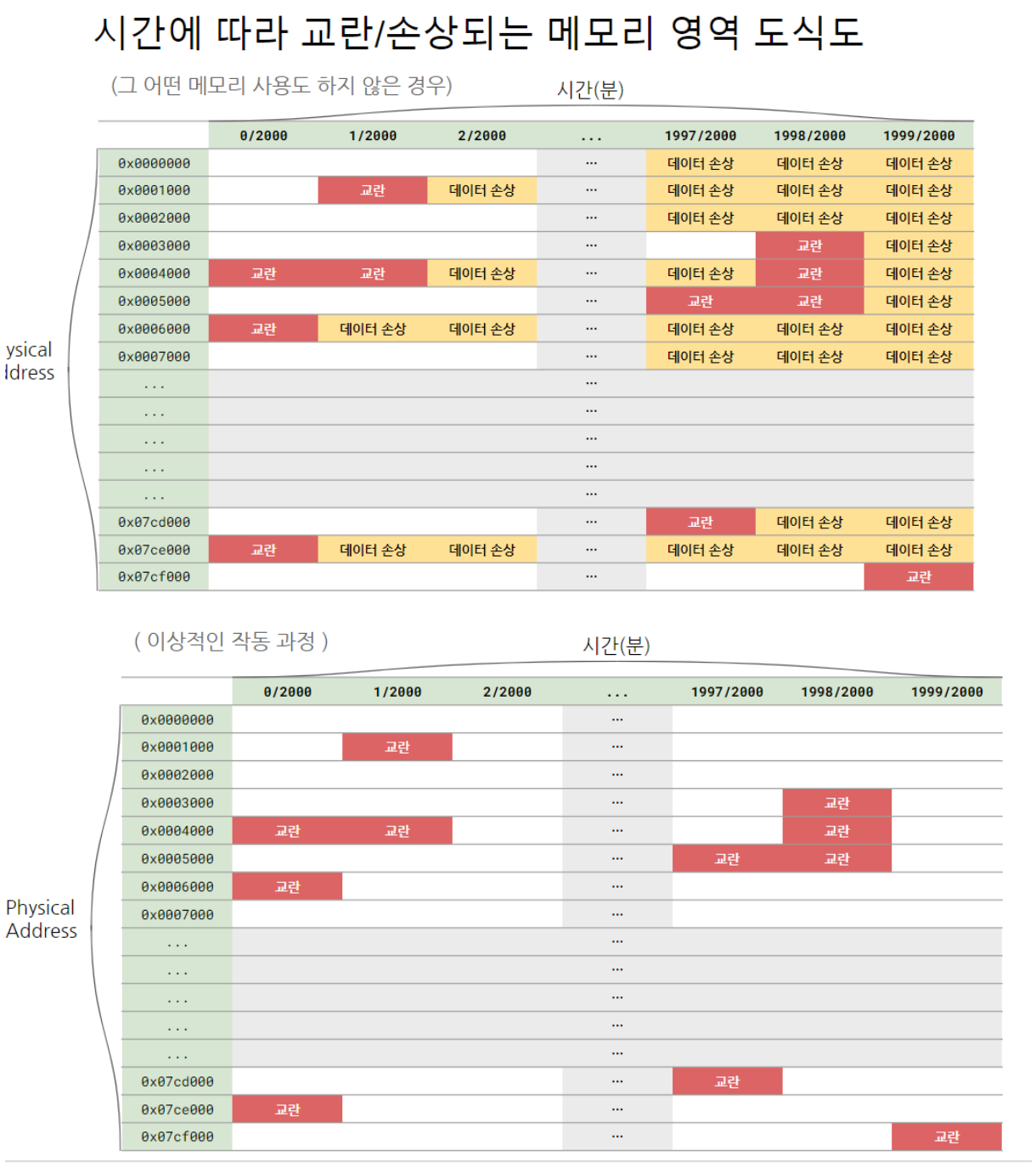
이에 해누리호의 성공적인 태양 탐사를 통해 해누리호 OS에 긴급 패치를 적용하고자 한다. 현재 상황으로 인해 해누리호에 발생한 문제 상황은 다음과 같다.

1. 태양풍 영향을 받는 RAM 영역에 접근시 해누리호가 폭발할 수 있다.
2. 어떤 RAM영역이 태양풍의 영향을 받을 경우 데이터 손실이 발생할 수 있다.
3. 해누리호에 성능 저하가 발생할 수 있다.

이 문제를 해결하기 위해 적용할 패치 OS는 해누리호의 폭발을 최우선적으로 막고, 교란된 데이터로 인한 유저 프로그램의 오동작을 피하며, 가급적 성능 저하를 최소화하여야 한다. 따라서 이 OS의 디자인 목표 우선순위는 다음과 같다.

1. 태양풍 영향을 받는 **RAM**영역에 유저 프로그램 접근을 방지한다**.**
2. 데이터 손상으로 인한 애플리케이션의 오동작을 피한다**.**
3. **OS** 패치로 인한 성능 저하를 최소화한다**.**

흑점 폭발로 인한 RAM영역의 교란은 30tick(1 Cycle) 주기로 변화하는 것으로 나타났다. 특정 시각에 태양풍에 노출되는 RAM 영역은 최대 14 frame이며, 이때 이 부분의 메모리에 접근하게 되면 폭발이 일어난다. 접근을 비롯해 메모리 사용을 하지 않더라도 적절한 조치를 취하지 않으면 결국 아래 그림처럼 저장하고 있는 모든 데이터를 손실하게 된다.



[Fig 2] 시간에 따라 교란/ 손상되는 메모리 영역 모식도
(위 : 그 어떤 메모리 사용도 하지 않은 경우 / 아래 : 구현하고자 하는 가장 이상적인 시나리오)

태양풍 예측 알고리즘을 통해 어떤 시점으로부터 30tick 내에 태양풍의 영향을 받을 RAM 영역을 예측할 수 있게 되었다. 해당 알고리즘이 구현된 함수는 다음과 같다.

```
int[]
disturbed_frame_indices_at(tick t);     /* Return index list of frames to be disturbed next cycle */
```

해당 함수는 다음 태양풍 도달 시에 교란받을 프레임의 index 리스트를 리턴한다.

우리는 이 함수를 이용하여 다음번에 교란될 프레임을 예측하여 선제적으로 대응할 것이다. 위의 디자인 목표를 달성하기 위해 다음과 같은 세부적인 Task를 설정하였다.

1. 30ticks동안 교란 프레임 영역에 유저 프로그램의 접근을 차단한다.
2. 교란될 데이터를 사전에 이전시켜 보호한다.
3. 위의 두 목표를 모두 달성하며 성능 저하를 최소화한다.

다음과 같은 우선순위의 평가 기준을 통해 해누리호 패치 OS를 평가하고자 한다.

1. 태양풍으로 인한 해누리호의 폭발을 완벽하게 방지하였는가?
2. 태양풍으로 인한 데이터 손실을 방지했는가?
3. 패치 OS의 성능 저하를 최소화하였는가?
    a. 해누리호의 안정성 보장을 위해 많은 리소스가 소요되지는 않는가
    b. 성능 저하가 일어나는 특정한 상황이 존재하는가
    c. 유저 프로그램의 동작에 많은 제한이 걸리지 않는가

해누리호의 기존 OS는 KAIST-Pintos와 유사하게 구성되어 각 프로세스에는 하나의 스레드만 존재한다. 뿐만 아니라 하나의 프레임은 하나의 스레드의 virtual address로 참조된다.

기존 OS에는 다음과 같은 구조체가 있다.

struct page

```
/* The representation of "page". */
struct page {
  const struct page_operations *operations;
  void *va;             /* Address in terms of user space */
  struct frame *frame;
```

struct page는 frame과 va에 대한 정보를 가지고 있다.

struct frame

```
/* The representation of "frame" */
struct frame {
  void *kva;
  struct page *page;
  struct list_elem elem;
}
```

struct frame은 page와 kva에 대한 정보를 가지고 있다.

Round 1 : Cohort Protection

Observation

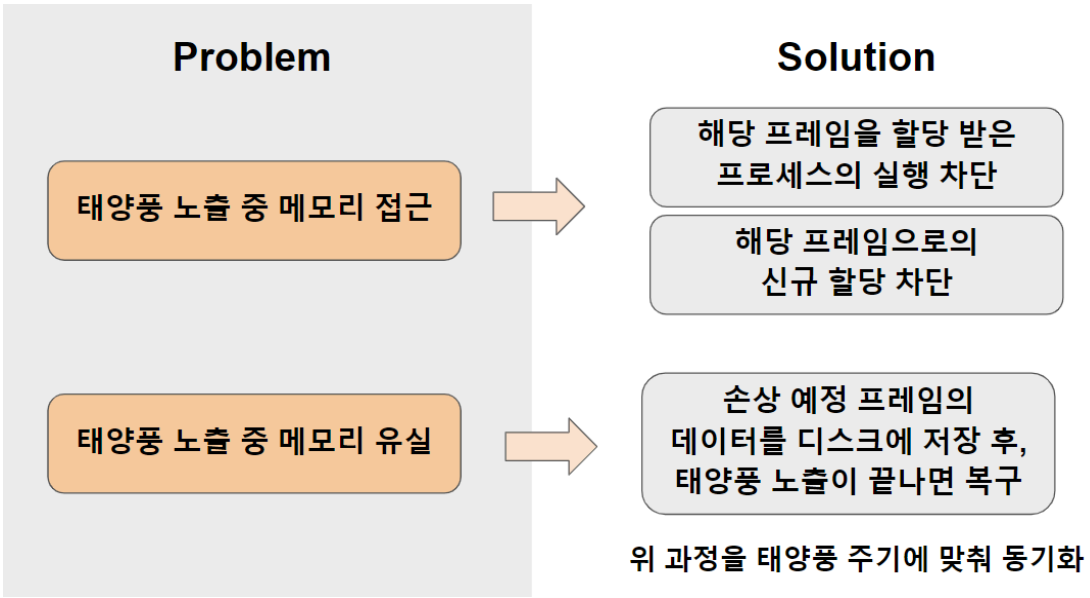
탐사선 해누리호의 RAM영역에 태양풍이 도달하여 교란이 발생하였다. 해당 교란으로 인해 발생할 수 있는 문제는 다음과 같다.

1. 태양풍 노출 중 메모리 접근 문제
   태양풍의 영향을 받는 메모리 영역을 점유하는 유저 프로그램이 해당 메모리 영역에 접근할 경우 해누리호가 폭발할 수 있다.
2. 태양풍으로 인한 메모리 유실 문제
   태양풍으로 교란된 메모리 영역의 데이터는 Junk 데이터가 되어 사용할 수 없게 되었다.

이번 디자인에서는 우선순위가 높은 위의 치명적인 문제를 최우선적으로 해결하여 해누리호의 안전과 유저 프로그램의 안정성을 보장하고자 한다.

Idea

최우선적으로 해결해야 하는 과제는 태양풍 교란중인 영역에 접근하는 것을 차단하는 것이다. 따라서 해당 영역을 사용하는 유저 스레드의 실행을 아예 차단하고, 해당 영역이 신규 할당되는것을 방지하여 폭발을 방지할 것이다. 또, 오염될 메모리 영역을 미리 디스크에 저장 후 복구하여 데이터의 손실을 막고자 한다. 따라서 이번 디자인에서 하고자 하는 Task를 다음과 같이 정리할 수 있다.



1. 교란 영역을 사용하는 유저 스레드의 실행을 교란 사이클동안 차단한다**.**
교란 영역의 데이터에 대한 접근을 방지하는 가장 간단한 방법은 해당 영역을 사용하는 스레드의 실행을 차단하는 것이다. 따라서 우리는 교란될 프레임을 사용하는 스레드를 추적하여 해당 스레드가 스케줄링되지 못하도록 하는 방법을 통해 데이터 접근을 방지할 것이다. 이를 통해 사용중인 교란될 예정인 프레임에 대한 접근을 방지할 수 있다.

