Pintos lab 2

Instructor: Youngjin Kwon

Big picture: Running a user program

- Running user programs in OS
  - Create process and thread
  - Setup virtual address of the program: code, data, stack
  - Load executable file
  - Start executables: passing parameters

Booting: **main()** threads/init.c

```
  /* Initialize ourselves as a thread so we can use locks,
     then enable console locking. */
  thread_init ();
  console_init ();

  /* Initialize memory system. */
  mem_end = palloc_init ();
  malloc_init ();
  paging_init (mem_end);

#ifdef USERPROG
  tss_init ();
  gdt_init ();
#endif

  /* Initialize interrupt handlers. */
  intr_init ();
  timer_init ();
  kbd_init ();
  input_init ();
#ifdef USERPROG
  exception_init ();
  syscall_init ();
#endif
  /* Start thread scheduler and enable interrupts. */
  thread_start ();
  serial_init_queue ();
  timer_calibrate ();

#ifdef FILESYS
  /* Initialize file system. */
  disk_init ();
  filesys_init (format_filesys);
#endif

#ifdef VM
  vm_init ();
#endif

  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);

  /* Finish up. */
  if (power_off_when_done)
    power_off ();
  thread_exit ();
}
```

Code flow

```
pintos --fs-disk=10 -p tests/userprog/args-single:args-single -- -q -f run 'args-single onearg'
```

run_actions() threads/init.c

```
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /*
      int argc;                         /*
      void (*function) (char **argv);   /*
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };
```

```
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv) {
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  if (thread_tests){
    run_test (task);
  } else {
    process_wait (process_create_initd (task));
  }
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

Process_create_initd() userprog/process.c

Code flow

```
pintos --fs-disk=10 -p tests/userprog/args-single:args-single -- -q -f run 'args-single onearg'
```

- Running user programs
  - **Create process and thread**
  - Setup virtual address of the program: code, data, stack
  - Load executable file
  - Start executables: passing parameters

```
tid_t
process_create_initd (const char *file_name) {
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
   * Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, initd, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

```
/* A thread function that launches first user process. */
static void
initd (void *f_name) {
#ifdef VM
  supplemental_page_table_init (&thread_current ()->spt);
#endif

  process_init ();

  if (process_exec (f_name) < 0)
    PANIC("Fail to launch initd\n");
  NOT_REACHED ();
}
```

Code flow

- Running user programs
  - Create process and thread
  - **Setup virtual address of the program:** code, data**, stack (kernel stack)**
  - Load executable file

thread_create() threads/thread.c

```
tid_t
thread_create (const char *name, int priority,
    thread_func *function, void *aux) {
  struct thread *t;
  tid_t tid;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Call the kernel_thread if it scheduled.
   * Note) rdi is 1st argument, and rsi is 2nd argument. */
  t->tf.rip = (uintptr_t) kernel_thread;
  t->tf.R.rdi = (uint64_t) function;
  t->tf.R.rsi = (uint64_t) aux;
  t->tf.ds = SEL_KDSEG;
  t->tf.es = SEL_KDSEG;
  t->tf.ss = SEL_KDSEG;
  t->tf.cs = SEL_KCSEG;
  t->tf.eflags = FLAG_IF;

  /* Add to run queue. */
  thread_unblock (t);

  return tid;
}
```

```
static void
init_thread (struct thread *t, const char *name, int priority) {
  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->tf.rsp = (uint64_t) t + PGSIZE - sizeof (void *);
  t->priority = priority;
  t->magic = THREAD_MAGIC;
}
```

```
/* Function used as the basis for a kernel thread. */
static void
kernel_thread (thread_func *function, void *aux) {
  ASSERT (function != NULL);

  intr_enable ();       /* The scheduler runs with interrupts off. */
  function (aux);       /* Execute the thread function. */
  thread_exit ();       /* If function() returns, kill the thread. */
}
```

Code flow

Kernel_thread() → initd() → process_exec()

- Running user programs
  - Create process and thread
  - **Setup virtual address of the program: code, data, stack (user stack)**
  - **Load executable file**
  - Start executables: passing parameters

Process_exec() userprog/process.c

```
/* Switch the current execution context to the f_name.
 * Returns -1 on fail. */
int
process_exec (void *f_name) {
  char *file_name = f_name;
  bool success;

  /* We cannot use the intr_frame in the thread structure.
   * This is because when current thread rescheduled,
   * it stores the execution information to the member. */
  struct intr_frame _if;
  _if.ds = _if.es = _if.ss = SEL_UDSEG;
  _if.cs = SEL_UCSEG;
  _if.eflags = FLAG_IF | FLAG_MBS;

  /* We first kill the current context */
  process_cleanup ();

  /* And then load the binary */
  success = load (file_name, &_if);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    return -1;

  /* Start switched process. */
  do_iret (&_if);
  NOT_REACHED ();
}
```

