Project 3 - Design Report (Team 28)

□□ Project 3□□□ □□□ 7□□ □□□□□ □□□□□ □□.

Frame Table

Basic Descriptions

Limitations and Necessity

Design Proposal

Supplemental Page Table

□□□ □□□□ CPU□ □□□ □□ □□□ □□ □□□ □□□ □□□□ □ □□□ □□□ □□□, □□□ □□□□ □□□ □□□ □□ □□□□ □□□□ □□□□ □□□□ □□□□□ □□□ □□□ □□□□ □□ □□□□ □□ □□□□ □□□ □□ □□□ □□□□ □□ □□ □□□ □□ □□□ □□□ □□ □□□ □□□□□ □ □□□ □□□□. □□□ □□ □□□, Pintos □□□□□ □□□ □□□□ □□□ □□□□□□□ □□□ □□□□ □□(supplement)□□ □□□□□ □□ □□□ □□□(Supplemental Page Table; □□ SPT□ □□)□□□ □□□□□ □□□□.

Basic Descriptions

SPT□ □□□ □□ □□□ □□ □□□ □□□ □□□□□□ □□□□ □□□ □□□ □ □□□ □□□□□□. □□□, □□□ □□□□ □□□ □□□ □□ □□□□ □□□ □□□□ □□□ □ □□ □□ □□□ □□□□□ □□ □□, SPT□ □□□ □□□ □□ □□ □□□ □□□□□□ □□□□ □□. □, SPT□ □□□□□ □ □(specification)□ □□□ □□□ □□□□ □□□□ □□□ □ □□ □□□ □□□ □□□□ □□ □□□□□□.

□□□□, □ SPT□ □□(entry, □□ SPTE□ □□)□□ □□□ □□□ □□□□ □□? Pintos □□ □□□ 'Managing the Supplemental Page Table' □□□ □□, □□□ □□□ □□□□ □□□□□ SPT□ □□□ □□ □□□ □□□ □□□□□ □□□ □□□ □□□ □ □□ □□ □□□□□(evicted) □□ □□□□□□ □□(load)□□ □□ □□□ □□□ □□□□□, □□□ □□□ □□ □□□ □□□ □□□□ □□ □□□ □ □□ □□□□□□ □□□□. □, □ SPTE□ □ □□□□ □□□ □□ □□ □□□□ □□ □□□ □□ □□□ □□. □□ □□ □□ □□□ □□ SPTE□ □□□ □□ □□ □□□□ □□ □□□□□ □□(context) □□□ □□□□ □□□□□ □□□□□□ □□ □□□□.

□□□, SPT□ □□□ □□ □□□□ □□ □□□□ □□□□□ □□ □□□□□ □□□ □□□ □□□□ □□ □□ □□□ □□□ □□ □□□. □□ □□ □□□□ □ □ □□□□□ □□, □□□ □□□□□ □□ □□□□ □□□ □□□□ □□ □□□ SPTE□ □□□ □□□ □□. □□ □□ `devices/block.c`□ `block_read()` □□□ □□□ □□□□.

```
/* From devices/block.c. */
/* Reads sector SECTOR from BLOCK into BUFFER, which must
   have room for BLOCK_SECTOR_SIZE bytes.
   Internally synchronizes accesses to block devices, so external
   per-block device locking is unneeded. */
void
block_read (struct block *block, block_sector_t sector, void *buffer)
{
  check_sector (block, sector);
  block->ops->read (block->aux, sector, buffer);
  block->read_cnt++;
}
```

□□ □□□□ □□□□ □□□□□, □□□□□ □□□□ □□ □□ □□□□ □□ □□□□ □□□□□ □□ □□□□ □□□□ □□□□□ □□ □□□□ □□ □□□□ □□□□□ □ □ □□. □□ □ □□ □□□□□ □□□□□□ □□□□□ □ □□□□□ □□□□ □□□□ □□□□ □□□□ 4 KiB□ □□□□ □□, □□□□ □□□□□ □□ □□ □□ □□ □□□□ □□ □□□□ □ □□ □□□□ □□□□□□ □□□□□ □□□□ □□□□□ □□ □□□□□ □□ □□□□□ □□ □□ □□ □□ □□□□□ □□ □□□□ □□□□□ □□ □□ □□□□□ □ □□□□ □ □ □ □□□□. □□, `block_read()`□ □□□□ □□□□ □□□□□ □□□□ □□□□□□, □□□□ □□□□□ □□ □□ □□□□□ □□□□ □□□□□□ □□□□ □□ □□ □□ □ □□□□ □□ □□□□, □□ □□ □□□□□ □□ □□□□ □□□□□□ □□□□ □□ □□□□ □□□□ □□□□ □□ □□□□□ □□ □□□□□ □□□□ □ □ □□□□ □□□□□□□ □□□□□ □□□□ □□□□ □□ □□□□ □□ SPTE□ □ □□□□ □□□□ □□□□.