2. 손상 예정인 프레임에 대해 다음 **cycle**동안 신규 할당을 금지한다**.**
비어있는 프레임이 손상 예정일 경우, 이 영역에 대해 태양풍 노출 기간동안 메모리 할당을 시도할 수 있다. 따라서 해당 기간동안 이 프레임에 대한 메모리 할당 시도를 차단한다.

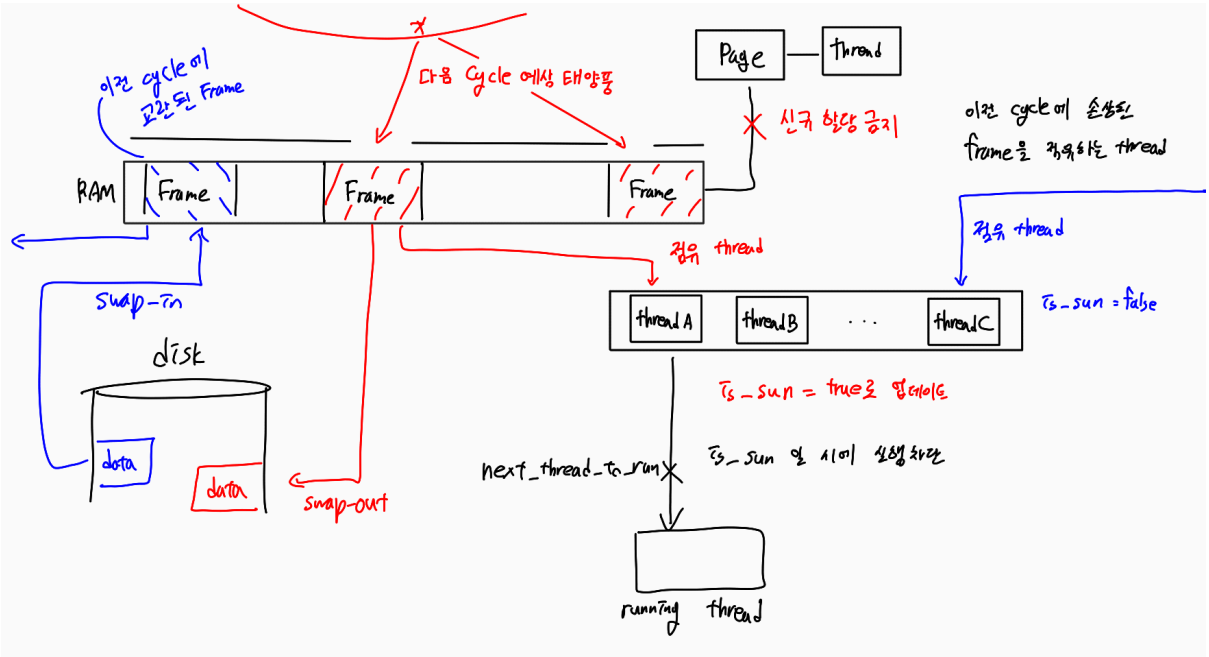
위의 두 Task를 통해 해누리호의 폭발을 방지할 수 있다.

3. 손상 예정인 데이터를 디스크에 저장 후 복구한다**.**
태양풍으로 인해 데이터가 오염되기 이전에 해당 프레임의 데이터를 disk로 swap-out하여 보존한다. 해당 프레임이 영향을 벗어나면 해당 데이터를 다시 swap-in하여 복구시킨다. 다만, 최적화를 위해 프레임이 사용중이 아닌 경우에는 데이터를 보존하지 않는다. disk접근에 소요되는 시간은 1tick이지만, 한 번에 교란되는 프레임 영역은 최대 14개이므로, 최악의 상황에서도 30tick 중 28tick이 디스크 접근에 소요되어 해당 작업을 통해 데이터의 손실을 효과적으로 막을 수 있다.

4. 전체 작업을 태양풍 주기와 동기화한다**.**
태양풍 예측 알고리즘을 통해 30tick 후의 교란 영역에 대해 선제적으로 파악할 수 있다. 위의 방법을 모두 적용하기 위해서는 태양풍 변화 주기와 위의 Task의 실행을 동기화해야 한다. 따라서 timer_interrupt를 활용하여 필요한 작업을 필요한 타이밍에 발생시켜 동기화한다.

Design : Cohort Protection

이 디자인은 문제가 발생할 가능성이 있는 스레드 전체를 실행되지 못하게 격리시키고, 손상될 데이터 또한 디스크에 통째로 백업 후 복구시킨다. 직관적으로 강력한 집단적 보호를 한다는 것을 알 수 있게 해당 디자인의 이름을 Cohort Protection이라고 명명한다. 이 디자인은 아래 도식도와 같이 작동한다.



[Fig 3] Cohort Protection의 전체 작동 과정

위 그림에서 보듯, 다음 Cycle에 태양풍의 영향을 받을 것으로 예상되는 프레임의 데이터를 디스크로 swap-out하여 보존하고, 해당 프레임과 연결된 페이지를 소유한 스레드의 is_sun을 true로 업데이트해 준다. 스케줄러는 is_sun이 true인 스레드를 실행시키지 않는다. 이후, 태양풍의 영향권을 벗어난 프레임에 다시 기존의 데이터를 swap-in시킨 후, 해당 프레임과 연결된 페이지를 사용하는 스레드의 is_sun을 false로 업데이트시켜 실행될 수 있도록 한다.

이 디자인의 목표를 달성하기 위해 세부 Task별로 나누어서 논의를 진행한다.

0. 사용할 변수**,** 함수 정의

Task들을 수행하기 위해 한 cycle 내에서 이전 cycle에 태양풍의 영향을 받던 프레임, 현재 영향을 받는 프레임, 다음 cycle에 영향을 받을 프레임을 기록하기 위해 커널 메모리 영역에 다음과 같은 변수를 정의하였다.

```
int prev_dist_frames[14]      /* Disturbed frames in previous cycle */

int curr_dist_frames[14]     /* Disturbed frames now */

int next_dist_frames[14]      /* Frame to be disturbed in next cycle */

int len_of_prev_dist_frames   /* The number of frames disturbed in previous cycle */

int len_of_next_dist_frames   /* The number of frames to be disturbed in next cycle */

int syn_off                /* Ticks offset to synchronize cycle */

struct frame* whole_frames[2000]      /* Pointers of whole struct frames */
```

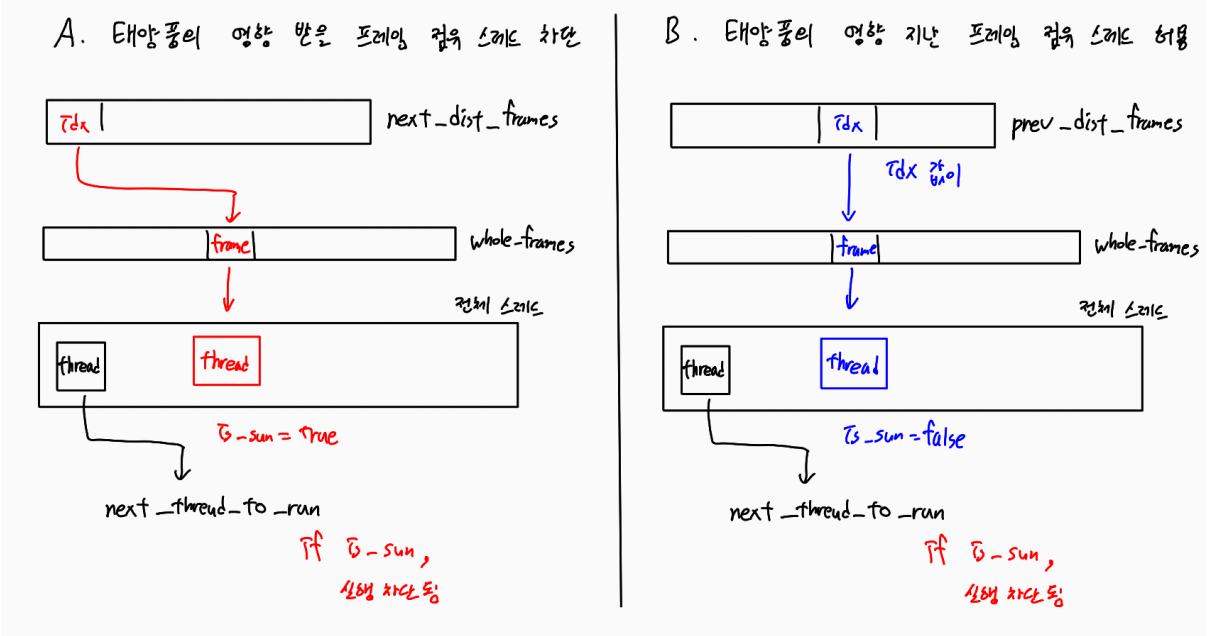
```
struct frame {
  …
   int disk_idx //disk index number for swap
   struct thread* thread; //the thread using this frame
```

```
int
frame_to_idx(struct frame* frame){
          return ((frame->kva)-ADDR_BASE)/PGSIZE;
}
  /* Return the index of frame using struct frame */
```

각 배열은 이전, 현재, 다음 cycle에 태양풍의 영향을 받을 프레임의 배열이고, 각 int는 각각 해당 시점의 영향 받는 프레임 개수, syn_off는 태양풍 변화 주기와 작업 주기 동기화를 위해 사용하는 ticks offset으로, 태양풍 변화 시점의 tick에 syn_off를 더하면 30의 배수가 된다. whole_frames는 전체 프레임 struct의 포인터를 저장한다.

1. 유저 스레드의 실행 차단

점유하는 프레임이 태양풍의 영향을 받고있는지 여부에 따라 유저 스레드의 실행을 차단시키고자 한다. 해당 task는 아래와 같이 진행된다.



[Fig 4] 태양풍의 영향을 받는 스레드의 스케줄 차단

위 그림과 같이, 태양풍의 영향을 받을 프레임의 스레드 점유를 is_sun이라는 boolean을 이용해 next_thread_to_run에서 차단시키고, 영향권을 벗어난 후에 다시 허용시키고자 한다.

먼저, thread structure에 is_sun이라는 bool 값을 추가해주었다. 해당 값은 해당 스레드가 점유한 프레임이 태양풍을 받고 있는지를 나타낸다. true일 경우 해당 스레드가 점유한 프레임이 태양풍의 영향권 아래 있음을 의미한다.

```
struct thread {
   // Omitted
   bool is_sun              /* Sun disturbed state */
}
```

다음으로, 스레드를 스케줄링할 때, next_thread_to_run()에서 is_sun을 확인하여 true일 경우 실행시키지 않는다.

```
struct thread*
 next_thread_to_run(void){
   // Omitted
   else{
      struct list_elem* tmp_elem = list_front(&ready_list);
      struct thread* tmp = list_entry(tmp_elem, struct thread, elem)
      while (tmp->is_sun){
         tmp_elem = tmp_elem->next;
         if (is_tail(tmp_elem)) return idle_thread;
```

```
            tmp = list_entry(tmp_elem, struct thread, elem);
        }
    return tmp;
    }
}
```

다음 Cycle에 태양풍의 영향을 받을 것으로 예측되는 프레임을 점유한 스레드의 is_sun을 true로 변경시켜주고, 이전 Cycle에 태양풍의 영향을 받은 프레임 중, 현재 영향을 받고 있거나 다음 번에 영향을 받을 예정이 아닌 경우에 해당 프레임을 점유한 스레드의 is_sun을 false로 변경시켜준다.

```
void is_sun_update_next(void){
    for (int i=0; i<14; i++){
        int idx = next_dist_frames[i];
        if (idx != NULL)
            whole_frames[idx]->thread.is_sun = true;
    }
}
void is_sun_update_prev(void){
    for (int i=0; i<14; i++){
        idx = prev_dist_frames[i];
        if (idx != NULL && !in_arr(idx, curr_dist_frames) && !in_arr(idx, next_dist_frames))
            whole_frames[idx]->thread.is_sun = false;
    }
}

bool
in_arr(int integer, int* arr)           /* Return true if integer is in array */
```

2. 손상될 프레임의 신규 할당 금지

태양풍의 영향을 받는 도중 프로세스가 페이지 신규 할당을 요청할 수 있다. 신규 할당 요청은 palloc_get_page가 palloc_get_multiple을 호출하면서 이루어진다. 따라서 신규 할당으로 리턴해주는 index 현재 태양풍 영역에 노출되는 경우, 다음과 같이 재탐색을 통해 알맞은 인덱스를 찾을 수 있다. palloc_get_multiple은 used_map이라는 bitmap 을 스캔하여 사용하지 않는(false)의 인덱스를 찾아 그 인덱스를 반환하였다. 이때, 반환한 인덱스가 curr_dist_frame(현재 태양풍의 영향을 받고 있는 frame list)에 존재하면 다시 다른 인덱스를 탐색한다. 따라서 태양풍의 영향을 받고 있는 frame으로의 신규 할당을 막는다.