Code flow

Kernel_thread() → initd() → process_exec() → load()

- Running user programs
  - Create process and thread
  - **Setup virtual address of the program: code, data, stack (user stack)**
  - **Load executable file**
  - Start executables: passing parameters

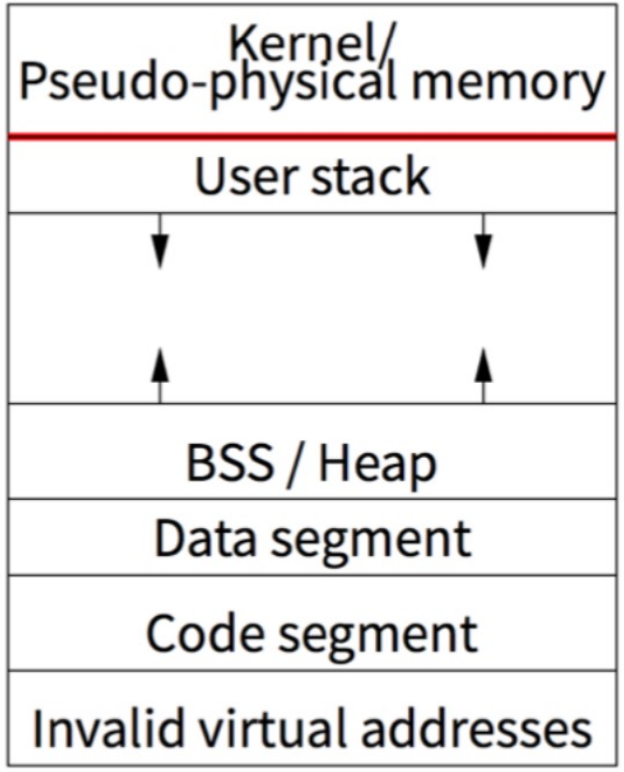
load() userprog/process.c

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }
```

[read executables to code, data]

```
/* Set up stack. */
if (!setup_stack (if_))
  goto done;

/* setup argument */
*(file_name + strlen(file_name)) = ' ';
setup_argument (if_, file_name);

/* Start address. */
if_->rip = ehdr.e_entry;

/* TODO: Your code goes here.
 * TODO: Implement argument passing (see project2/argument_passing.html) */

success = true;
```

Pintos virtual memory layout



KERN_BASE: 0x8004000000

USER_STACK: 0x47480000

0x0400000

Parameter passing

- User strtok_r in lib/string.c to break “char *file_name” into command line and arguments
  - /bin/ls –l foo bar → “/bin/ls”, “-l”, “foo”, ”bar”
  - Put the arguments onto the user-level stack
  - You must follow **calling convention** (ABI)

Example: /bin/ls –l foo bar

Address	Name	Data	Type
0x4747fffc	argv[3][...]	'bar\0'	char[4]
0x4747fff8	argv[2][...]	'foo\0'	char[4]
0x4747fff5	argv[1][...]	'-l\0'	char[3]
0x4747ffed	argv[0][...]	'/bin/ls\0'	char[8]
0x4747ffe8	word-align	0	uint8_t[]
0x4747ffe0	argv[4]	0	char *
0x4747ffd8	argv[3]	0x4747fffc	char *
0x4747ffd0	argv[2]	0x4747fff8	char *
0x4747ffc8	argv[1]	0x4747fff5	char *
0x4747ffc0	argv[0]	0x4747ffed	char *
0x4747ffb8	return address	0	void (*) ()

Point `%rsi` to `argv` (the address of `argv[0]`) and set `%rdi` to `argc`

X86_64 ABI

The calling convention of the System V AMD64 ABI is followed on Solaris, Linux, FreeBSD, macOS,[21] and is the de facto standard among Unix and Unix-like operating systems. The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9 (R10 is used as a static chain pointer in case of nested

main function of a user-level application

```
#include "tests/lib.h"

