**Pintos Project 0-2: Pintos Data Structure**

|  |  |
| --- | --- |
| 담당 교수 : | 문의현 |
| 학번 : | 변성진 |
| 이름 : | 20180546 |
|  |  |

**반드시 아래의 양식과 순서를 따라서 작성하기 바랍니다.**

1. **testlib**

|  |  |
| --- | --- |
| **Prototype** | int main(); |
| **Parameter** |  |
| **Return** | NO RETURN |
| **Function** | 우선 라이브러리를 테스트하기 위한 main function은 all\_list라는 자료구조를 가진다. 해당 자료구조에는 struct\_elem 이라는 자료구조를 list\_elem으로 이어주고 있다    struct\_elem은 create으로 생성된 모든 자료구조(hash,list,bitmap)의 포인터를 저장하고 있다. 또한 자료구조를 create {datastructure} {name} 으로 생성함으로 유일한 id는 생성할때의 {name}이므로, name을 후에 strcmp로 검사하며 자료구조를 찾아올 것이다. 그리고 모든 자료구조마다 dumpdata의 함수를 조금씩 달리 해줘야함으로 enum으로 자료구조의 타입(struct\_type)도 가지고 있을 것이다  우선 main이 처음 실행되면, all\_list를 list\_init한다.  그리고 while 루프를 도는데 quit 명령어가 stdin에 입력될때까지계속 (명령어 파싱->동작 실행) 의 구조다  모든 stdin에 입력되는 input을 분석한 결과 create과 quit을 제외한 모든 명령어는  {structType\_Impl} {name} {…args}의 형태를 가지는데, create은 create {structType} {name}으로 구조체의 이름이 제일 마지막에 온다. 따라서 create만 별도의 예외처리를 해준다    자료구조마다 초기화하는 방법이 달라서 3가지 분기로 처리해줘야한다  나머지 함수들은 모두 파싱한 char\*\* args와 입력받은 name을 바탕으로 struct\_elem의 포인터를 all\_list에서 찾아온 struct\_elem\* s\_elem 두가지를 인자로 받는 wrapper함수로 처리하였다    여기서의 name은 함수의 이름이다    그리고 args[0]을 바탕으로    총 47개의 명령어를 strcmp를 통해 찾으며 함수를 실행한다(여기서 아마 binarysearch을 썼으면 더 빨랐을수도 있겠지만 성능이 중요한게 아니므로 간단히 구현)  그리고 해당 프로젝트는 완결성 있는 프로그램보다 구현된 함수들의 test에 있으므로, stdin에 입력되는 input에 예외처리가 하나도 되어있지않다. \*.in파일과 올바른, 정상적인 명령어만 받음을 가정했으므로 논리적으로 이상한 input을 넣으면 정상작동되지 않을것이다. |

1. **Additional Implementation**

|  |  |
| --- | --- |
| **Prototype** | struct bitmap\* bitmap\_expand(struct bitmap\* bitmap,int size) |
| **Parameter** | Struct bitmap\*, int size |
| **Return** | Struct bitmap\* |
| **Function** | 1. 새로운 사이즈+기존 bitmap의 사이즈만큼의 bitmap을 메모리할당(초기화) 한다, 변수는 ret 2. 기존 입력받은 bitmap의 비트를 모두 새로할당받은 ret에메모리카피(memcpy)한다 3. 기존 bitmap의 주소와 bits를 모두 메모리할당해제한다 4. Ret을 리턴한다. |

|  |  |
| --- | --- |
| **Prototype** | unsigned hash\_int\_2(int key) |
| **Parameter** | Int key |
| **Return** | Unsigned |
| **Function** | <https://github.com/markokr/pghashlib/blob/master/src/inthash.c>의 소스를 참고했다  수학이나 통계의 범주까지 생각하기엔 너무 어려워서, 최대한 난수를 생성하고자 하는 의도로 만들었다.  2^31과의 XOR연산, key와 key/16의 덧셈, 그리고 key와 key\*2^3의 XOR연산, 그리고 무작위한 수와의 XOR연산을 통해 어떠한 난수를 생성했다 |