뿐만 아니라 2000개의 프레임이 꽉 차 있는 경우 evict_frame을 통해 이미 할당받은 프레임에서 태양풍에 노출 중인 frame을 victim으로 선정하여, 태양풍 노출 중인 프레임에 swap_out과 swap-in을 하려는 시도가 있을 수 있다. 따라서 vm_get_victim 함수에서 반환하는 victim의 프레임이 태양풍에 노출되고 있으면 다른 victim을 찾도록 조건을 추가한다.

```
/* This function is called by palloc_get_page() */
void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt) {
    /* Scan bitmap and find the index that is marked as false */
    size_t page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);

    while(!in_arr(page_idx, curr_dist_frames)){
        /* Find new index until it does not return the frame under the sun */
        bitmap_set(pool->used_map, page_idx, false);
        page_idx=bitmap_scan_and_flip(pool->used_map,page_idx+1,page_cnt,false);
    }
    //Omitted
}

/* This function is called by vm_evict frame */
static struct frame *
vm_get_victim (void) {
    //Omitted
    struct list_elem* c;
    /* Search for the frame usable */
    for (c = list_front(&frame_table); c != list_end(&frame_table); c = c->next){
        victim = list_entry(c ,struct frame, elem);

        /* If the obtained frame is safe, return */
        if (!in_arr(frame_to_idx(victim), curr_dist_frame))
            return victim;
    }
    //Omitted
}
```

3. 손상 예정 데이터 디스크 백업

태양풍에 노출되면 데이터가 오염되므로, 우리는 이를 디스크에 저장하여 보관할 것이다. 따라서 특정 프레임의 데이터를 디스크에 복사하고, 그 정보를 저장하는 과정과 저장된 데이터를 복구하는 과정이 요구된다. 이를 위해 프레임과 디스크 간 데이터를 복사하는 supplementary 함수를 추가로 구현하였다.

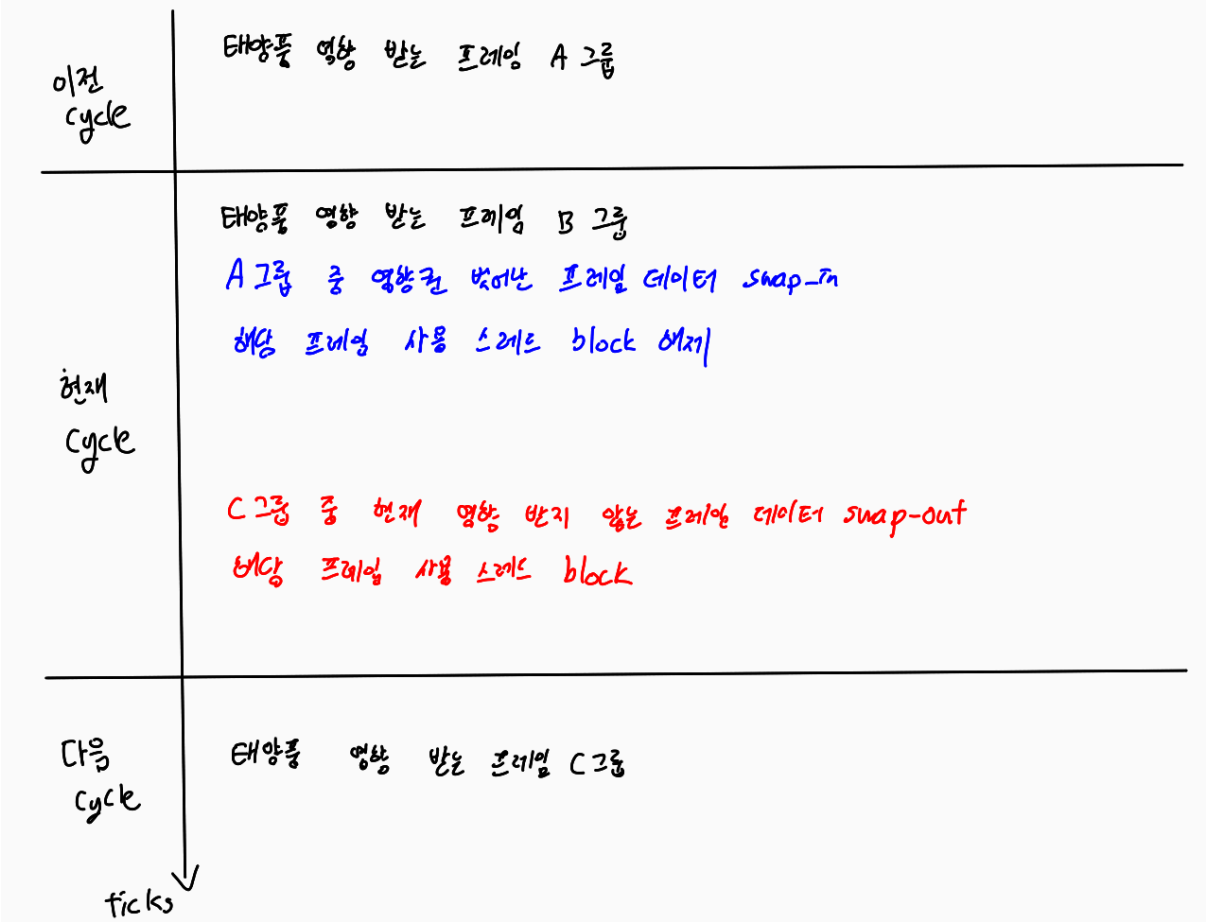
move_data_f2d 함수는 프레임의 데이터를 디스크에 복사하는 함수이다. 이 함수 내에서 복사한 디스크의 주소 정보를 프레임 구조체의 disk_idx에 저장한다. 이후, 이 정보를 반환받아 데이터를 복귀한다.

```
/* Supplementary function for copy using Disk */
 /* Move data from disk to dst frame */
void move_data_d2f(struct frame* dst){
        size_t index = frame->disk_idx;
        for( int i =0 ; i < sectors_in_page; i++){
        disk_read(swap_disk, sectors_in_page*index+i,src->kva+DISK_SECTOR_SIZE*i);
         }
        bitmap_set(swap_table, index, false);
}

/* Move data from src frame to disk */
void move_data_f2d(struct frame* src){
        size_t index = bitmap_scan_and_flip(swap_table,0, 1 ,false);
        for( int i =0 ; i < sectors_in_page; i++){
                disk_write(swap_disk, sectors_in_page *index + i ,
                                src->kva+DISK_SECTOR_SIZE*i);
        }
        frame->disk_idx = index;
}
```

위에서 14-length array에 프레임 index 정보를 저장하였다. 아래 함수들은 그 index를 이용해 접근한 프레임의 데이터를 디스크에 복사하고, 프레임에 복구한다.
오염될 데이터를 디스크에 저장해 보호하는 함수인 data_backup함수에서는 프레임이 사용중인 경우, 해당 프렝임이 현재 태양풍에 노출되지 않을 경우 디스크에 데이터를 복사한다.
반대로 디스크에 저장된 데이터를 프레임으로 복구하는 함수인 data_recover은 프레임이 다음 사이클에도 태양풍에 영향을 받는 경우를 제외하고 디스크에 저장된 데이터를 다시 프레임에 복구한다.

```
/* Swap in or out the frame corresponding the index of frame arrays*/
/* Store data before getting damaged : Move data from Frame to Disk*/
void data_backup(int idx){
      if(!in_arr(next_dist_frame[idx], curr_dist_frames)&&next_dist_frame[idx]->page)
      /* When the frame is not going to be affected by sun in the next cycle */
      move_data_f2d(whole_frames[next_dist_frames[idx]]);
}

/* Restore data to frames*/
void data_recover(int idx){
      if(!in_arr(prev_dist_frame[idx], curr_dist_frames)&&prev_dist_frame[idx]->page)
      /* When the frame is not already stored in disk in previous cycle */
      move_data_d2f(whole_frames[prev_dist_frames[idx]]);
   }
}
```

4. 태양풍 주기와 동기화



[Fig 5] 태양풍 주기에 맞춰 실행하는 작업

위 그림과 같이 전체 과정을 태양풍 주기에 맞게 동기화하여 실행한다. 매 ticks마다 timer_interrupt()가 호출되므로, 해당 함수 내에서 위의 함수들을 알맞은 타이밍에 호출한다. 각 작업들과 호출되어야 하는 타이밍은 다음과 같다.

1. cycle_update_global_variables() : 0 tick of current cycle
해당 함수는 위에서 정의한 global variable들을 새 cycle에 맞게 업데이트해 준다.
: prev_dist_frame으로 curr_dist_frame 복사하고, curr_dist_frame으로 next_dist_frame 복사한다. 뿐만 아니라 next_dist_frame으로 disturbed_frame_indices_at(current_ticks+30); 값을 리턴한다.

2. 영향 받을 프레임 점유 스레드 is_sun 업데이트 : 29 tick of current cycle
해당 작업은 다음 cycle이 되기 직전에 업데이트한다.

3. 영향 끝난 프레임 점유 스레드 is_sun 업데이트 : 13 tick of current cycle
해당 작업은 영향권을 벗어난 프레임들의 데이터를 복구한 후 진행한다. 14tick일 때, 영향권을 벗어난 프레임이 14개일 경우에도, 이 작업 진행 직후에 timer_interrupt가 끝나기

전에 데이터를 복구하므로, 스레드가 실행되기 전에 데이터가 복구되어 문제가 발생하지 않는다.

4. 영향 끝난 프레임 데이터 복구 : 0 ~ len_of_prev_dist_frames of current cycle
해당 작업은 새 Cycle이 되자마자 영향권을 벗어난 프레임들의 개수에 맞춰 진행한다.

5. 영향 받을 프레임 데이터 백업 : 30 - len of next_dist_frames ~29 of current cycle
해당 작업은 다음 Cycle이 되기 전, 영향을 받게 될 프레임들의 개수에 맞춰 진행한다.

태양풍 주기와 동기화하는 코드는 아래와 같다.

```
void
timer_interrupt(struct intr_frame *arg){
   //Omitted
   /* Update global variables to fit the next cycle */
   if ((ticks+offset)%30 == 0)
      cycle_update_global_variables(ticks);

   /* Block threads which are predicted to disturbed */
   else if ((ticks+offset)%30==29) {
      is_sun_update_next();
      if(thread_current()->is_sun)
         thread_yield();
   }

   /* Unblock threads which are now safe to run */
   else if ((ticks+offset)%30 == 13){
      is_sun_update_prev();
   }

   /* Swap-out data from frames which are predicted to disturbed */
   else if ((ticks+offset)%30 >= 30-len_of_prev_dist_frames)
      data_backup(29 - (ticks + offset)%30);

   /* Swap-in data from disk to frames */
   if ((ticks+offset)%30 < len_of_next_dist_frames )
      data_recover((ticks + offset)%30);

   //Omitted
}
```

Validation & Discussion

가장 시급한 문제인 해누리호의 폭발과 유저 프로그램 오동작을 막기 위한 디자인을 설계하였다. 위에서 설정한 우선순위별 평가 기준을 통한 이 디자인에 대한 평가는 다음과 같다.