Limitations and Necessity

```
/* From userprog/exception.c. */
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  ...

  /* If it is caused by invalid address passed to the kernel, kill user
process
     while making no harm to the kernel. */
  if (is_user_vaddr (fault_addr) && !user)
    process_exit (-1);

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

`userprog/exception.c`□ □□□ □□ □□□ □□□ □□□□□, □□□ Pintos □□□ SPT□ SPTE□ □□□□ □□□ □□, □□□ □□□ □□□ □□□ □□□ □□□ □ □□ □□ □□□ □□□□□ □□□ □□□□ □□ □□□□□ □□□□□ □□. □, □□□ □□□ □□ □□□□ □□□ □□□□□ □ □ □□□ □□□ □□□□□ □□□□ □□□□□ □□□ □□ □□□□, □□ □□□□ □□□□ □□□ □□□□□ □□□□□ □□□□ □□□□ □□. □□□, □□ □□□ □□□ □□(lazy loading)□ □□□ □□ □□(memory mapped file), □□□ □□ □□ □□□□ □□□ □□□ □ □□ □□□□ □□ □. □□ □□□ □□□□□ □□□ □□□ □□□ □□□ □□□□ □□ □□□ □□□ □□□ □□□□ □□□ □□□ □□□□ □ □□.

SPT□ □□ □□□□, □□□ SPT□ □□□ □□ □□□□ □□□□ □□□ □□□□□ □□□ □□□ □□□□, □□□ □□□ □□□□□ □□ □□□□ □□ □□ □□ □□ □□□ □□□□ □□□□□ □□□, □□□ □□□□□ □□□□ □□ □ □□ □□□ □□ □□□ □□□ □□ □□ □□□□□ □□ □□□ □□□□ □□□ □ □□□□□ □□□ □□□ □□□ □□□□ □□□ □ □□ □□□.

□ □□ □□□ □□□ □□□□□ □□□□□, □□□ □□ □□□□ □□ □□□□ □□□□□ □□□ □□ □ □□□□□ □□□ □□□□□□□□□□ □□ □□□ □ □. □□□□□, SPT□ □ □□□□□ □□□ □□□□□□ □□□, □□□□ □ □□□□□ □□ □□□ □□□□ □□ □□□ □□ □□ □□□ □□ □□□□ □□□ □□□, □□ □□ □□□□ □□ □□□□ □□□ □□ □□□□.

Design Proposal

□□□□, □□□ SPT□ □□□□ □□ □□□ □□□□□ □□□□? SPT□ □□□□□ □□□ □□□□ □□□ □□□ □□ □□□□ □□ □□□□ □□□ □ □ □. □□□ □□□ □□□□ □□□□ □□□□□□ □□ □□□ □□□ □□ □□□ □□ □□□□ □□□ □□□, SPT □□□ □□□ SPTE□ □□□ 2^20□ □□, (□□□□ □□□ □□□ □□□ □□□ 20 bit□□□) □□ SPTE □ □□□□ □□□□ □□□□ □□□□ □□□ □□ □□□□□ □□□ □□□ □□□ □□□. □□ □□□□ □□ □□□ □□□□ □ □□□□ □□ □□□ □□□□ □ O(n)□ □□□ □□□□ □□□ □□ □□ □□□□□ □□□.

□□ □ □□, □□□ □□ □□□□ □□□□ □□□□ □□□ □□ □□□□□ □ □ □□. □□□, SPT□ □□□□ □□□□□ □□ □□□ □□□□ □ □□□□ □ □ □□□ □□□□□ □□ □□, □□ □□ □□□ □□□ □□ □□ □□□ □□□ □□. □□ □□□ □□□□ □□□□□ □□□ □□ □□□□ □□ □□□ □□□ □ □ □□ □□□ □□□□□ □ □□□. □□□, SPT□ □□□□ □□□□□ □□□□(sparsely) □□□ SPTE□□ □□□□□ □□□□ □□. □, □□□□ SPTE□ □□□□□ □□□□ □□□ □□□□ □□□ □□.

□□□ □□□ □□□ □□□□□□ □□□ □□□□ □□ □□ □□□□□ □□ □□□□ □□. □□ □□□□ □□ □(key)□□ □(value)□□□ □□□ □□□ □ □□ □□□□ □□□□□, □□ □□□□□ □□□ □□□ □□ □□ □□ □□□ □□ □□ □□□□ □□ □ □□□. □□ □□□□ □□ □□ □□ □□ □□□ □ □□ □□ □□ □□ □□□, □ □□ □□ □□□ □□□ □□□□ □□ □□□□. □□ □□ □□ □□□ □□□□ □□□ □□□ □□□ □□ □□□ □□ □□□□ □ □ □□□□ □□□ □□. □□□, □□ □□□□ □□□ □□□ □□ □□□ □□ □□□ □□ □□□ □□□ □□□□ □□□ □□ □□□□□ □□ □□(hash collision)□ □□□□ □□□. □□□, □□□□, □□□ Pintos□□ □□□□ □□ □□□ □□□ □ □□ □□□□ □□□ □□□□ □□□ □□ □□□□ □ □□□ □□ □□□ □□□□ □□. □□ □□ □□ □□ □□□□□ □ □□□ □□□□□, □□□ □□□□□ □□ □□□ □□□ □□ □□ □□□□□ □□□□ □□□ □□□ □□□.

struct spte

```
/* Maybe into vm/spt.h. */
struct spte
  {
    int size;
    bool swapped;
    void *uaddr;
    mapid_t mapid;
    block_sector_t index;
    struct hash_elem elem;
  }
```

spte □□□□ SPTE□ □□□□ □□ □□□□□. struct spte□ □□, □□ □□□□ 4 KiB □□ □□□ □□□ □□□□ □□□ □□ □□□ □ □□□□ □□□□ □□ size □□□ □□□ □□. □□ □□ □□□ □□□□ □□□□□ □□□□□ □□□□ □□□□□ □□, □□ □□□ □□ 0□□ □□□ □ □ □□□□. □□, □□ SPTE□ □□□□ □□□□ □□ □□ □□□ □□□□□ □□ □□ □□□ □□□ □□□□□□ □□□□ □□ swapped □□□ □□□ □□. uaddr□ □□ SPTE□ □□□ □□□□ □□□ □□ □□ □□□□.

□□, □□ SPTE□ □□ □□□ □□ □□□□ □□ □□ □□□ □□□ □□□□□, □□ SPTE□ □□□ □□□ □□□(map identifier)□ □□□□ □ □ `mapid` □□□ □□□ □□. □□ □□ SPTE□ □□□ □□□□ □□□ □□ □□□□ □□□□ □□□□ □□□□ `mapid`□ `MAPID_ERROR`□ □ □□□□ □□ □□□□ □□□ □□ □□□ □□□ □□□□ □□□ □□□□. □, □□ □□□□ □□ □□□ □□ □□□□ □□□□□□ □□□□ □□ □□□ `index`□ □□□ □□□, □□ □□□□ □□ □□□□ □□□□ □□ □□□ `elem`□ □□□ □□.

struct process