|  |  |
| --- | --- |
| **Prototype** | void list\_swap(struct list\_elem \*a, struct list\_elem \*b) |
| **Parameter** | Struct list\_elem\*, struct list\_elem\* |
| **Return** | Void |
| **Function** | 우선 바로 떠오르는 방법은  A의 next elem<->B<->A의 prev elem를 우선 연결시켜주고  B의 next elem<->A<->B의 prev elem을 다시 연결시켜주는것이다  Anext,aprev,bnext,bprev의 포인터를 위 과정을 실행하기전 저장해두고 각각 연결시켜주면된다  하지만 a와 b가 바로 인접한 elem일 경우에 위 과정이 어떠한 코드상에 오류로 정상적인 작동을 하지 않는 것을 발견하여    A<->B, B<->A의 경우에만 위와 같이 예외처리를 통해 a를 제거하고 anext(b)의 next, 즉 b의 다음 자리에 insert 하는 방법으로 구현했다 |

|  |  |
| --- | --- |
| **Prototype** | void list\_shuffle(struct list \*list) |
| **Parameter** | Struct list\* |
| **Return** | Void |
| **Function** | List\_size 1의 시행만큼 list front를 list\_pop\_front로 뽑아서 랜덤한 index를 정하고 (rand()%size) 해당 index의 위치만큼 가서 (for loop)  해당 자리에 insert해준다  또한 랜덤한 shuffle 외에 좀더 불규칙한 shuffle을 실행하기 위해 위 과정이 끝난 후 마지막 원소를 뽑아서 list\_push\_front한다 |

1. **List**

|  |  |
| --- | --- |
| **Prototype** | void list\_init (struct list \*); |
| **Parameter** | Struct list\* list |
| **Return** | Void |
| **Function** | Pintos의 리스트는 double linked list 구조로 되어있다  처음 init을 하면 head와 tail이 위와 같은 구조로 초기화되고    List\_elem을 insert하면 위와 같은 구조로 head와 tail사이에 list\_elem이 doubly linked 된다. |

|  |  |
| --- | --- |
| **Prototype** | size\_t list\_size(struct list \*list) |
| **Parameter** | struct list\* |
| **Return** | Size\_t |
| **Function** | 해당 함수를 통해 list\_begin, list\_end, list\_next를 모두 이해할 수 있다  struct list\_elem \* list\_next(struct list\_elem \*elem)  은 list포인터를 인자로 받아 head의 next, 즉 첫번째 elem 포인터를 반환한다  struct list\_elem \* list\_end(struct list \*list)  list\_end는 list의 tail 포인터를 반환한다  struct list\_elem \* list\_next(struct list\_elem \*elem)  list\_next는 list\_elem 포인터를 인자로 받아 다음 elem의 포인터를 반환한다  list\_Size에서는 3가지 함수를 통해 list전체를 iterate하며 count를 증가시키며 전체 원소의 개수를 구하고 있다 |

|  |  |
| --- | --- |
| **Prototype** | *#define* list\_entry(LIST\_ELEM, STRUCT, MEMBER) |
| **Parameter** | LIST\_ELEM , STRUCT ,MEMBER |
| **Return** | struct STRUCT \* |
| **Function** | List\_entry함수는 실제 리눅스에서도 container\_of 로 사용되는 굉장히 강력한 매크로 함수다    Struct의 메모리 순서는 런타임에 동적으로 정의되는 것이 아니라 컴파일타임에 정의되기 때문에 offsetof 매크로와 결합하여 struct의 시작 주소를 반환할 수 있다  List\_elem->next는 곧 list\_elem의 시작주소다 (next, prev 순으로 정렬)  예를 들어 struct thread 에 저장된 list\_elem을 통해 struct thread의 시작주소를 구한다고 하자  *#define* offsetof(s,m) ((size\_t)&(((s\*)0)->m))  Offsetof 매크로를 뜯어보면 위와 같이 0x0에서 struct thread이 시작함을 가정하고 , struct thread에서 list\_elem이 시작되는 곳의 주소값 (예를들면 0x40)을 반환한다.  만약 원래 list\_elem이 0Xffffff60에 있다고 하면, struct thread의 시작 주소는 0Xffffff20에 위치하는 것이다(60-40)  List\_begin, list\_next,list\_end와 함께 사용하면    위와 같이 list를 iterating 하며 struct thread의 member variable list\_elem을 통해 struct thread의 포인터를 구해서 어떠한 procedure을 실행할 수 있다 |

|  |  |
| --- | --- |
| **Prototype** | void list\_splice(struct list\_elem \*before,                   struct list\_elem \*first, struct list\_elem \*last) |
| **Parameter** | Struct list\_elem\*, struct list\_elem\*, struct list\_elem\* |
| **Return** | Void |
| **Function** | Before의 다음 위치에 first~last까지의 list\_elem 을 넣는다  이때 first~last는 같은 리스트의 원소이어야 한다. 실행 후 first~last의 원 리스트에서 first~last는 제거된다 |

|  |  |
| --- | --- |
| **Prototype** | void list\_unique(struct list \*list, struct list \*duplicates,                   list\_less\_func \*less, void \*aux) |
| **Parameter** | struct list\* , struct list\*, list\_less\_func\*, void\* |
| **Return** | Void |
| **Function** | 만약 struct list\* list의 중복된 원소를 모두 제거한다. 만약 duplicates의 포인터가 주어지면 중복된 원소는 모두 duplicates에 삽입된다 |