1. 해누리호 폭발 방지
다음 cycle에 태양풍에 노출될 프레임을 선제적으로 예측하여 해당 프레임을 사용하는 스레드가 있을 경우에 해당 스레드의 실행을 원천적으로 차단한다. 따라서, 모든 경우에 태양풍에 노출되는 메모리 영역에 접근함으로써 발생하는 폭발 문제를 효과적으로 차단한다.

2. 데이터 손실 방지
다음 cycle에 태양풍으로 인해 오염될 데이터를 선제적으로 디스크에 백업한다. 이후, 해당 프레임이 태양풍의 영향을 벗어났을 경우에만 데이터를 복원하여 효과적으로 데이터를 보존한다. 또한, 해당 작업에 소요되는 시간은 최대 28ticks으로 최종 swap-in이 다음 cycle의 시작 이전에 종료되기 때문에, disk 접근에 소요되는 시간이 길더라도 데이터 백업으로 인한 노출영역 참조가 발생하지 않아 폭발 문제를 발생시키지 않는다.

3. 성능 저하 최소화
해누리호의 데이터 안정성 보장을 위해 디스크 접근이 빈번하게 발생한다. 사용하지 않는 프레임의 데이터는 백업하지 않으므로, 교란되는 프레임 개수가 동일하다면 디스크 접근 시간은 사용중인 프레임 수에 비례하여 증가한다. Best-case, Worst-case에서 데이터 백업에 소요되는 시간은 다음과 같다.

Best case: 0 ticks 소요
사용중인 프레임이 교란되지 않는 경우이다. 태양풍의 영향이 작을수록, 사용 중인 프레임이 적을수록 빈번하게 발생한다.

Worst case: 28 ticks 소요
교란 영역을 모두 사용 중인 경우이다. 태양풍의 영향이 최대일 때에 발생하며, 사용 중인 프레임이 많을수록 빈번하게 발생한다.

데이터 손실 방지를 위해 시간이 많이 소요되어 유저 프로그램이 실행될 수 있는 시간이 상당히 감소하였다. 또한, 유저 프로그램의 실행 자체를 원천적으로 차단시키기 때문에 유저 프로그램의 동작에 제한이 걸린다. 이러한 이유로 인해 성능 저하가 뚜렷하게 나타날 수 있다.

해당 디자인은 해누리호의 안전 보장을 위해 설계된 긴급 패치 OS로 최우선순위 문제상황인 폭발 방지와 데이터 안정성 확보는 효과적으로 달성하였다. 하지만, 성능 저하가 뚜렷하여, 탐사 중 유저 프로그램이 가동되는 수가 많아짐에 따라 효과적인 태양 탐사가 어려울 것으로 예측된다.

이를 개선하기 위해 해누리호의 안전은 보장하면서도 디스크 접근 시간을 최소화하여 성능을 개선할 수 있는 디자인이 요구된다.

Round 2 : Unemployed Frame Recruit

Observation

Cohort Protection은 태양풍에 노출될 메모리 영역을 점유하는 유저 스레드를 선제적으로 실행을 차단시켜 노출 영역에 대한 접근을 차단했다. 또한, 태양풍으로 인해 교란될 데이터를 사전에 디스크에 백업하고, 영향권을 벗어난 후에 이를 복구하여 메모리 유실 문제 또한 해결하였다.

하지만, 사용중인 프레임의 개수가 많아짐에 따라 데이터 보호를 위해 잦은 디스크 접근이 발생하여, 데이터 이전에 소요되는 시간이 길어질 수 있는 문제가 발생하였다. 이에 따라 유저 프로그램의 실행에 사용할 수 있는 시간이 상당히 줄어들어 성능 저하가 매우 뚜렷하게 나타나는 시간 효율성 문제가 발생한다.

Idea

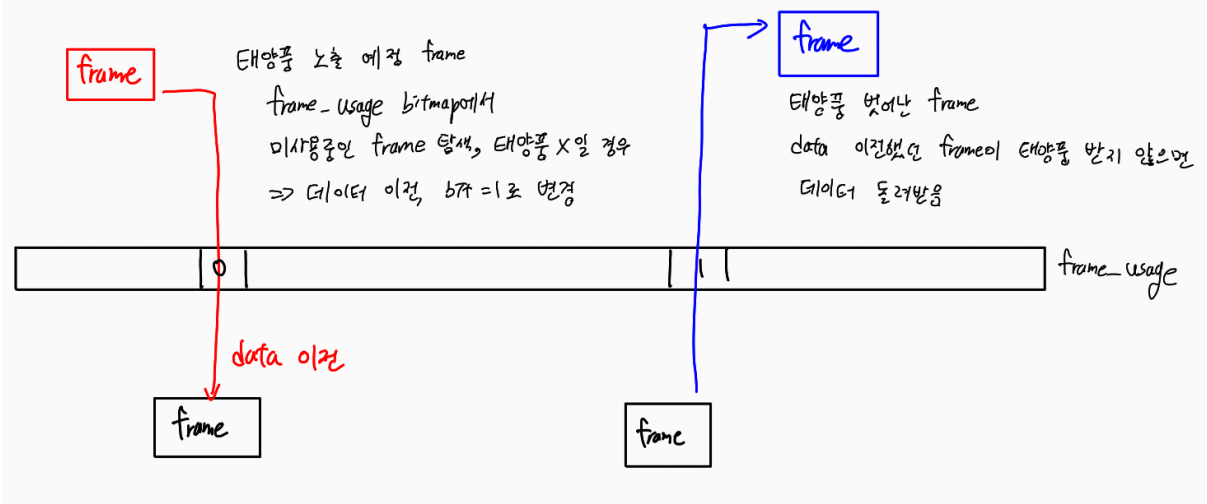
디스크 접근 시간은 메모리 접근 시간에 비해 매우 느리다. 이전 디자인은 데이터를 디스크에 백업하여 많은 시간이 디스크 접근에 소모되어 유저 프로그램이 작동할 수 있는 충분한 시간을 확보하지 못하였다. 따라서 이번 디자인에서는 디스크 대신 **RAM**영역을 활용하여 데이터를 백업하고자 한다.

사용하지 않는 **RAM**영역을 데이터를 백업하기 위한 장소로 사용할 것이다. 이를 위해 RAM영역 내 비어있는 프레임을 찾아 이 곳에 데이터를 임시로 백업한 후, 태양풍의 영향이 끝난 후에 원래 프레임으로 데이터를 복구할 것이다. 다만, 비어있는 프레임이 없을 경우에는 이전과 마찬가지로 해당 데이터를 디스크에 백업한 후, 태양풍의 영향이 끝난 후에 다시 기존 프레임으로 복구할 것이다.

이 방법을 사용하면 디스크 대신 비어있는 프레임을 활용하여 데이터의 백업을 진행하기 때문에 디스크 접근 시간을 효과적으로 단축시킬 수 있다. 이를 통해 유저 프로그램이 실행될 수 있는 시간을 확보하여 성능 개선이 이뤄질 것이다.

Design

이 디자인은 사용하지 않는 프레임에 태양풍의 영향을 받을 데이터를 백업해둔 후 태양풍의 영향이 끝나면 데이터를 복구시킨다. 사용하지 않는 프레임을 사용한다는 의미에서 직관적으로 이해할 수 있도록 이 디자인의 이름을 Unemployed Frame Recruit라고 명명하였다. 이 디자인은 Cohort Protection을 기반으로 하여 만들어졌다. 즉, 변경되지 않은 사항은 이전과 같다. 이 디자인은 아래 도식도와 같이 작동한다.



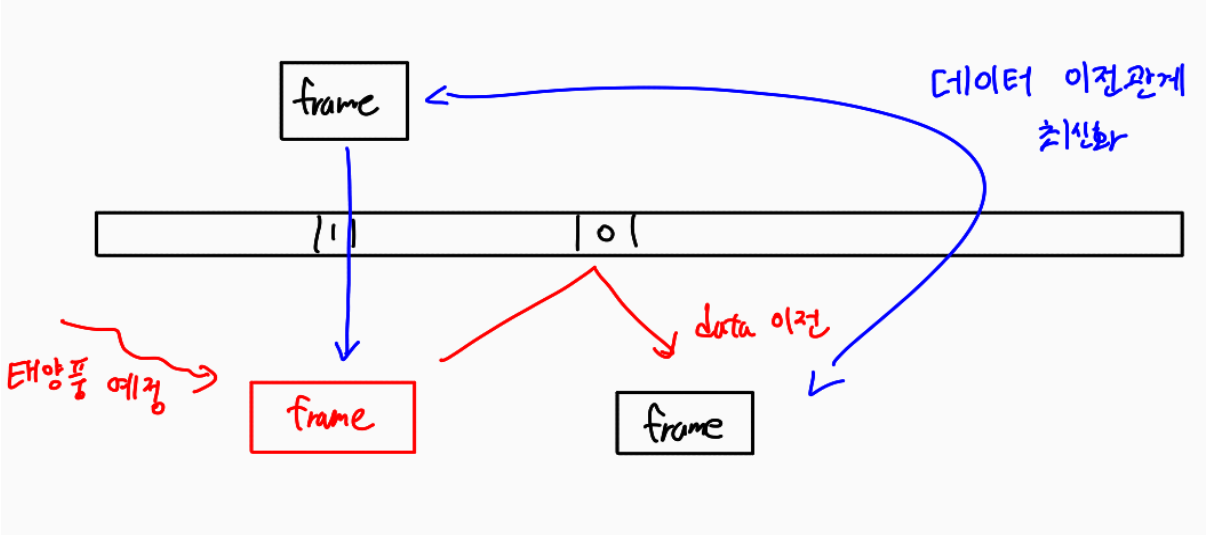
[Fig 6] 태양풍 영향 데이터의 프레임 간 이전

위 Figure처럼 이 디자인은 사용하지 않는 프레임을 frame_usage_bitmap을 통해 찾고, 찾은 프레임으로 오염될 데이터를 사전에 다른 안전한 프레임 영역으로 백업한 뒤, 태양풍의 영향이 끝나면 복구할 것이다. 미사용중인 안전한 프레임이 없어 RAM내에서 데이터를 백업할 수 없을 때에는 이전과 같은 방식으로 디스크에 데이터를 백업한다.

프레임간에 데이터를 이전받은 프레임과 돌려줄 프레임의 위치를 기억하기 위해 아래와 같이 struct frame에 각각의 index를 기록한다.

```
struct frame {
    //Omitted
    int from_frame;     /* Frame index which gave data */
    int to_frame;       /* Frame index where data backuped */
};
```

각각 NULL인 경우에는 데이터의 백업에 사용하지 않았음을 의미한다.
또, 디스크에 swap-out한 경우에는 to_frame을 3000으로 설정한다.



[Fig 7] 프레임들 내 데이터 다중 이전

다만 위의 그림과 같이 데이터 백업에 사용한 프레임이 데이터를 다시 돌려주기 전에 태양풍의 영향을 받게 되는 경우에는 데이터의 손실이 발생할 수 있다. 이를 막기 위해, 태양풍에 노출 예정인 프레임이 데이터 백업 용도로 사용되었는지 확인한 후, 맞을 경우 새 안전한 비어 있는 프레임에 데이터를 이전시킨 후, 두 프레임 간의 데이터 이전 관계를 최신화해 준다.