```
/* From userprog/process.h. */
/* An user process. */
struct process
  {
    ...
    /* Owned by vm/spt.c. */
    struct hash spt;
  };
```

□□□ `process` □□□□ □ □□□□□ □□□ □□□□□□ □□□□□, □□□ □□□□□ □□□ □□□ SPT□ □□□□ □□□□ □□, SPT□ □ □□□ □□□□□□ □□□□ □□□□□□□ □ `struct process`□ □□□ SPT□ □□□□ □ □□□. □□, □□□□□ □□□ □□□□ SPT □□ `hash_init()` □□□ □□ □□□□□□ □□.

page_fault()

```
/* From userprog/exception.c. */
static void
page_fault (struct intr_frame *f)
{
  ...

  /* If it is caused by invalid address passed to the kernel, kill user
process
     while making no harm to the kernel. */
  if (is_user_vaddr (fault_addr) && !user)
    process_exit (-1);

  /*
    1. Check if current process's SPT has an entry about the faulting
page.
    2. If the SPT does not have such SPTE, terminate the malicious user
process.
    3. Else, according to the information in the entry, allocate a
physical
       frame which the page would be loaded into. This is done with the
frame
       table.
      - If the physical page frame is full, evict a frame by LRU-
approximating
        page replacement algorithm.
      - Also, find the swap slot by the swap table.
      - Then, modify the evicted page's SPTE so that it now holds the
```

```
  block and
          sector where the page is evicted.
      4. Load the faulting page into the frame using block_read().
      5. Go back to the normal execution flow.
    */
    ...
  }
```

□□□ □□ □□□□ □□ □□□ □□□ □□ □□ □□□ □□□ □□□□ □□□□ □□ □□□□ □□ □□□ □□□□□□ □□□□ □□, □□□ □□□ □□□ □ □□□ □□□ □□□□□ □□□□□□ □□□□ □□□. □□□ □□□□□, □□ □□□ □□ □□□ □□ □□ □□□□ □□□ □□□□ □□ □□□ □□□□□□ □ □□□, □□ □□ □□□ □□□□□ □□ □□□□□ □□□□ □□□ □□□□□ □□.

Lazy Loading and Paging

□□ Pintos□ □□ □□ □□ □□□ □□ □□ □□ □□□ □□□□ □□□□□ □□. □□□ □□□ □□□□□, □□ □□□ □□ □□□ □□□□ □□□□ □□ □□□ □□□□ □□□□ □□□ □□□ □□□ □□□□ □□ □□□ □ □□ □□□□ □□□□ □□□ □□□□ □□□ □□□. □□□, □□ □□□□□□□ □□ □□□ □□□□ □□ □ □□ □□□□ □□ □□□ □□□□ □□□□ □□□ □□(lazy loading)□ □□□□□ □□.

□□, □□□□ □□□□ □□□ □, □□□ □□□ □ □□ □□ □□□ □□□ □□ □□□□ □□□□ □□ □□□ □ □□□ □□ □□□□ □□□ □□□ □□□□□ □ □□□. □□ □□ □□□□ □□□□□ □□□□ □□□□ □□□□□, □□□ □□ □□□□ □□□ □□ □□□ □□ □□□□ □□□ □□ □□□□ □□□ □□ □ □ □□□ SPT□ □□□□ □ □□□. □□, □□□□ □□□□ □□□□ □□□ □, □□ □□ □□□□ □□□ □□□□ □□ □□ □□, □□ □□□□ □□□□□ (dirty) □□□ □□ □□□□ □□□ □□□□ □□ □□ □□□□ □□□. □□□□□, □□□ □□ □□ □□□ □□ □□□□ □□□ □□ □□□ SPT□ □□□ □ □□ □□□□ □□□□ □□□□□□ □□□□ □□ □□□□□ □□□□ □□ □□□□ □□□□ □□. □□□ □□□ □□□ □□□□□ □□□.

basic descriptions

□□ □□□□□ □□□ □□□□□ □□□ □, □□□□□□□ □□□□□ □□ □□ □□□ □□□□ □□□□ □□□ □□□ □□. □□□ □□ □□ □□ □□□ □□ □□□□, □□ □□ □□□ □ □□ □□□□ □□□ □□□ □□. □, □□□□ □□ □□□ □□□ □□□ □□□□ □□□ □□□, □ □□□□ □□ □□ □ □□ □ □ □□□□ □□□□, □□ □□□ □□□ □□ □□□□ □□□ □□□ □□□□ □□□□□ □□□ □□□ □□□ □□□□ □□□□□ □□ □□ □□□ □□□ □ □ □ □□□.

□□□ □□□ □□ □□□ □□□□, □□ □□□ □□ □□□□ □□□ □ □□ □□ □□□ □ □□ □□□□ □□□□ □ □□ □□□ □□□□ □□ □□□□ □□□ □ □□□ □□ □□□ □□□ □□□. □□□ □□□ □□□ SPT□ □□□□□ □ □□□. □, □□□ □□□ □□□□ □□□□ □□□□□ □□ □□ □□□□ □□ □□ □□□ □□ □□□ SPT□ □□□□ □□□ □□□□. □□ □□□ □□□ □□□□ □□□ □□□ □□ □□□ □□ □□□ □□ □□ □□□□ □□ □□ □□□ □□□□ □□□ □□□ □□.

□□, □□□□ □□□ □□□ □□ □□ □□□ □□□□ □□□□□ □□□ □ □□ □□□ □□□□ □□□□ □ □□□ □□, □ □□□ □□□ □□□ □□□□ □□ □□ □□□□ □□□ □□□□ □□□□, □□ □□ □□□ □ □□□□ □□□□□□ □ □□□□ □□□ □□ □□ □□ □□ □□ □□□ □□□ □□ □□□ □□□ □□□□ □□□ □□□ □□.

limitations and necessity

□□□ □□□ □□□□ □□□□ □□, □□ Pintos□ □□ □□ □□ □□□ □□□□.

