Disk Manager 분석 1주차 질문

1. 범용 볼륨이 데이터 볼륨과 인덱스 볼륨의 역할을 함께 수행할 수 있는 것인지 궁금합니다.
2. `storage/disk_manager.c`의 `disk_manager_init` 함수에서 가장 처음 부분에 제시된 시스템 파라미터 값인 `PRM_ID_BOSR_MAXTMP_PAGES`의 `BOSR`의 의미가 `Boot Management at Server`인 것 같은데, `Boot Management`라고 하는 것이 어떤 것인지 간략한 설명이 듣고 싶습니다!
3. `PRM_ID_BOSR_MAXTMP_PAGES` 열거 값으로 초기화 되어 `sector` 당 `page` 수로 나누어 이용되는 `disk_Temp_max_sects` 변수의 역할이 궁금합니다.

```
4937    disk_Temp_max_sects = (DKNSECTS) prm_get_integer_value (PRM_ID_BOSR_MAXTMP_PAGES);
4938    if (disk_Temp_max_sects < 0)
4939      {
4940        disk_Temp_max_sects = SECTID_MAX; /* infinite */
4941      }
4942    else
4943      {
4944        disk_Temp_max_sects = disk_Temp_max_sects / DISK_SECTOR_NPAGES;
4945      }
4946
```

4. `disk_manager_init` 함수이 불가능하여 `disk_manager_final` 함수로 분기하는 부분에 보면, 단순히 `disk_cache_final` 함수를 호출해주는 것으로 보았습니다. 이 때 만일 `SERVER_MODE`라면 `데몬`도 종료하는 것으로 확인되는데, `SERVER_MODE`의 의미와 이 때의 `데몬`이 무엇인지 궁금합니다.

```
4976  /*
4977   * disk_manager_final () - free disk manager resources
4978   */
4979  void
4980  disk_manager_final (void)
4981  {
4982  #if defined (SERVER_MODE)
4983    disk_auto_volume_expansion_daemon_destroy ();
4984  #endif /* SERVER_MODE */
4985
4986    disk_cache_final ();
4987  }
4988
```

5. `fileio_map_mounted` 함수로 `disk_cache_load_volume` 함수를 내부적으로 호출하는 부분이 영구 볼륨과 일시 볼륨 2개로 반복문이 나뉘는 것을 볼 수 있었습니다. 영구 볼륨은 증가 방향, 일시 볼륨은 감소 방향으로 반복이 되던데, 서로 반복 방향이 다른 이유와 이것이 의미하는 바가 무엇인지 궁금합니다. 그리고 `FILEIO_VOLINFO_INCREMENT`라는 매크로 상수 값이 왜 `32`로 되어 있는지도 궁금합니다.

```
3453    * fileio_map_mounted () - Map over the data volumes
3454    *   return:
3455    *   fun(in): Function to call on volid and args
3456    *   args(in): argumemts for fun
3457    *
3458    * Note : Map over all data volumes (i.e., the log volumes are skipped),
3459    *        by calling the given function on every volume. If the function
3460    *        returns false the mapping is stopped.
3461    */
3462   bool
3463   fileio_map_mounted (THREAD_ENTRY * thread_p, bool (*fun) (THREAD_ENTRY * thread_p, VOLID vol_id, void *args),
3464                       void *args)
3465   {
3466     FILEIO_VOLUME_INFO *vol_info_p;
3467     FILEIO_VOLUME_HEADER *header_p;
3468     int i, j, max_j, min_j, num_temp_vols;
3469
3470     FILEIO_CHECK_AND_INITIALIZE_VOLUME_HEADER_CACHE (false);
3471
3472     header_p = &fileio_Vol_info_header;
3473     for (i = 0; i <= (header_p->next_perm_volid - 1) / FILEIO_VOLINFO_INCREMENT; i++)
3474       {
3475         max_j = fileio_max_permanent_volumes (i, header_p->next_perm_volid);
3476
3477         for (j = 0; j <= max_j; j++)
3478   {
3479     vol_info_p = &header_p->volinfo[i][j];
3480     if (vol_info_p->vdes != NULL_VOLDES)
3481       {
3482         if (((*fun) (thread_p, vol_info_p->volid, args)) == false)
3483   {
3484     return false;
3485   }
3486       }
3487   }
3488       }
3489
3490     num_temp_vols = LOG_MAX_DBVOLID - header_p->next_temp_volid;
3491     for (i = header_p->num_volinfo_array - 1;
3492          i > (header_p->num_volinfo_array - 1
3493         - (num_temp_vols + FILEIO_VOLINFO_INCREMENT - 1) / FILEIO_VOLINFO_INCREMENT); i--)
3494       {
3495         min_j = fileio_min_temporary_volumes (i, num_temp_vols, header_p->num_volinfo_array);
3496
3497         for (j = FILEIO_VOLINFO_INCREMENT - 1; j >= min_j; j--)
3498   {
3499     vol_info_p = &header_p->volinfo[i][j];
3500     if (vol_info_p->vdes != NULL_VOLDES)
3501       {
3502         if (((*fun) (thread_p, vol_info_p->volid, args)) == false)
3503   {
3504     return false;
3505   }
3506       }
3507   }
3508       }
3509
3510     return true;
3511   }
```