먼저 각 프레임의 사용 여부를 나타내는 frame_usage bitmap을 추가한다.

```
/* Represent frames are used or not */
struct bitmap* frame_usage = bitmap_create (2000);
```

다음은 데이터를 백업하는 함수이다.

```
void
data_backup(void){ /* Store Data */
   for (int i=0; i<14; i++){
      int idx = next_dist_frames[i];
      if (idx != NULL){
         size_t bef_idx = whole_frame[idx]->from_frame;
         /* Generally, bef_idx is NULL except for the case this frame contains other frame's
         Data temporarily*/

         /*Special Case: The frame that keeps other frame's data :
         If the data was backed up from another and can be recoverd now */
         if (bef_idx != NULL) {
            if (!in_arr(whole_frame[bef_idx], curr_dist_frames)
                && !in_arr(whole_frame[bef_idx], next_dist_frames)) {

               move_data_f2f(whole_frame[idx], whole_frame[bef_idx]);
               whole_frame[bef_idx]->to_frame = NULL;
               whole_frame[idx]->from_frame = NULL;
               goto fin;
            }
         }

         /* General Case */
         size_t to_idx = bitmap_scan(frame_usage, 0, 1, false);
         if (to_idx == BITMAP_ERROR)
            goto disk;

         /* Find a safe frame which is empty */
         while (in_arr(to_idx, curr_dist_frames) || in_arr(to_idx, next_dist_frames)){
            to_idx = bitmap_scan(frame_usage, to_idx+1, 1, false);
```

```
            if (to_idx == BITMAP_ERROR)
                goto disk;
        }
        bitmap_flip(frame_usage, to_idx);

        /* If the data was backed up from another frame */
        if (bef_idx != NULL){
            whole_frame[bef_idx]->to_frame = to_idx;
            whole_frame[to_idx]->from_frame = bef_idx;
            whole_frame[idx]->from_frame = NULL;
        }

        /* If the data was not backed up */
        else{
            whole_frames[idx]->to_frame = to_idx;
            whole_frames[to_idx]->from_frame = idx;
        }

        /* Backup data to frame */
        move_data_f2f(whole_frame[idx], whole_frame[to_idx]);
    }
fin:
    continue;

disk:
    /* If the data was backed up from another frame */
    if (whole_frame[idx]->from_frame != NULL){
        whole_frame[whole_frame[idx]->from_frame]->to_frame = 3000;   // Data moved to disk
        whole_frame[idx]->from_frame = NULL;
    }
    else {
        whole_frame[idx]->to_frame = 3000;
    }

    /* Backup data to disk */
    move_data_f2d(whole_frame[idx]);
  }
}
```

위 함수에서는 먼저 데이터를 이전할 수 있는 안전한 비어있는 프레임이 있는지를 확인한다. 여기에서 안전한 프레임은 현재 교란중이 아니며, 다음 Cycle에 교란 예정이지 않은 프레임을 의미한다. 만약 비어 있는 프레임이 있다면, 빈 프레임으로 데이터를 이전하고, 없을 경우에는 디스크에 swap_in한다. 만약 데이터가 원래 있던 프레임 또한

이전에 다른 프레임으로부터 데이터를 이전받았다면, 그 이전 프레임이 복구 가능한 상태인지 확인하여 복구한다. 복구 가능한 상태가 아니라면 연결 관계를 최신화해 준다.

다음은 데이터를 복구하는 함수이다.

```
void
data_recover(int i){
        idx = prev_dist_frames[i];
        if (!in_arr(whole_frame[idx], curr_dist_frames)
            && !in_arr(whole_frame[idx], next_dist_frames)){

            /* If data backed up to a frame */
            if (whole_frame[idx]->from_frame != 3000){
                bitmap_flip(frame_usage, whole_frame[idx]->to_frame);
                move_data_f2f(whole_frame[whole_frame[idx]->from_frame], whole_frame[idx]);
                whole_frame[whole_frame[idx]->to_frame]->from_frame = NULL;
                whole_frame[idx]->to_frame = NULL;
            }

            /* If data backed up to a disk */
            else{
                move_data_d2f(whole_frame[idx]);
                whole_frame[idx]->to_frame = NULL;
            }
    }
```

위 함수에서는 기존 프레임이 태양풍의 위협을 벗어나 데이터를 복구할 수 있게 된 경우, 프레임에 백업하였는지, 디스크에 백업하였는지의 여부를 확인하여 데이터를 복구한다.

위 함수들에서 사용한 데이터를 이전하는 함수는 다음과 같다.

```
void
move_data_f2f(struct frame* src, struct frame* dst){    /* Move data from src frame to dst frame*/
    memcpy(dst->kva, src->kva, PGSIZE);
}
```

또, bitmap_frame_usage를 사용하기 위해서 프레임이 신규 할당될 때에도 이를 업데이트하여 주어야 한다.

```
static struct frame *
vm_get_frame (void) {
    //Omitted
    /* Set the added features of frame */
    frame->to_frame = NULL;
    frame->from_frame = NULL;
    int idx = frame_to_idx(frame);
```

```
    bitmap_set(frame_usage, idx, true);
    return frame;
}
```

위 함수에서는 프레임이 할당될 때, from_frame, to_frame을 초기화해 주고, frame_usage에서 해당 프레임의 사용 여부를 업데이트한다. 마찬가지로 프레임을 반납할 때, 해당 영역의 사용 여부를 아래와 같이 업데이트해 준다.

```
void
vm_dealloc_page (struct page *page) {
    int idx = frame_to_idx(page->frame);
    bitmap_set(frame_usage, idx, false);
    //Omitted
}
```

timer_interrupt내에서 호출하는 과정은 이전과 동일하므로 생략한다.

Validation & Discussion

위에서 설정한 우선순위별 평가 기준에 따른 Unemployed Frame Recruit의 평가는 다음과 같다.

1. 해누리호 폭발 방지
해당 디자인은 이전 디자인과 동일하게 태양풍의 영향을 받는 프레임을 점유하는 스레드의 실행을 막아 접근을 원천적으로 차단시킨다. 또한 태양풍의 영향을 받는 동안 태양풍의 영향을 받는 할당되지 않은 프레임들이 할당되지 않기 때문에 메모리 접근 문제로 인한 폭발이 발생하지 않는다.

2. 데이터 손실 방지
태양풍이 도달하기 전 손상될 것으로 예측되는 데이터를 영향을 받지 않는 빈 프레임으로 이전해 준다. 빈 프레임이 없을 경우에는 디스크에 백업하고, 영향이 끝나면 다시 원래 프레임으로 데이터를 백업한다. 따라서, 태양풍의 영향을 받기 이전에 데이터가 백업되어 손실이 발생하지 않는다. 해당 과정은 Cycle이 바뀌어 태양풍의 영향을 받기 이전에 완료되기 때문에 이로 인한 해누리호 폭발 문제도 발생하지 않는다.

3. 성능 저하 최소화
사용하지 않는 프레임이 있을 경우에 해당 프레임으로 데이터를 이전한다. RAM 내에서 데이터를 이전하는 데 소요되는 시간은 무시할 정도로 작으므로, 빈 프레임이 존재한다면, 데이터 이전으로 인한 시간 효율성 문제가 발생하지 않아 손상되기 전 해누리호의 작동에 비해 성능이 크게 저하되지 않는다.

다만, 사용하지 않는 프레임의 개수가 14개 미만이라면, RAM내에서 데이터를 이전할 수 없어 디스크에 데이터를 백업하게 되는 경우가 발생할 수 있다. 이렇게 될 경우, 디스크 접근 시간의 발생으로 인해 유저 프로그램이 실행 가능한 시간이 급격하게 감소하여 시간 효율성 문제가 발생한다. 또, 유저 프로그램의 실행을 원천적으로 차단하기 때문에 유저 프로그램의 실행에 제한이 발생한다.

Unemployed Frame Recruit의 Best, Worst case에서 데이터 백업에 소요되는 시간은 다음과 같다.

Best case: 0 ticks 소요
사용하지 않는 프레임의 개수가 교란되는 프레임의 개수보다 적지 않을 때 발생한다. 2000개 중 거의 대부분의 프레임이 사용중이지만 않다면 이 경우에 해당한다.
Worst case: 28 ticks 소요
모든 프레임이 사용 중일 때 이전과 다음 Cycle에 14개의 프레임이 교란받을 때 발생한다.

해당 디자인은 최우선순위 해결 과제인 해누리호 폭발 방지와 데이터 손실 방지를 효과적으로 이뤄냈다. 또, 성능 문제도 이전의 디자인에 비해 상당히 개선되어, 메모리 영역을 대부분 사용하지 않는다면 크게 문제가 발생하지 않는다. 하지만, 탐사 중 동시에 실행해야 하는 유저 프로그램이 많아질 경우, 대부분의 프레임을 사용하게 되는 경우가 빈번히 발생할 수 있다. 이 경우에는 디스크 접근 시간의 증대로 인해 급격한 성능 저하가 일어날 것이며, 해누리호의 성공적인 탐사에 걸림돌이 될 수 있다.

따라서, 최악의 상황에서도 디스크 접근시간을 발생시키지 않아 성능 저하가 일어나지 않는 디자인의 설계가 요구된다.

Round 3 : Memory Storehouse

Observation

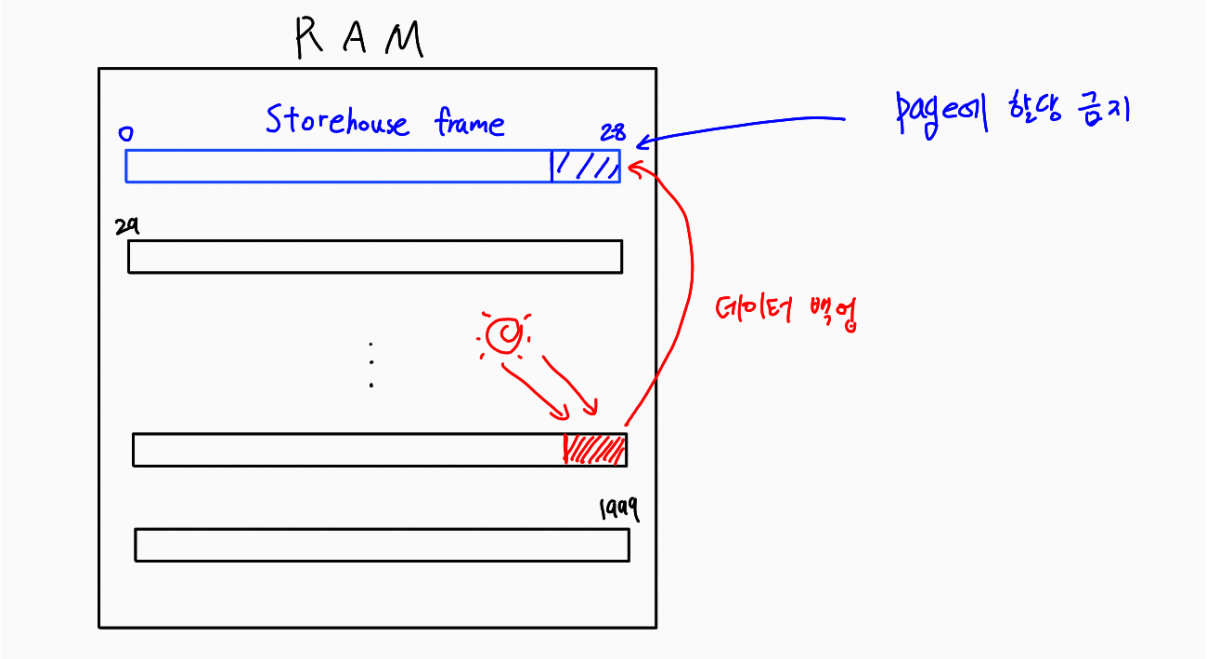
Unemployed Frame Recruit는 Cohort Protection의 토대 위에서 설계되어 해누리호의 폭발 방지와 데이터 손실 방지를 이뤄냈고, 디스크 접근 시간을 최소화하여 성능을 향상시키고자 하였다. 이는 사용하지 않는 프레임을 백업 장소로 이용하여 디스크 접근 시간을 단축시키는 방법을 통해 달성하였다. 따라서, 프레임이 거의 대부분 사용되는 경우가 아니라면 성능 개선이 상당히 이뤄졌다.