```
/* From userprog/process.c. */
/* Loads an ELF executable from FILE_NAME into the current thread.
   Stores the executable's entry point into *EIP
   and its initial stack pointer into *ESP.
   Returns true if successful, false otherwise. */
```

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  /* Read program headers. */
  file_ofs = ehdr.e_phoff;
  for (i = 0; i < ehdr.e_phnum; i++)
    {
      struct Elf32_Phdr phdr;
      ...
      switch (phdr.p_type)
        {
          ...
        case PT_LOAD:
        ...
              if (!load_segment (file, file_page, (void *) mem_page,
                                 read_bytes, zero_bytes, writable))
                goto done;
        ...
        }
    }
    ...
}
```

userprog/process.c의 load() 함수를 보면, ELF 파일의 각 세그먼트를 처리할 때 실제 작업을 load_segment() 함수를 호출하여 수행하고 있다. 그렇다면, load_segment() 함수에서 어떤 작업을 수행하는지?

```
/* From userprog/process.c. */
/* Loads a segment starting at offset ofs in file at address
   upage.  In total, read_bytes + zero_bytes bytes of virtual
   memory are initialized, as follows:

        - Read_bytes bytes at upage must be read from file
          starting at offset ofs.

        - Zero_bytes bytes at upage + read_bytes must be zeroed.

   The pages initialized by this function must be writable by the
   user process if writable is true, read-only otherwise.

   Return true if successful, false if a memory allocation error
   or disk read error occurs. */
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ...
  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
```

```
          we will read page_read_bytes bytes from file
          and zero the final page_zero_bytes bytes. */
       size_t page_read_bytes = read_bytes < pgsize ? read_bytes : pgsize;
       size_t page_zero_bytes = pgsize - page_read_bytes;

       /* Get a page of memory. */
       uint8_t *kpage = palloc_get_page (pal_user);
       if (kpage == null)
         return false;

       /* Load this page. */
       if (file_read (file, kpage, page_read_bytes) != (int)
  page_read_bytes)
         {
           palloc_free_page (kpage);
           return false;
         }
       memset (kpage + page_read_bytes, 0, page_zero_bytes);

       /* Add the page to the process's address space. */
       if (!install_page (upage, kpage, writable))
         {
           palloc_free_page (kpage);
           return false;
         }

       /* Advance. */
       read_bytes -= page_read_bytes;
       zero_bytes -= page_zero_bytes;
       upage += pgsize;
     }
   return true;
 }
```

load_segment() □□□□□□ □□□□ □□□□□ □□□□□ □□ file□ □□□□ ofs□□ □□□□□ read_bytes□□ □□□□□□, □□ □□ □□ □□ □□□□□ □□□□ □□ □□□□ □□□□□ □□ □ □ □□. □□, install_page() □□□□ □□□□ □□□□ □□□□□ □□□□ □□ □□ upage□□ □□ □ □□□□□ □□□□, □□ □□□□ □□ kpage□□ □□□□ □□ □□□□□□ □□□□ □□□□□□ □□□□□ □□ □ □ □□. □□ □□□□ □□ load_segment()□ □□ □□□□□□ □□□□ □□□□ □□□□ □□□□ read_bytes□ zero_bytes□ 0 □□□□ □ □□ □□□□□ □□ □□ □ □ □□□□ □□ □□□□□ □□□□□□□ □□□□. □, Pintos□ □□ □□□□ □□□□ □□□□ □□□□ □□□□□ □□ □□, □□ □□□□ □□ □□□□ □□□□□ □□□□□□ □□.

□□, □□□□ userprog/exception.c□ □□□□ □□ □□□□□ □□□□ □□ □□, □□ Pintos□□□ □□□□ □□ □□ □ □□ □□ □□□□□ □□ □ □□□□□ □□□□□□ □□ □□ □□□□□ □□□□ □□□□□ □□□□□ □□ □□, □□ □□□□ □□□□ □□□□□ □□ □□□□□□ □□□□□□ □□ □ □ □□. □, □□ Pintos□□□ □□□□ □□ □□□□□□□ □□□□ □□□□(write-back) □□□□ □□□□□ □□□□□ □□□□□ □□□□.