6. `fileio_map_mounted` 함수 내에서 `FILEIO_CHECK_AND_INITIALIZE_VOLUME_HEADER_CACHE`매크로 함수 사용 부분을 보면 `fileio_Vol_info_header` 라는 전역 변수를 이용하는 것을 볼 수 있었습니다. 해당 변수는 내부의 `volinfo`라는 필드를 `storage/file_io.c`의 `922` 라인 함수에서 `FILEIO_VOLUME_INFO*` 크기 `n`개를 할당 받아서 운용되는 것을 볼 수 있었는데요. 이 때 `n`을 계산해보니 `1024`라는 값을 얻을 수 있었습니다. `fileio_Vol_info_header.volinfo`가 왜 2차원 배열인지, 그리고 그 크기를 `1024`로 두게 되는 이유가 궁금합니다.

```
921  static int
922  fileio_initialize_volume_info_cache (void)
923  {
924    int i, n;
925    int rv;
926
927    rv = pthread_mutex_lock (&fileio_Vol_info_header.mutex);
928
929    if (fileio_Vol_info_header.volinfo == NULL)
930      {
931        n = (VOLID_MAX - 1) / FILEIO_VOLINFO_INCREMENT + 1;
932        fileio_Vol_info_header.volinfo = (FILEIO_VOLUME_INFO **) malloc (sizeof (FILEIO_VOLUME_INFO *) * n);
933        if (fileio_Vol_info_header.volinfo == NULL)
934    {
935      er_set (ER_ERROR_SEVERITY, ARG_FILE_LINE, ER_OUT_OF_VIRTUAL_MEMORY, 1, sizeof (FILEIO_VOLUME_INFO *) * n);
936      pthread_mutex_unlock (&fileio_Vol_info_header.mutex);
937      return -1;
938    }
939        fileio_Vol_info_header.num_volinfo_array = n;
940
941        for (i = 0; i < fileio_Vol_info_header.num_volinfo_array; i++)
942    {
943      fileio_Vol_info_header.volinfo[i] = NULL;
944    }
945      }
946
947    pthread_mutex_unlock (&fileio_Vol_info_header.mutex);
948    return 0;
949  }
```

7. `disk_log` 함수 → `_er_log_debug` 함수 → `LOG_THREAD_TRAN_ARGS` 매크로 함수 → `LOG_FIND_CURRENT_TDES` 함수 → `LOG_FIND_TDES` 함수를 타고 가면서 생긴 궁금증입니다.

   a. `LOG_TRHEAD_TRAN_ARGS`는 매크로 함수로 작성되어 있는데, `LOG_FIND_CURRENT_TDES`, `LOG_FIND_TDES` 함수는 일반 함수인 것으로 확인했습니다. 이 때 함수 네이밍 컨벤션이 다른 함수들과 달리 대문자로 이뤄져 매크로 함수 컨벤션처럼 작성된 이유가 있는지 궁금합니다.

   b. `TDES`라는 것이 트랜잭션 디스크립터라고 이해를 했습니다. 그리고 `transaction/log_impl.h`의 `1298` 라인에서 `log_Gl`이라는 전역 변수의 `Trantable` 구조체, 그리고 그 안에 `all_tdes`를 유지하고 있는 것을 확인할 수 있었는데요. 이 때 트랜잭션 당 여러 디스크립터가 있는 이유가 무엇이고, 왜 `all_tdes`는 더블 포인터로 되어 있는지 궁금합니다. (`all_tdes`로 사용된 `log_tdes` 구조체가 (동일 파일 내 `513` 라인) `MVCC`와 관련 있어 보이는데, 더블 포인터로 둔 이유가 `MVCC` 상의 버전 때문에 그런 것인가요?)

   c. `LOG_FIND_TDES` 함수에서 `tran_index` 매개 변수가 유효한 인덱스 범위인지 확인 후, `LOG_SYSTEM_TRAN_INDEX`라는 매크로 상수 `0`과 동일하면 `logtb_get_system_tdes`를 호출하는 것을 확인했습니다. 그리고 `log_tb_get_system_tdes` 함수 내에선 시스템 사용자인지 확인을 하는 과정을 통해 `log_Gl.trantable.all_tdes`인 `LOG_TDES*`를 반환할지, 쓰레드 엔트리의 `get_system_tdes()→get_tdes()`인 `LOG_TDES*`를 반환할지가 달라지는 것을 볼 수 있었습니다. 전자와 후자의 `LOG_TDES*`가 어떤 차이가 있는 것인지 궁금합니다.

```
6028  LOG_TDES *
6029  logtb_get_system_tdes (THREAD_ENTRY * thread_p)
6030  {
6031    if (thread_p == NULL)
6032      {
6033        thread_p = thread_get_thread_entry_info ();
6034      }
6035    // if requesting system tran_index and this is a system worker, return its own log_tdes
6036    if (thread_p->tran_index == LOG_SYSTEM_TRAN_INDEX && thread_p->get_system_tdes () != NULL)
6037      {
6038        return thread_p->get_system_tdes ()->get_tdes ();
6039      }
6040    else
6041      {
6042        return log_Gl.trantable.all_tdes[LOG_SYSTEM_TRAN_INDEX];
6043      }
6044  }
```