다만, 해누리호가 탐사를 지속함에 따라 가동중인 애플리케이션이 많아져 대부분의 프레임을 사용하게 되는 경우, 시간 효율성 문제가 발생하여 최악의 경우에는 Cohort Protection과 비슷한 성능을 보일 수 있다. 이 경우 성능은 크게 감소하여 30tick당 28tick을 디스크 접근에 사용하게 된다.

따라서, 최악의 경우에도 디스크 접근을 발생시키지 않아 급격한 성능 저하가 나타나지 않는 디자인이 요구되었다.

Idea

이번 디자인의 핵심 아이디어는 **RAM**의 일부분을 스레드에 할당하지 않고**,** 임시 저장소의 용도로만 사용하는 것이다. 다시 말해, 이전에는 사용하지 않는 프레임을 찾아 데이터를 백업하는 데에 사용했다면, 이번에는 아예 일부 프레임을 데이터 백업 용도로 지정하는 것이다. 이 프레임 영역을 **Storehouse frame**이라고 한다. 이를 간단히 도식화한 그림은 아래와 같다.



[Fig 8] Memory Storehouse 전체 작동 과정

한 시점에 태양풍의 영향을 받는 프레임의 최대 개수는 14개, 다음 번 태양풍의 영향을 받는 프레임의 개수도 14개이다. 따라서, 한 시점에 메모리 내 임시 저장소로 사용해야 하는 안전한 프레임의 개수는 28개가 되어야 한다. Storehouse frame에 태양풍이 도달하는 경우에도, 해당 프레임의 개수만큼 다른 구역에서 영향을 받을 수 있는 최대 프레임의

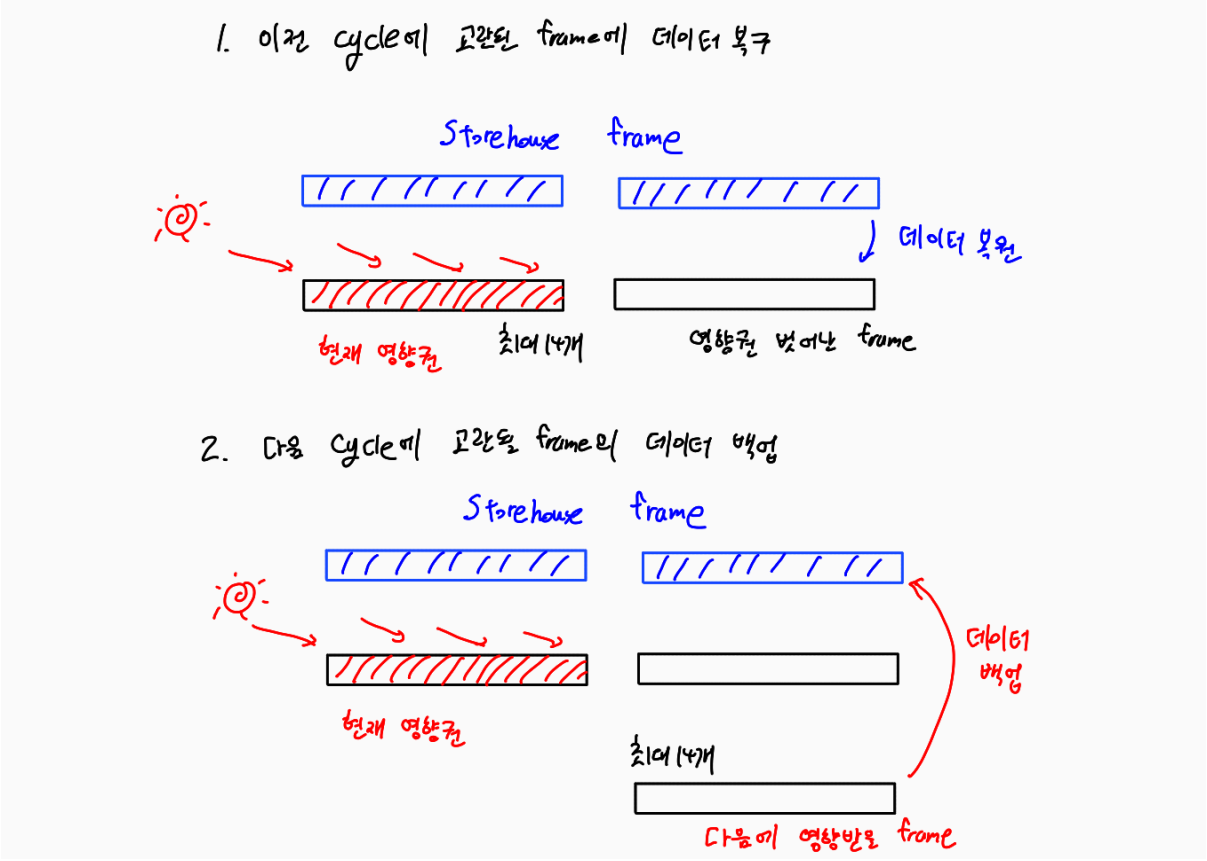
개수는 줄어들기 때문에 총 28개의 Storehouse frame으로 데이터를 모두 백업할 수 있다. 따라서, 전체 2000개 프레임 중 28개를 제외한 1972개의 프레임만을 기존 용도로 사용하게 된다.

이 아이디어를 사용하면 비록 28개 프레임을 사용하지 못하게 되지만, 데이터 백업으로 인한 디스크 접근을 원천적으로 차단시켜 대부분의 프레임을 사용 중인 경우에도 성능 저하가 발생하지 않으며, 최악의 경우에도 디스크 접근 시간 발생으로 인한 성능 저하를 막을 수 있다.

Design

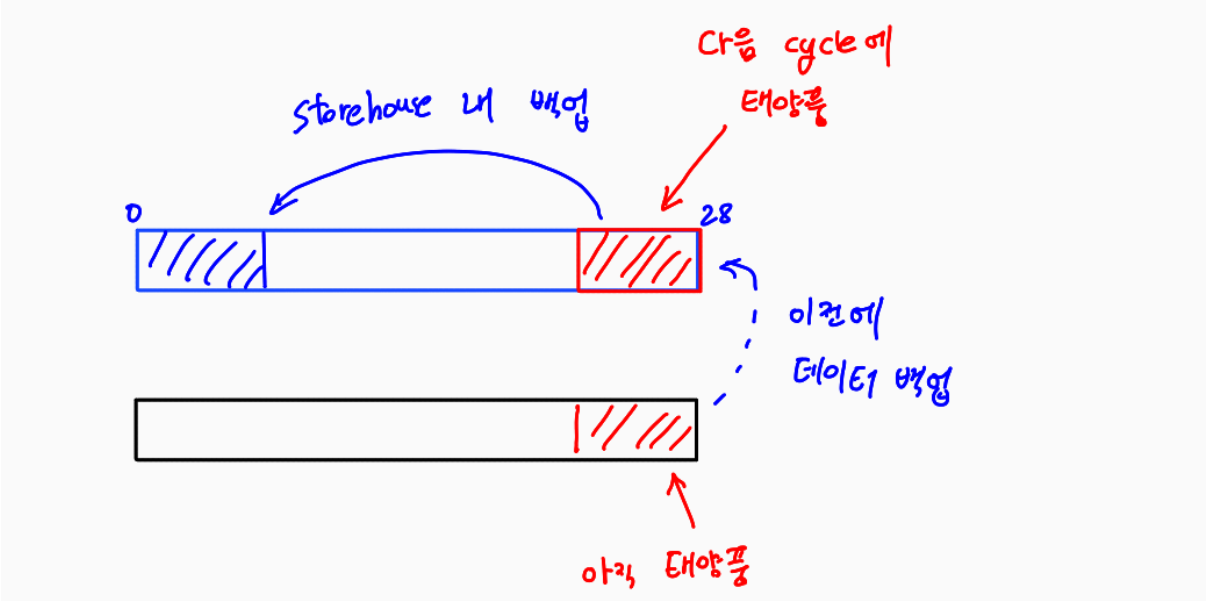
Unemployed Frame Recruit의 토대 위에서 28개 프레임을 다른 프레임들처럼 일반적인 용도로 사용하지 않고, Storehouse frame으로 지정하여 다른 프레임들의 데이터를 백업하고자 한다. 이 디자인의 특징인 일부 메모리를 백업용으로 사용한다는 것에서 착안하여 이 디자인의 이름을 Memory Storehouse라고 명명하였다. 2000개 프레임 영역 중 가장 앞의 0~27번째 프레임 영역을 Storehouse frame으로 사용할 것이다.

한 Cycle 내에서 현재 교란받는 중인 프레임의 데이터는 그대로 Storehouse frame에 두고, 이전 Cycle에 태양풍의 영향을 받고 영향이 끝난 프레임으로 데이터를 이전해야 한다. 이후, 다음 Cycle에 태양풍의 영향을 받을 것으로 예정되는 프레임들의 데이터를 빈 Storehouse frame으로 이전한다. 이 과정은 다음과 같은 그림으로 설명할 수 있다.



[Fig 9] 한 Cycle 내 데이터 이전

다만 아래 그림처럼, Unemployed Frame Recruit때와 마찬가지로 원래 데이터가 태양풍을 받고 있어 데이터를 복원하지 못하는 상황에, Storehouse frame이 태양풍의 영향을 받아 데이터가 오염될 가능성이 있다. 따라서, 이 경우에 Storehouse frame 내에서 데이터를 이전하고, 프레임간의 연결 관계를 from_frame과 to_frame을 최신화해 줌으로써 업데이트해 준다.



[Fig 10] Storehouse 내 데이터 다중 이전

구현을 위해 우선 Storehouse frame 중 사용중인 영역을 나타내는 비트맵인 Storehouse_usage를 추가하였다.

```
/* Represent storehouse frames are used or not */
struct bitmap* storehouse_usage = bitmap_create (28);
```

이 비트맵은 28개 Storehouse frame의 사용 여부를 나타낸다.

뿐만 아니라 0~27의 index를 가지는 프레임에 대한 할당을 금지하였다.

in palloc.c

```
/* Use frame index start from 28th in the palloc_get_page(USER) */
size_t page_idx = bitmap_scan_and_flip (user_pool->used_map, 28, page_cnt, false);
```