Design Proposal

□□□ □□□□ □□□□ □□□□□ □□□□□ □□□□□, □□□□ □□□□ □□□□ □□ □□□□□ □□□□ □□ □□□□□ □□ □□□□ □□□□□ load_segment()□ □ □□□□ □□□□□ □□. □, □□ load_segment()□ □□□□□ □□□□ □□□□ □□ □□□□ □□ □□□□□□ □□ □□□□, □□□□ load_segment()□□□ □□□□ □□ □□□□ SPT□□ □□□□□, □□ □□ □□□□□ □□□□□ □□ □□ □□ □□□□□ □□ □□□□□ □□□□□ □□ □□

□ □□□□ □□ □□□ □□□ □□□ □□□□□ □□□□□ □□.

```
/* From userprog/process.c. */
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ...
  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         we will read page_read_bytes bytes from file
         and zero the final page_zero_bytes bytes. */
      size_t page_read_bytes = read_bytes < pgsize ? read_bytes : pgsize;
      size_t page_zero_bytes = pgsize - page_read_bytes;

      /*
        After calculating the number of bytes to be read and the number of
bytes
        to be filled with zeros, we should do the following.

        1. Gets the sector number of the underlying block of the file.
        2. Create new SPTE which holds the sector number from the first
step.
        3. Insert the SPTE with the hash key of upage into the SPT of this
newly
           executed process.
        4. After this step, the executable file will be demand-paged
whenever
           the process trys to executed yet unloaded portion of the
executable,
           by the modified page fault handler discussed above.
      */

      /* Advance. */
      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      upage += pgsize;
    }
  return true;
}
```

□, □□□ □□□ `load_segment()`□□□ □□ □□□ □□ □□□□□ □□□□□ □□, □□ □□ □□□ □□□ □□ □□□□□ □□ □□ □□□ □□□□□ □ □ □□□□□ SPT□ □□□□□.

Stack Growth

Basic Descriptions & Limitations

□□ pintos□□□ stack □ single page(4 KB)□ □□□□ □□□. □, single page□ □□□□ stack space□ □□□ □□□□ page fault□ □□□□ □□ handling□ □□□, stack □□□ □□□□ □□ □□□ □□□ □□□ □□□□ □□ □□□□. □□ □□ □ □□□ □□ □ □□□□□ □□□ □ □□ □□ □□□ □□□□ □□. □□□ □□ □□□□□ □□.

Necessity

□□ □□□ □□□ □□□□ `userprog/exception.c` □ □□□ `static void page_fault (struct intr_frame *f)` □ □□□ □□□□ □□. □□ handler□ □□ □□ □□□ fault□ □□□ address□ □□□ stack growth □ □□□ □□ □□□ □ □□□ □□□ □□□□ □□. □□ □□□ □□ □□□□ □□□□ □□□ □□□ □□ □□□ □□ □□□□ □□□□ □□. `fault_addr`□ □□□□ stack growth □ □□□□ □□□ □□□ □□. □□ □□□□ □□□ □ □□ □□□ x86 □□ □□□□□ esp □□□ □□□ □□□□ □□□ PUSH □□ PUSHA □□□□□□ □□□□□ □□ 4 btye, 32 byte □□□ □□□ □□□□ □□. □□□ □□ 32 byte □□□ □□□ □□□□ □ stack growth□ □□ □□□□ □□, □□□ □□□ □□□□ □□ □□□ □□□ □□□□, □□□□□ □□□□ □□. □□, □□□□□ □□ □□□□ □□□□ □□□ 8 MB (PHYS_BASE□ □□)□ □□ stack size□ □□□□, □ □□□ □□□□ □□□ heap □□□□ □□□ □ □□□□ □□□□□ 8 MB□ □□□ □□□□ □□□□□ □□□□ □□. □□□□□ □□□ □□□ □□□□□ □□□□□□ □□□ □□□□ □□. □, Supplemental Page Table □ □□□□ `fault_addr`□ □□□□ page□ □□ □□□ □□□□□ □□□ □□□□, □□ page□ □□□ □□□□ □□□ □□ □□□□ □□ □□ □, write□ □□□ □□□□ □□ page□ □□ memeory□ □□□□ □□. □□, □□□□□ □□□ □□ user mode□ □□ kernel mode □□ □□□□□ □□□□ `f->esp`□ kernel □ stack □□□ stack growth□ □□□□ □□□□ user thread□ esp□ □□ □□□□, □ stack□ □□□□ □□□ □□□□ □□.

Design Proposal

□□□ □□□ □□ □□ exception.c □ `page_fault()` □ □□□□ □□□□ □□□□. □□ □□□ □□□□ not_present, write, user□ □□□□ not_present □□□ `fault_addr`□ □□□□ Supplemental Page Table□ □□ □□□□ load □□□□ □□□ □□□□ □□□□ □□ □□□ 32 bytes □□□□ □□, `PHYS_BASE` □□□ 8MB □□□□ □□□ □□□□ □□. □□ □□ □□□□□, user mode □ □□□□ `f->esp`□ □□□□, kernel mode□ □□□□ □□□ □□□□ □□ user thread□ □□□ stack pointer□ □□ □□ □□ esp□ growth □□□ □□□□ □□□□□ □□. □ □ □□□□ pointer□ syscall.c□□ `syscall_handler()`□ □□□ □ thread□ `f->esp`□ □□□□ □□, □□ `page_fault()` □□ □□□ □ pointer □□ □□□□ □□□□ □□□□□ □□. □□□□□ stack growth □□□ `pg_round_down (fault_addr)` □ □□□□ □□□ □□□ page□ □□□□ □□□□ □□□□□ □□.