1. **Hash Table**

|  |  |
| --- | --- |
| **Prototype** | Bool  hash\_init (struct hash \*h,             hash\_hash\_func \*hash, hash\_less\_func \*less, void \*aux) |
| **Parameter** | Struct hash\*, hash\_hash\_func\*, hash\_less\_func\*, void\* |
| **Return** | Bool |
| **Function** | Hash init을 실행하기전 h에 할당된 포인터가 있어야한다  Hash\_hash\_func은 실제 해싱을 실행하는 함수로 본 과제에선 hash\_int가 들어가면 된다. Hash\_less는 hash 리스트를 오름차순으로 정렬해주는 compare 함수이다.  Struct hash의 멤버변수로는  Struct list\* 로 표현된 hash bucket이 있고, 초기 bucket의 개수 bucket\_cnt는 4로 초기화된다 |

|  |  |
| --- | --- |
| **Prototype** | unsigned  hash\_int (int i) |
| **Parameter** | Int |
| **Return** | Unsigned |
| **Function** | hash int 는 integer를 해싱하는 해시함수로 hash\_bytes(&i,4)를 리턴하는데    위의 hash\_byte 한수를 살펴보면 while 루프를 돌며 무작위 32\_PRIME 과 MULTIPLY,그리고 XOR연산을 한다 |

|  |  |
| --- | --- |
| **Prototype** | static struct list \*  find\_bucket (struct hash \*h, struct hash\_elem \*e) |
| **Parameter** | Struct hash\*, struct hash\_elem\* |
| **Return** | Struct list\* |
| **Function** | size\_t bucket\_idx = h->hash (e, h->aux) & (h->bucket\_cnt - 1);  *return* &h->buckets[bucket\_idx];   1. 먼저 주어진 hash\_elem을 해싱한 값과 bucket\_cnt와 and연산을 취한다 2. List\*의 array buckets에 idx를 취한값을 return 한다 |

|  |  |
| --- | --- |
| **Prototype** | static struct hash\_elem \*  find\_elem (struct hash \*h, struct list \*bucket, struct hash\_elem \*e) |
| **Parameter** | Struct hash\*, struct list\*, struct list\_elem\* |
| **Return** | Struct hash\_elem\* |
| **Function** | *if* (!h->less (hi, e, h->aux) && !h->less (e, hi, h->aux))  *return* hi;  하나의 bucket, list에서 iterating 하며 less연산이 왼쪽/오른쪽 elem 모두 false인, 즉 왼쪽과 오른쪽 모두 크지도 작지도않은, 다시말해 값이 같은 원소를 반환한다  만약 찾지 못하면 NULL 포인터를 반환한다 |

|  |  |
| --- | --- |
| **Prototype** | struct hash\_elem \*  hash\_find (struct hash \*h, struct hash\_elem \*e) |
| **Parameter** | struct hash\*, struct hash\_elem\* |
| **Return** | Struct hash\_elem |
| **Function** | *return* find\_elem (h, find\_bucket (h, e), e);  위에서 살펴봤던 find\_bucket으로 우선 bucket을 찾는다.  그후 해당 bucket을 파라미터로 find\_elem으로 elem이 존재하는지/존재하지 않는다면 NULL포인터를 반환한다 |

|  |  |
| --- | --- |
| **Prototype** | static void  rehash (struct hash \*h) |
| **Parameter** | Struct hash\* |
| **Return** | Void |
| **Function** | Hash\_elem의 개수에 따라 bucket 개수를 power of 2개로 조정하여 좀더 hash table에서 elem을 찾기 쉽게 재배치한다   1. 재배치할 hash table의 bucket 개수를 정하고 메모리 할당 2. 기존 hash bucket들을 모두 iterate하며+제거 새로운 hash bucket에 insert 3. 기존 hash table에 삽입 |