위 Unemployed Frame Recruit에서 사용한 data_backup 함수와 data_recover 함수를 수정하였다.

```
void
data_backup(void){ /* Store data that will be damaged by Sun*/
   for (int i=0; i<14; i++){
      int idx = next_dist_frames[i];
      if (idx != NULL){
         size_t bef_idx = whole_frame[idx]->from_frame;
```

```
        /*Special Case :  If the data was backed up from another and can be recovered now */
        if (bef_idx != NULL) {
            if (!in_arr(whole_frame[bef_idx], curr_dist_frames)
                && !in_arr(whole_frame[bef_idx], next_dist_frames)){
                /* when the given storehouse frame is already used as a storehouse,
                    and its original frame is safe now, we can  move the data to the original frame*/
                move_data_f2f(whole_frame[idx], whole_frame[bef_idx]);
                whole_frame[bef_idx]->to_frame = NULL;
                whole_frame[idx]->from_frame = NULL;
                goto fin;
            }
        }

        /* General Case*/
        size_t to_idx = bitmap_scan(storehouse_usage, 0, 1, false);

        /* Find a safe frame which is empty */
        while (in_arr(to_idx, curr_dist_frames) || in_arr(to_idx, next_dist_frames)){
            to_idx = bitmap_scan(storehouse_usage, to_idx+1, 1, false);
            ASSERT(to_idx != BITMAP_ERROR);
        }
        bitmap_flip(storehouse_usage, to_idx);

        /* If the data was backed up from another frame */
        if (bef_idx != NULL){
            whole_frame[bef_idx]->to_frame = to_idx;
            whole_frame[to_idx]->from_frame = bef_idx;
            whole_frame[idx]->from_frame = NULL;
        }

        /* If the data was not backed up */
        else{
            whole_frames[idx]->to_frame = to_idx;
            whole_frames[to_idx]->from_frame = idx;
        }

        /* Backup data to frame */
        move_data_f2f(whole_frame[idx], whole_frame[to_idx]);
    }
fin:
```

```
        continue;
    }
}
```

data_backup 함수에서는 백업 장소로 Storehouse frame만을 사용하며 Storehouse frame 중 어느 프레임이 사용 가능할지는 storehouse_usage 비트맵을 스캔하여 판단한다. 데이터 보존을 위해 디스크를 사용하는 경우는 발생하지 않는다. 데이터의 다중 이전이 발생하는 경우에는 데이터를 이전한 후, 이 프레임들 간의 연결 관계를 최신화해 준다. 또, 전체 프레임 영역 중 태양풍의 영향을 받는 부분을 파악하기 때문에, Storehouse frame이 태양풍의 영향을 받는 경우에도 데이터가 이전된다.

```
void
data_recover(void){
    int idx;
    for (int i=0; i<14; i++){
        idx = prev_dist_frames[i];
            /* Check the safety of the frame*/
        if (!in_arr(whole_frame[idx], curr_dist_frames)
            && !in_arr(whole_frame[idx], next_dist_frames)){

            /* Recover data from Storehouse frames */
            bitmap_flip(storehouse_usage, whole_frame[idx]->to_frame);
            move_data_f2f(whole_frame[whole_frame[idx]->from_frame], whole_frame[idx]);
            whole_frame[whole_frame[idx]->to_frame]->from_frame = NULL;
            whole_frame[idx]->to_frame = NULL;
        }
    }
}
```

data_recover 함수에서는 Storehouse frame에 저장해뒀던 데이터를 기존 프레임이 태양풍의 영향권을 벗어났을 경우 다시 복구한다. 디스크에 데이터를 저장하지 않았기 때문에, 디스크 저장 여부를 확인해 디스크에서 데이터를 불러오는 부분은 제거되었다.

데이터 이전을 timer_interrupt와 동기화하는 부분은 이전과 동일하므로 수정하지 않는다.

Validation & Discussion

우선순위별 평가 기준에 따른 Memory Storehouse의 평가는 다음과 같다.

1. 해누리호 폭발 방지
태양풍의 영향을 받는 프레임을 점유하는 스레드의 실행을 막아 해당 프레임 영역에의 접근을 원천적으로 차단시킨다. 또, 사용중이지 않은 태양풍의 영향을 받는 프레임들이 스레드에 할당되는 것을 방지하여 메모리 접근 문제로 인한 폭발이 발생하지 않는다.

2. 데이터 손실 방지
태양풍이 도달하기 전에 손상될 것으로 예측되는 데이터를 Storehouse frame 중 태양풍의 영향을 받지 않는 영역으로 백업한다. 이후 태양풍의 영향권을 벗어나면 다시 기존 프레임에 데이터를 복구한다. 해당 과정은 한 Cycle 내에서 일어나며 메모리 접근 시간은 무시할 수 있으므로 데이터 백업으로 인한 해누리호 폭발 문제도 발생하지 않는다.

3. 성능 저하 최소화
데이터 손실 방지를 위해 Storehouse frame을 임시 저장소로 활용한다. 이 때, 백업을 위한 모든 데이터 복사가 메모리 내에서만 이뤄지기 때문에 디스크에 접근하지 않아 성능 저하가 크게 발생하는 특정한 경우가 없다. 이 측면에서 이 디자인은 RAM의 크기를 Resizing하여 시간 효율성 문제를 해결하였다고 볼 수 있다.

이 디자인은 태양풍의 영향을 많이 받지 않을 경우 28개의 프레임 영역을 사용하지 않아 메모리 측면에서 비효율이 발생한다고 볼 수 있다. 하지만, 전체 RAM영역 중 그 영역이 차지하는 비율이 미미하고, 성능 개선으로 인해 각 스레드가 기존 디자인보다 일찍 종료될 수 있기에, 평균적인 사용중인 frame개수가 감소할 수 있어 실제로 성능 저하에 끼치는 영향은 미미할 것으로 예측된다.

위의 두 이유로 해누리호의 안정성 보장을 위해 많은 리소스가 사용되고 있지는 않다고 볼 수 있다. 다만, 태양풍에 노출되고 있는 프레임을 점유한 스레드들이 작동이 중지됨으로써 유저 프로그램의 동작에 제한이 발생하고, 이는 성능 저하로 이어질 수 있다. 특히, 작동이 정지된 스레드의 lock 점유 등으로 인해 deadlock현상이 발생한다면, 이는 큰 성능 저하로 이어진다.

따라서 성능 개선을 위해 시간 효율성 문제 외에 해누리호의 폭발을 방지하면서 스레드의 작동을 중지시키지는 않는 디자인이 요구된다. 이에 대한 논의는 지금까지 성능 개선을 위해 다루었던 디스크 접근 시간 최소화 외의 다른 측면에서의 성능 개선의 지평을 열어준다.

Round 4 : Remapping

Observation

Memory Storehouse는 태양풍 노출 프레임 영역에 스레드의 접근을 원천적으로 봉쇄시켜 해누리호의 폭발을 방지하고, Storehouse frame에 데이터를 안전하게 백업시켜 데이터의 손실을 방지했다. 또한, 데이터의 백업을 디스크에 하지 않고, 메모리 중 일부 영역을 사용하여 진행함으로써 디스크 접근 시간을 발생시키지 않아 성능 개선을 이루어냈다.

하지만, 해누리호 폭발 방지를 위해 일부 스레드의 실행을 영향 기간동안 차단시키기 때문에, 유저 프로그램의 동작에 제한이 발생하여 성능 저하가 발생할 수 있었다. 특히, deadlock현상이 발생할 경우 이는 큰 성능 저하로 이어질 수 있다.

따라서, 해누리호의 폭발을 여전히 방지함에도 유저 스레드의 실행을 차단시키지는 않는 방법이 요구되었다. 또한 디자인 목표 우선순위에 부합하도록 데이터의 손실도 방지하여야 한다.

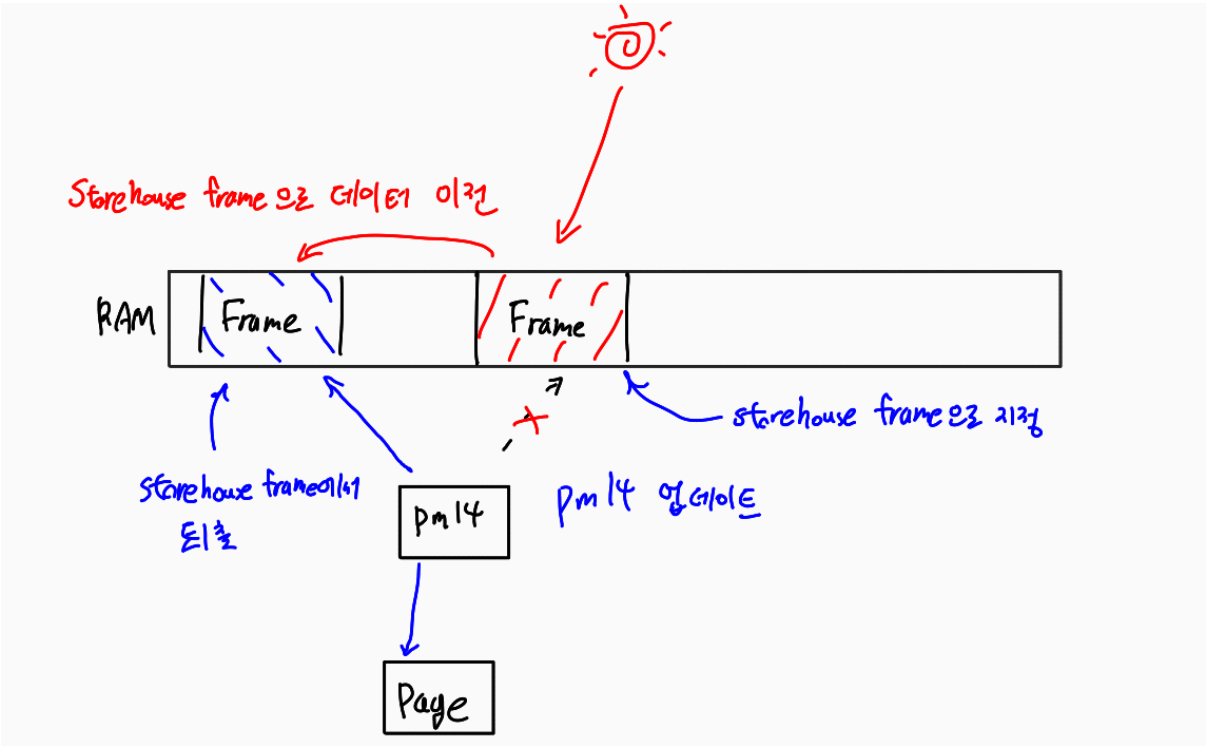
Idea

이번 디자인에서는 스레드의 실행을 차단시키지 않는 디자인을 구상하고자 하였다. 스레드의 실행이 차단되지 않기 위해서는 스레드가 항상 실행 가능한 상태여야 한다. 따라서 스레드가 점유한 프레임이 손상 예정일 경우, 스레드가 다른 프레임을 사용하여 실행될 수 있어야 한다.

이를 충족시키기 위해 데이터를 보유한 프레임이 손상 예정일 경우, 이를 비어 있는 프레임으로 이동시킨 후, 새 프레임과 스레드 페이지를 연결할 수 있도록 plm4를 업데이트시킬 것이다. 이는 이전에는 28개의 빈 프레임을 Storehouse frame으로 항상 지정하여 그 목적으로만 쓰던 것과 다르게, 28개의 빈 프레임을 두되, 매 Cycle마다 태양풍의 영향에 따라 **Storehouse frame**으로 지정되어 있던 프레임이 주기적으로 변화하게 됨을 의미한다. 이를 대략적으로 묘사한 모식도는 아래와 같다.

이 방법을 통해 데이터를 사전에 이전하면, 유저 스레드의 손상중인 프레임에 대한 접근을 막을 수 있으며, 데이터의 손실을 방지할 수 있을 뿐 아니라, 유저 스레드의 실행을 차단하지 않기 때문에, 이로 인한 성능 저하가 발생하지 않는다.

Design



[Fig 11] Remapping 작동 과정

이 디자인에서는 위 그림처럼 태양풍으로 인해 손상 예정인 프레임의 데이터를 사용 가능한 Storehouse frame이전시킨 후, 이전받은 새 프레임과 페이지가 연결되도록 pml4를 업데이트해 준다. 또한, list storehouse를 이용해 Storehouse frame으로 지정된 프레임이 누구인지 최신화해준다. 이는, 데이터를 이전받은 Storehouse frame을 지정 해제하고, 데이터를 이전한 기존 프레임을 Storehouse frame으로 지정시키는 것이다.

이전 Memory Storehouse에서는, 0~27번째 프레임을 Storehouse frame으로 명명해 사용하였다. 하지만 이번 디자인은 빈 프레임의 위치가 유동적으로 변경된다. 따라서 이를 추적하기 위한 Storehouse 로 사용될 frame에 대한 정보를 저장할 structure가 필요하다. 여기서는 desig_storehouse라는 리스트를 사용해 임시적으로 사용할 프레임을 리스트로 관리할 것이다.