File Memory Mapping

□□□ □□□□□ □□□ □□□□ □, `read()` □□□ □□□ □□□ □□ □□□ □□□ □□ □□□ □□□ □□□ □ □□ □□□□ □□ □□□, □□ □□□□ □□□ □□□□□ □□ □□ □□□ □□□ □□ □□□ □□□□ □□ □□□ □□□ □□ □□□ □□□□ □□□□ □□ □□. □□□ □□□ □□□ □□ □□□ □□ □ □□□ □□ □□(memory mapped file)□□ □□□. □□□ □□ □□□ □□□□ `read()` □□□ □□□ □□□ □□□ □□□□□ □, □□□ □□□□□ □□ □□□ □□□□ □□□ □□ □□□□□□ □□□ □□. □□ □□□□ □□□ □□□ □□□ □□□ □□ □□□□ □□□□ `read()`□ □□□ □ □□ □□□□ □ □□□, □□□ □ □□□ □□□□ □□ □□□ □□□□ □□□□ □□□□ □□□ □□ □□□ □□□□ □□□.

Basic Descriptions

□□□ □□ □□□ □□□ □□□ □□□□□ □□□□ □□□ □□ □ □□□□ □□□ Unix□ `mmap()`□ □□□□. Pintos □□□ □□□ `mmap()`□ □□ □□(prototype)□ Unix `mmap()`□ □□ □□□ □□ □□□□, □□□ □□ □□□ □□□ □□□ □□□ □□ □□ □□□ □□ □□□ □□ □□ □□ □□□ □□□□□ □□□□ □□. `mmap()`□□ □□□ □□□□□ □□□ □□ □□ □□□ □□□ □□□□ □□. `munmap()`□ Unix□□ □□□ □ □□ □□□□ □□□ □□□, Pintos□ `munmap()`□ □□□ □□□□ □ □□ □□ □□□ □□□□ □□ □□□ □□□ □□□ □□□ □□□ □□□□ □□ □□ □□ □□ □□.

`mmap()`□ □□ □□□ □□□ □□□□, □□ □□□(file descriptor) `fd`□ □□ □□□ □□□□ □□ □□ `addr`□ □□ □□ □□□ □□ `mappid_t` □□ □□□ □□□□ □□ □ □ □□. □, Unix□ `mmap()`□ □□□ □□□ □□□ □□□ □ □□ □□□ □□□ □□□ □□□ □□□□□ □□ □□, Pintos□ `mmap()`□ □□□ □□ □□□ □□□□□ □□ □□□ □□ □□□ □□ □□□ □□□ □□□ □□□□, □□□ □□ □□ □□□ □□ □ □□ □□□□□□□ □□ □□ □□□ □□ □□□ □□□□ □□ □ □ □□.

munmap()□ □□ □□□ □□ □□□ □□□□ □□□ □□ □□ □□ □□□ □□ □□□□ □□ □ □ □□. mmap()□ □□ □□ □□□ □□ □□□ □□ □□ munmap()□ □□ □□ □□□ □□ □□□□□ □□ □□□□ □□□□ □□□ □□, □□□ □□□ □□ □□□□ □□ □□□ □□□□ □□□□ □□□ □ □□. □□□, mmap()□ munmap()□ □□□□ □□□□ □□□ □□□ □□□□ □□□□□ □□□□□, SPT□ □□□ □□□ □□□□ munmap() □□□□ SPT□ □□□□ □□□ □□□□ □□□□ □□□ □□□ □□. □ □□□□□ □□□ □□□ □□□□.

Limitations and Necessity

```
/* From userprog/syscall.c. */
static void
syscall_handler (struct intr_frame *f)
{
  int syscall_number = (int) dereference (f->esp, 0, WORD_SIZE);

  switch (syscall_number) {
    case SYS_HALT: halt (); break;
    case SYS_EXIT: exit (f->esp); break;
    case SYS_EXEC: f->eax = exec (f->esp); break;
    case SYS_WAIT: f->eax = wait (f->esp); break;
    case SYS_CREATE: f->eax = create (f->esp); break;
    case SYS_REMOVE: f->eax = remove (f->esp); break;
    case SYS_OPEN: f->eax = open (f->esp); break;
    case SYS_FILESIZE: f->eax = filesize (f->esp); break;
    case SYS_READ: f->eax = read (f->esp); break;
    case SYS_WRITE: f->eax = write (f->esp); break;
    case SYS_SEEK: seek (f->esp); break;
    case SYS_TELL: f->eax = tell (f->esp); break;
    case SYS_CLOSE: close (f->esp); break;
    /* There's no handling routine for mmap() and munmap()! */
  }
}
```

□□□ Pintos □□□ □□□□, mmap()□ munmap() □□□ □□□ □□□□ □□ □□□ □□ □□□□ □□□□ □□ □□□, □□ □□□ □□□ □ □□□□ □ □□□□ □□ □□□ □□□□ □□□ □□□□ □□ □□ □□ □ □ □□.

Design Proposal

syscall_handler()

```
/* From userprog/syscall.c. */
static void
syscall_handler (struct intr_frame *f)
{
  int syscall_number = (int) dereference (f->esp, 0, WORD_SIZE);

  switch (syscall_number) {
    ...
    case SYS_MMAP: f->eax = mmap (f->esp); break;
    case SYS_MUNMAP: munmap (f->esp); break;
```

```
    }
  }
```

□□, □□ □□□ □□□ □□□□□ □□□□ □□□□ □□ □□□ □□□ □□ □□□□ □□ □□□ □□□□ □□□ □□□□ □□.