1. **Bitmap**

|  |  |
| --- | --- |
| **Prototype** | struct bitmap \*  bitmap\_create(size\_t bit\_cnt) |
| **Parameter** | Size\_t |
| **Return** | Struct bitmap \* |
| **Function** | Struct bitmap 메모리 할당  Bitmap 내부의 bits(실제 bitmap data) 메모리 할당  모든 bitmap을 0으로 initialize 후 리턴 |

|  |  |
| --- | --- |
| **Prototype** | void bitmap\_set(struct bitmap \*b, size\_t idx, bool value) |
| **Parameter** | Struct bitmap\*, size\_t idx, bool value |
| **Return** | Void |
| **Function** | Bitmap의 idx에 value(true or false)의 값을 set 하는 함수다  Bitmap\_mark, bitmap\_reset을 호출한다 |

|  |  |
| --- | --- |
| **Prototype** | static inline elem\_type  bit\_mask(size\_t bit\_idx) |
| **Parameter** | Bit\_idx |
| **Return** | Elem\_type (unsigned long) |
| **Function** | *return* (elem\_type)1 << (bit\_idx % ELEM\_BITS);  1을 bit\_idx에 elem\_bits(64)를 모듈러 연산한 값으로 쉬프트한다  Struct bitmap의 단위가 8byte, 64비트이기 때문에  69를 마스크한다고 하면 배열의 두번째에 64 모듈러 연산을 취한  5번째 bit가 1로 set 된다 |

|  |  |
| --- | --- |
| **Prototype** | void bitmap\_mark(struct bitmap \*b, size\_t bit\_idx)  void bitmap\_reset(struct bitmap \*b, size\_t bit\_idx)  void bitmap\_flip(struct bitmap \*b, size\_t bit\_idx) |
| **Parameter** | Struct bitmap\*, size\_t |
| **Return** | Void |
| **Function** | size\_t idx = elem\_idx(bit\_idx);    elem\_type mask = bit\_mask(bit\_idx);  elem\_idx는 위 bit\_mask에서 설명했던것처럼, 만약 bit\_idx가 69라면 69/64=1, 1이 반환된다  그리고 inline assembly가 실행되는데    asm("orl %k1, %k0"        : "=m"(b->bits[idx])        : "r"(mask)        : "cc");  %k0에는 b->bits[idx]가  %k1에는 mask가 들어간다. Or 연산을 통해 00100 | (어떤 bitmap)을 통해 e.g 3번째 bit를 set한다    asm("andl %k1, %k0"        : "=m"(b->bits[idx])        : "r"(~mask)        : "cc");  위는 bitmap\_reset의 inline assembly인데 or 대신 and 연산이 사용되었다    asm("xorl %k1, %k0"        : "=m"(b->bits[idx])        : "r"(mask)        : "cc");  위는 bitmap\_flip으로 XOR 연산을 사용한다  0010110101에서 6번째 원소를 FLIP 한다고 하자  0000010000  0010100101 으로 XOR 연산결과가 bit->bits\_idx에 저장된다. |

|  |  |
| --- | --- |
| **Prototype** | bool bitmap\_test(const struct bitmap \*b, size\_t idx) |
| **Parameter** | Const struct bitmap\*, size\_t |
| **Return** | bool |
| **Function** | *return* (b->bits[elem\_idx(idx)] & bit\_mask(idx)) != 0;  bits 에서 idx를 64로 나눈 elem\_idx, modular 연산을 한 bit\_mask와의 AND연산결과가 1이면 TRUE, 0이면 false를 반환한다 |

|  |  |
| --- | --- |
| **Prototype** | bool bitmap\_contains(const struct bitmap \*b, size\_t start, size\_t cnt, bool value) |
| **Parameter** | Const struct bitmap\*, size\_t,size\_t,bool |
| **Return** | Bool |
| **Function** | bitmap에서 start에서 start+cnt까지 value(true or false)가 하나라도 포함되어있으면 true, 아니면 false를 반환한다 |

|  |  |
| --- | --- |
| **Prototype** | size\_t  bitmap\_scan(const struct bitmap \*b, size\_t start, size\_t cnt, bool value) |
| **Parameter** | Const struct bitmap\*, size\_t, size\_t, bool |
| **Return** | Size\_t |
| **Function** | [start,start+cnt]가 모두 value값인 bitmap에서 첫번째 idx를 찾는다  1000010111000001  Start=0 , cnt=5라면 00000이 연속으로 있는 첫번째 위치인 10이 반환된다 |