리스트를 선언하고, 이 리스트에 0~27까지의 index 주소를 할당받은 프레임을 넣어주는 과정을 storehouse_init()에 선언한다.

```
struct list desig_storehouse;

void storehouse_init (void) {
   list_init(&desig_storehouse);
   for (int i=0; i<28; i++){
      //this is the first palloc_get_page(USER) request
      //So 0~27 frame will be allocated in this frame.
      struct frame* frame=vm_get_frame();  // explanation below
      list_push_back(&desig_storehouse,&frame->elem);
```

```
    }
}

static struct frame *
vm_get_frame (void) {   // get a new frame and its corresponding structure
    void* kva = palloc_get_page(PAL_USER);
... // palloc_get_page를 통해 얻은 kva를 가진 frame structure를 malloc해서 반환한다.
palloc_get_page를 실패하는 경우 evict_frame을 실행하지만 위의  init단계에서는 관계없다.
```

한 사이클 내에서 태양풍으로 인해 최대 14개의 프레임이 손상될 수 있다 따라서, data_backup 함수에서 14번의 루프를 통해 desig_storehouse 리스트의 앞에서 빼낸 프레임에 데이터를 이전한 후, pml4 mapping을 업데이트한다. pml4가 새 프레임과 페이지를 연결해주기 때문에 데이터를 복구하는 과정은 필요하지 않다.

이 디자인의 핵심은 Storehouse frame으로 사용하는 프레임의 위치를 지속적으로 추적하여 매핑 시에 사용하는 것이다. 태양풍에 노출 된 프레임의 데이터는 오염되어 사용하지 않으므로 빈 프레임으로 간주하여 후에 백업할 프레임으로 다시 사용한다. 이 프레임을 desig_storehouse 리스트의 뒤에 push하여 다음 Cycle에 Storehouse frame으로 활용한다.

```
void
data_backup(void){
    for (int i=0; i<14; i++){
        int idx = next_dist_frames[i];
        if (idx != NULL){
            //src_Frame : that will be affected by sun
            struct frame* scr_frame = frame_table[idx];
            //dst_Frame : list_front in storehouse list
            struct frame * dst_frame =
            list_entry(list_pop_front(&desig_storehouse),struct frame, elem);


            //Get dst_frame that won't be affected by sun at next cycle
            //And rearrange the list order if there's a damaging frame ( push back )
            while(!in_arr(frame_to_idx(dst_frame),nex_dist_frames)){
                list_elem* next_frame_elem = dst_frame->elem.next;
                list_remove(&dst_frame->elem);
                list_push_back(&desig_storehouse, &dst_frame->elem);
                dst_frame = list_entry(next_frame_elem, struct frame, elem);
            }

            //copy data
            move_data_f2f(src_frame , dst_frame);
```

```
        //Remapping the relationship between page and frame
        struct page* page = src_frame->page;
        dst_frame->page = page;
        page->frame = dst_frame;
        //page table setting
        pml4_clear(src_frame->thread->pml4, page->va);
        pml4_set_page (t->pml4, page->va, dst_frame->kva, page->writable)

        src_frame->page=NULL;
        src_frame->thread=NULL;

        //to recycle the damaged frame, push back to the back.
        //The max_num of damaged frame is 14, so it cannot be reached in the current cycle.
        //we can recycle in next cycle.
        list_push_back(&desig_storehouse,&src_frame.elem);
      }
    }
}
```

위의 함수를 매 Cycle마다 호출한다.

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args) {
...
    if(ticks+offset%30==0){
        cycle_update_global_variables(ticks);
        data_backup();
    }
```

이 디자인에서는 pml4를 업데이트하여 스레드가 사용 가능한 프레임만을 점유하기 때문에, 스레드의 실행을 차단시킬 필요가 없다. 따라서, 기존에 struct thread 내에 정의하였던 is_sun을 제거하고, timer_interrupt 함수에서 스레드 차단 관련 코드를 제거하였다.

Validation & Discussion

우선순위별 평가 기준에 따른 Remapping의 평가는 다음과 같다.

1. 해누리호 폭발 방지
다음 Cycle에 태양풍의 영향을 받을 것으로 예상되는 프레임 대신 새 프레임과 유저 페이지와 연결 관계를 만들어 pml4를 업데이트해 준다. 이후부터는 유저 스레드가 기존 프레임에 접근하게 되는 경우가 발생하지 않기 때문에, 해누리호의 폭발은 완벽히 방지된다.

2. 데이터 손실 방지
태양풍으로 인해 손실될 것으로 예정되는 데이터를 사전에 빈 프레임으로 옮겨 주어, 새 프레임과 유저 스레드의 페이지를 연결시켜 주기 때문에, 데이터는 온전히 보존된다.

3. 성능 저하 최소화
데이터의 손실 방지를 위해 데이터를 빈 프레임으로 옮겨줄 때, 이 과정에서 디스크 참조가 발생하지 않는다. RAM 영역에 항상 사용하지 않는 28개 프레임이 존재하기 때문에, 데이터의 보존 작업에 swap-in / out이 발생하지 않으므로, 디스크 접근으로 인한 성능 저하가 발생하지 않는다**.**

또한, 이전 디자인들과 다르게 Remapping에서는 손상될 예정이거나 손상 중인 프레임을 점유하는 유저 스레드의 실행 자체를 차단시키지 않는다. 유저 스레드는 이 경우 새 안전한 프레임을 할당받아 사용하기 때문에 계속 실행 가능하며, 따라서 유저 스레드에 차단으로 인한 제한이 발생하지 않는다. 이전에 유저 스레드 차단으로 인한 큰 성능 저하도 Remapping에서는 발생하지 않아, deadlock과 같은 상황을 초래하지 않는다.

Remapping 디자인 최우선순위에 해당하는 해누리호 폭발 방지와 데이터 손실 방지를 성공적으로 이뤄내었으며, 성능 문제에서도 디스크에 접근하지 않고, RAM 내에서 데이터를 이전하기 때문에 해누리호의 안정성 보장을 위해 많은 리소스가 소요되지 않는다. 또한, 유저 스레드를 차단시키지 않는 방법을 통해 성능 저하가 일어나는 스레드 차단으로 인한 deadlock과 같은 상황을 발생시키지 않아 성능 개선도 이뤄내었다.

다만, 이 디자인에서는 28개 프레임 영역을 기존 목적이 아닌 특수한 '데이터 이전'의 용도로만 사용한다. 따라서 해누리호의 탐사 지속에 따라 Workload가 증가하였을 때, 해당 프레임 영역을 사용하지 못함에 따라 **Page-fault**가 더 자주 발생하는 상황을 초래할 수 있으며, Working set 사이즈를 줄일 수밖에 없다. 이는 곧 실행 가능한 프로세스의 감소로 이어져 성능 감소를 초래한다. 이러한 상황은 2000개 가량의 많은 페이지를 동시에 자주 사용하는 상황에서 발생하기 때문에, 해누리호가 여러가지 작업을 동시에 진행해야 하는 특수한 상황에서 성능 저하가 발생할 수 있다. 예를 들어, 기본적으로 항상 실행중인 Malfunction 탐지, 안전을 위한 관측 시행, 통신망 유지와 더불어 태양 관측, 데이터 송신, 해누리호 자세 조정 등 여러가지 일을 동시에 처리하게 된다면 더 많은 페이지를 지속적으로 자주 사용하게 되고, 이는 더 넓은 메모리 영역을 필요로 해 Page-fault로 인한 성능 저하를 초래할 수 있다.

이렇게 해누리호가 충돌하기 이전의 OS보다 성능이 더 저하되는 경우는, 이렇게 많은 유저 프로그램 동시 실행으로 인해 1972개 이상의 페이지를 자주 사용하는 경우에 발생한다. 이 문제점은 기존처럼 "시간"이라는 리소스 대신에 "메모리 영역"이라는 리소스에 대한 논의를 요구한다. 달성해야 하는 최우선 과제인 폭발 방지와 데이터 보존과 더불어 디스크

참조시간 최소화 등과 함께 어떻게 이 메모리 리소스를 절약할 수 있을지에 대한 추가적인 논의가 필요하다.

Conclusion

해누리호에 발생한 3가지 문제 상황을 해결하기 위해 4가지 디자인을 구상하였다. 각각의 디자인이 가지는 특징과 장, 단점은 다음과 같다.

Cohort Protection

태양풍의 영향을 받는 프레임 중 사용 중인 프레임의 데이터를 모두 디스크에 백업하고, 해당 프레임을 사용하고 있는 스레드의 실행을 차단한다. 영향이 끝나면 데이터를 복구시키고, 스레드의 실행을 허용한다.

장점: 해누리호의 안전 확보, 데이터 손실 방지
단점: 프레임의 사용비율에 비례하여 성능 하락, 유저 스레드 차단으로 성능 저하

Unemployed Frame Recruit

사용하지 않는 프레임이 있을 경우 이 프레임에 데이터를 백업 후 복구한다. 해당 프레임을 사용하는 스레드의 실행은 마찬가지로 차단된다. 사용하지 않는 프레임이 부족할 경우 디스크에 데이터를 백업한다.

장점: 해누리호의 안전 확보, 데이터 손실 방지
단점: 대부분의 프레임이 사용중일 경우 성능 대폭 하락, 유저 스레드 차단으로 성능 저하

Memory Storehouse

28개 프레임을 Storehouse frame으로 지정하여 페이지에 할당하지 않고, 태양풍의 영향을 받는 프레임의 데이터 백업 저장소로 사용한다. 해당 프레임 사용하는 스레드의 실행은 차단된다.

장점: 해누리호의 안전 확보, 데이터 손실 방지, 데이터 백업 위한 디스크 접근 방지
단점: 유저 스레드 차단으로 성능 저하, 사용 가능한 메모리 감소로 Page fault 증가

Remapping

Storehouse frame으로 28개 프레임을 유동적으로 지정하여 태양풍의 영향을 받는 프레임의 데이터를 이전하고, 새로운 프레임과 페이지를 연결하여 pml4를 업데이트함. 태양풍으로 인해 스레드의 실행이 차단되지 않음.

장점: 해누리호의 안전 확보, 데이터 손실 방지, 데이터 백업 위한 디스크 접근 방지, 태양풍 상황에도 스레드 실행 보장
단점: 사용 가능한 메모리 감소로 Page fault 증가

논의를 지속함에 따라 디자인 iteration을 반복하며 점차 개선된 디자인을 선보일 수 있었다. 첫 디자인인 Cohort Protection은 최우선순위 디자인 목표인 해누리호 폭발 방지와 데이터 보존의 달성에 초점을 맞춰 디자인하였다. 이후로는 이전 디자인이 가지는 성능 한계를 개선하기 위한 디자인을 구상하였으며, 이의 성능을 안정성 보장을 위한 리소스 사용 과다 여부, 성능 저하를 유발하는 특정 상황 존재 여부, 유저 프로그램의 동작에 큰 제한 존재 여부라는 기준을 통해 평가하였다. 이에 따라 “시간”이라는 리소스의 사용 과다를 방지하기 위해 디스크 접근 시간을 최소화시키고, 유저 프로그램의 제한 최소화를 위해 유저 스레드의 차단을 방지하였다.

다만, 이 과정에서 28개의 프레임 영역을 사용하지 못함에 따라 “메모리”라는 리소스 측면에서 비효율이 발생하였다. 사용 가능한 메모리 리소스가 줄어듦에 따라, Page fault의 빈도가 증가할 수 있게 되었다. 이는 애초에 2000개 프레임으로 구성된 해누리호가 평상시 탐사를 진행할 때에는 문제가 되지 않겠지만, 많은 유저 프로그램을 동시에 가동시켜야만 하는 특정 상황에서는 많은 페이지를 사용하여 Locality가 Working set size보다 커지게 되며, 이는 Page fault의 증가를 초래할 수 있다.

이보다 성능이 더 개선된 디자인은 이처럼 지금까지 살펴왔던 “시간”이라는 리소스 외에도 “메모리”측면에서의 효율성 또한 고려해야 한다. 디스크 접근 시간을 발생시키지 않으면서도 메모리 공간 손실이 적은 디자인을 제시하여 해누리호가 더욱 개선된 성능으로 임무를 수행할 수 있도록 추가적인 논의가 필요하다.