mmap()

```
/* Maybe into userprog/syscall.c. */
static uint32_t
mmap (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  void *addr = (void *) dereference (esp, 2, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if (fp == NULL)
    return MAPID_ERROR;

  /*
    To implement mmap() system call, this function should do followings;

    1. Allocate a mapid by allocate_mapid() call.
    2. Get the size of file, divide it by the size of a page (4 KiB). Let
the
       quotient be N.
    3. Add (N + 1) SPTEs into the SPT of current process.
      - The SPTE should have the underlying block, sector number, and the
        size within the page if the size of the file is not aligned with
the
        size of a page.
      - The SPTE should also have the mapid for this mapping.
    4. Return the mapid allocated in the first step.
  */
}
```

mmap() □□□ □□□□□ □□□ □□□□ □□□□, □□□ □□□ □□ □□□ □□□ SPTE□ □□ □□□□□ SPT□ □□□□. □□ □□□ □□□ 15 KiB□□, □ 4□(= 15 / 4 + 1 □)□ SPTE□ SPT□ □□□□, □□□ SPTE□ size □□□ 3□□ □□□□□□ □ □□□. □□, □□ SPTE□ □□ fd□ □□□□ □□ □□□ □□□ □□ □□□ □□ □□ □□□ □□ □□□ □□□ □□□ □□. □□ □□ □□□ □□□□ □□□□ □□ □□□ □□□ □□□□, □□□ □□ □□□□ mmap()□□ □□□ SPTE□ □□□ □□□ □□□□ □□□□□ □□□□.

allocate_mapid()□ □ □□□□ □□□ □□□□ □□□□ □□□, threads/thread.c□ allocate_tid()□ filesys/file.c□ allocate_fd()□ □□ □□□ □□□ □□□ □□.

munmap()

```
/* Maybe into userprog/syscall.c. */
static void
munmap (void *esp)
```

```
{
  mapid_t mapid = (mapid) dereference (esp, 1, WORD_SIZE);

  /*
    To implement munmap() system call, this function should do followings;

    1. Iterate through the SPT of current process, mark all the pages as
not
       present if the page's mapid is same with the mapid to unmap.
    2. Also, for each pages to be unmapped, write them back to the disk if
       the page is dirty.
    3. Remove all of SPTEs whose mapid is to be unmapped, from current
process's
       SPT.
    4. Deallocate all physical frames occupied by the mappings.
  */
}
```

munmap() □□□ □□□ □□□□ □□□, □□□□□ □□ □□□ □□□ mapid□ □□□ mapid □□□ □□ SPTE□ □□ □□□□□ SPT□□ □□□. □□, □□□ □□□ □□□ SPTE□□ □□□ □□ □□□□□ □□□ □□□ □□□ present □□□ 0□□ □□□ □□ □□□□ □ □□ □□□□ □□□ □□□□, □□ □□ □□□□ dirty □□□ 1□□□ □□ □□□□ □□ □□□□ □□. □□□□ □□ □□□ □□□ □□□□□ SPTE□ SPT□□ □□□□, □□□ □□□ □□□ □□□ □□□ □□ □□□□.

Swap Table

Basic Descriptions and Limitations

□□□□□ □□□□ HW□ □□□ physical memory space□ □□ □□. □ □□ □□□ □□□ □□□□ □□ □□ □□□ □□ □□□□ □□ □□ □□□□ □□□, □□□ □□□□ □□ □ □□□□ □□ □□□ □□□ □□□, □□□ □□□□ □□□□ □□. □□□, pintos□□□ memory mapping□ □□□□ □□ □□□□ □□□□ □□ □□□□ □□□ □□□□ □□ □□□ □□□□ □□.

□ □ □□ □□□□ □□□□ □□□□ □□□ □ □□□ □□□□ □□□ □□ swap disk □□. disk □□ □□□□ □□□□, □□ □□□□ physical memory□ □□□ □□ □□□ □ □□ □□□□. □ □ □□□□□ swap disk□ □□□□ □□ swap out, □□□ □□□□ □□□□ □□ swap in □□□ □□. □□ pintos □□□ □□□ swap disk □ struct block□ □□ □□□□ □□. □ block□ sector(512 B) □□□ □□□□ □□□, swap-in, out□ page □□□ □□□□ □□□ pintos □□□□ 8□□ sector□ □ □□ swap slot□□ □□□□□ □□□ □ □□. □□, □ □□□ □□ block□ □ □□ □□□□ □□, block_get_role() □ □□ □□□□ □□(BLOCK_SWAP)□ □□ block□ □□□□ □□□ □ □□ □□. □□ □□□ □□ □□ □□ swap disk□ pintos-mkdisk swap.dsk □□□□ --swap-size □□□ □□ □ □□ □□□ □□ swap disk□ □□□ □ □□. □□□ swap disk□ □ □□□ swap slot□ page□ □□□ □□□ □□□□, □□□ □□□ □ □□□□ □□□□ □□. swap disk□ □□ □□□ physical memory□ □□□ page□ □□□□, □□ □□□□ □□□ □□□□□ □□ □□ □ □□ page size(4 KB)□ □□□ swap slot□□□ □□. □□□ swap slot □□, page-aligned □□□ □□, □□□ slot□ □□□ □ □□□□ □□□□ □□ swap table□□ □□□ □ □□. □□□ swap table□ □ slot□ □□□□ □□□ □□□ □□□□ □□□ □□, □□ □□□ □□□ □□□ □□(□□ □□ □□)□□ □□ □□□□ □□□□ □□. □□ pintos □□□□ struct block □ □□□□ □□ □□□, □□ □□□□ □□ □ □□□ □□ □□ □□□□. □□□ □□□ □□□ □□ □□ physical memory□ □□□□ □□□ □ □□ □□□ □ □□□□ □□ □□□□ □□.

Necessity

swap table□ □□□ □□□ □□□ swap slot□ □□□□ □□ table□□. □□□ □ slot □ □□ □□□□ □□□, □□ □□□□□ □□ □□ □ □□□ □□□ □□. □□, physical memory □□ page□ □□□□ □□□□, □□ page□ □□□□ □□ □□□□ swap slot□ □□□□ □ □□ □ □□□ □□, □□□ □□□ □□ □□□□ □□□□ □□ □□ swap slot□ □□□□ process□ □□□ □□□□ □□□ □□□ □□ □□ □□□□

□□. □□, swap disk □□ □□ □□ executable data□ □□ □□□□□ □□□, □□ □□□ slot□ □□□ □□□□□ □□□. □□□ lazy allocation□ □□□□ □□ eviction □□(physical memory□ □□□ □)□ □□□□ slot□ □□□ □□□□ □□.

□□□ swap table□ □□□□ □□□□ □□ swap out, swap in, swap free□ □□□□. □□ swap out□ memory□□ page□ □□□□ swap slot□ □□□□□ □ □ swap table□ □□□ □□□□ swap slot□ □□□□, □□ slot□ `block_write()`□ □□□ page□ □□□□ □□□□ □□, □□□ □□□□ □□□ □□□□□ □□ □□. □ □ □□□□ □□ □□□ □□□ eviction policy □□, □□ physical memory □□ □□□ page □□□ □□ page□ □□□□ eviction □ □□ □□□ □□. □□□ □□□ page□ eviction □□ swap disk □ □□□□□ □□□ □ □□. □□□ swap in □□□ □□□□ □□□□ page fault □□ □□□ swap disk □ □□□□ □□ page□ physical memory□ □□ □□□ □□□, □ □ □□□ page□ □□□□, □□ page□ `block_read()` □ □□ □□□ swap slot□ □□□□ □□□□ □□. □□, □□ □□□□ □□□□ □□ slot□ □□□ □□□□□ □□. □□□□□ swap free□ □□□□ swap in□□ □□□□ □□□ physical memory□ □□ □□□□ □□ □□ slot□ □□□□ process□ □□□ □ □□□ □□□□ □□ □□ slot□ free □□□ □□.

□□, swap disk □ □□□□ □□ process □□□□ □□ □□□□ □□□□. □□□ swap table□ □□ □ □□□□ □□□□ □□□□□ □□□ □□ □□□ □□□□□ □ □□ overhead□ □□□ □ □□. □□□ □□□□□ vm/swap.c □ □□□□ □□□□ □□□□□, □ □□ bitmap□ □ □ □□ disk □ □□□□□ □□□ □ □□ □□□. □□□ □ □□ lock □ □□□ □□ bitmap □ □□ □ □□ □□ □ atomic □ □□□□□ □□ □□□□ □ □□□.

Design Proposal

swap table□ □□ □□□□ □□□ □□□ struct bitmap□ □□□□ □□□□□ □□. vm/swap.c □ □□□ □□□ □□□□ □□ □□ □□ □□□ □□□ □□□□□ □□.

On Process Termination

Basic Descriptions and Limitations

□□ □□□ □□□□ □□ □□□ □□□□ process □ □□□ □□□ □□□ □□□□ □□ □□□□ □□ □□□ □□ □□□□ □□. □□□ □□ □□□ □□□ □□□ frame table, supplementary page table, file memory mapping, swap table □□ □□□ □□□ □□□ □ □□□ □□, lock □ □□□ □□□□ □□ □□□□ □□ □□ □□ □□□□ □□□ □□□□ □ □□□□ □□□□□, □□ □□□□□ userprog/process.c □□ □□□ process_exit()□ destory_process() □ □□ □□□□ □□□, □□ □□□ process□□ □□ □□ file pointer □□ close □□□, directory □□ free □□□ □□□□ □□□□ □□. □□□ □□□ □□□□□ □□□ □□□ □□□ □□□□ □ □□□□ □□□, swap slot □□ □□□ □□ □□□ □□ □□□□ □□ □□□□ □□. □□□ □□ □□□ □□□□□ □□.

Necessity and Design Proposal

□□ on process termination □ □□□□ □□□ userprog/process.c□ □□□, process_exit() □□ □□□ □□□ □□□ □ □. □□□□ Supplementary Page Table□ □□□□ □□ □□□□. □□□□ □□ □□□ Page□ □□□□□ □□□, □ □□ dirty □ □□ □□ dirty □ □□ □□□□ write □□ □ □□□ □□□□□ □□□□ □□□□ □□. Swap Table□□□ □□ process□ □□□□ □□ (□□ □ □□□) swap slot□□ □□ □□□ swap table□□ □□ □□□□□ □□□□ □□. File Memory Mapping □ □□ □□□□ close_file() □ □□□□□, load □□ □□ □□□ □□□ □□□□ S-page table □□ □□□□ □□ file □ □□ □□□ □□ □□□□ □□ file □ close □□□ □□ □□□□ □□□□□ □□. □□, dirty □ □□□□□ □□ □□□□ □□□□□□ □□ write □□□ □□□ □□□□, □ □□ □□ memory mapping□ □□□□□ □□□□ □□ process□ □□ □□ file memory mapping □ □□□□□ □□. □□□□□ □□ □□ □ □□ frame table □□, □ □□ □□ □□□□□ □□ process□ □□□□ □□ frame □□□ □□□□ □□ □□□□. □□□ □□ process □ □□□□ frame□ frame table□□ □□□ □□ □□□□, □ □ □□ □□ frame□ dirty □ □□□□□ (□□□□□□) □□ frame □ □□□□ □□□□ write □□ □□□□ □□□□□ □□. □□ frame□ □□□□ process □ □□□ □ □□ □□.