int
main (int argc, char *argv[])
{
  int i;
```

Code flow

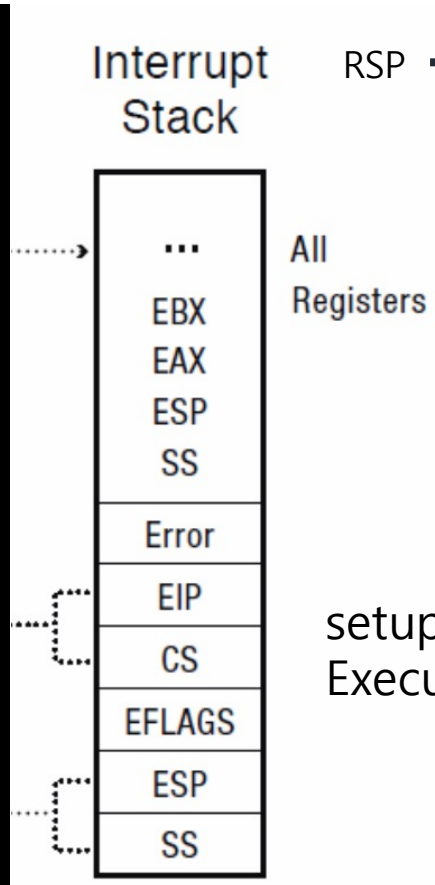
Kernel_thread() → initd() → process_exec() → do_iret()

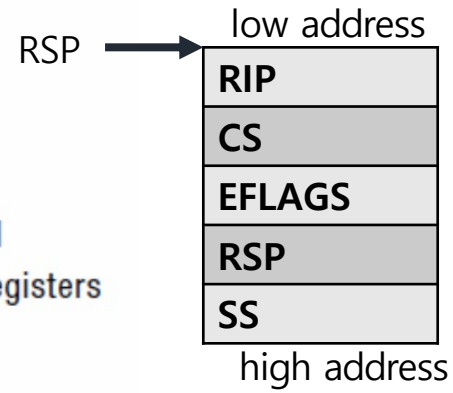
- Running user programs on top of OS
  - Create process and thread
  - Load executable file
  - Setup virtual address of the program: code, data, stack (user stack)
  - **Start executables: passing parameters and jump to the user-level application**

```
/* Switch the current execution context to the f_name.
 * Returns -1 on fail. */
int
process_exec (void *f_name) {
  char *file_name = f_name;
  bool success;

  /* We cannot use the intr_frame in the thread structure.
   * This is because when current thread rescheduled,
   * it stores the execution information to the member. */
  struct intr_frame _if;
  _if.ds = _if.es = _if.ss = SEL_UDSEG;
  _if.cs = SEL_UCSEG;
  _if.eflags = FLAG_IF | FLAG_MBS;

  /* We first kill the current context */
  process_cleanup ();

  /* And then load the binary */
  success = load (file_name, &_if);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    return -1;

  /* Start switched process. */
  do_iret (&_if);
  NOT_REACHED ();
}
```

Magic of jumping from kernel to user-level

```
struct intr_frame {
  /* Pushed by intr_entry in intr-stubs.S.
     These are the interrupted task's saved registers. */
  struct gp_registers R;
  uint16_t es;
  uint16_t __pad1;
  uint32_t __pad2;
  uint16_t ds;
  uint16_t __pad3;
  uint32_t __pad4;
  /* Pushed by intrNN_stub in intr-stubs.S. */
  uint64_t vec_no; /* Interrupt vector number. */
/* Sometimes pushed by the CPU,
   otherwise for consistency pushed as 0 by intrNN_stub.
   The CPU puts it just under `eip', but we move it here. */
  uint64_t error_code;
/* Pushed by the CPU.
   These are the interrupted task's saved registers. */
  uintptr_t rip;
  uint16_t cs;
  uint16_t __pad5;
  uint32_t __pad6;
  uint64_t eflags;
  uintptr_t rsp;
  uint16_t ss;
  uint16_t __pad7;
  uint32_t __pad8;
} __attribute__((packed));
```





When iret is executed,

The instruction to continue execute

The code segment selector to change to

The value eflags register to load

The stack pointer to load

The stack segment selector to change to

do_iret()

setup the intr_frame and
Execute iret to do jump to the user-level entry point

Now CPU jumps to the address where RIP points....

Things you should do. Good luck!

- Safe user-memory access
- System calls
- Process exit message
- Deny writes to executable memory
- Get used to file system APIs

```c
static void
__do_fork (void *aux) {
  struct intr_frame if_;
  struct thread *parent = (struct thread *) aux;
  struct thread *current = thread_current ();
  /* TODO: somehow pass the parent_if. (i.e. process_fork()'s if_) */
  struct intr_frame *parent_if;
  bool succ = true;

  /* 1. Read the cpu context to local stack. */
  memcpy (&if_, parent_if, sizeof (struct intr_frame));

  /* 2. Duplicate PT */
  current->pml4 = pml4_create();
  if (current->pml4 == NULL)
    goto error;

  process_activate (current);
#ifdef VM
  supplemental_page_table_init (&current->spt);
  if (!supplemental_page_table_copy (&current->spt, &parent->spt))
    goto error;
#else
  if (!pml4_for_each (parent->pml4, duplicate_pte, parent))
    goto error;
#endif

  /* TODO: Your code goes here.
   * TODO: Hint) To duplicate the file object, use `file_duplicate`
   * TODO:       in include/filesys/file.h. Note that parent should not return
   * TODO:       from the fork() until this function successfully duplicates
   * TODO:       the resources of parent.*/

  process_init ();

  /* Finally, switch to the newly created process. */
  if (succ)
    do_iret (&if_);
error:
  thread_exit ();
}
```

__do_fork()

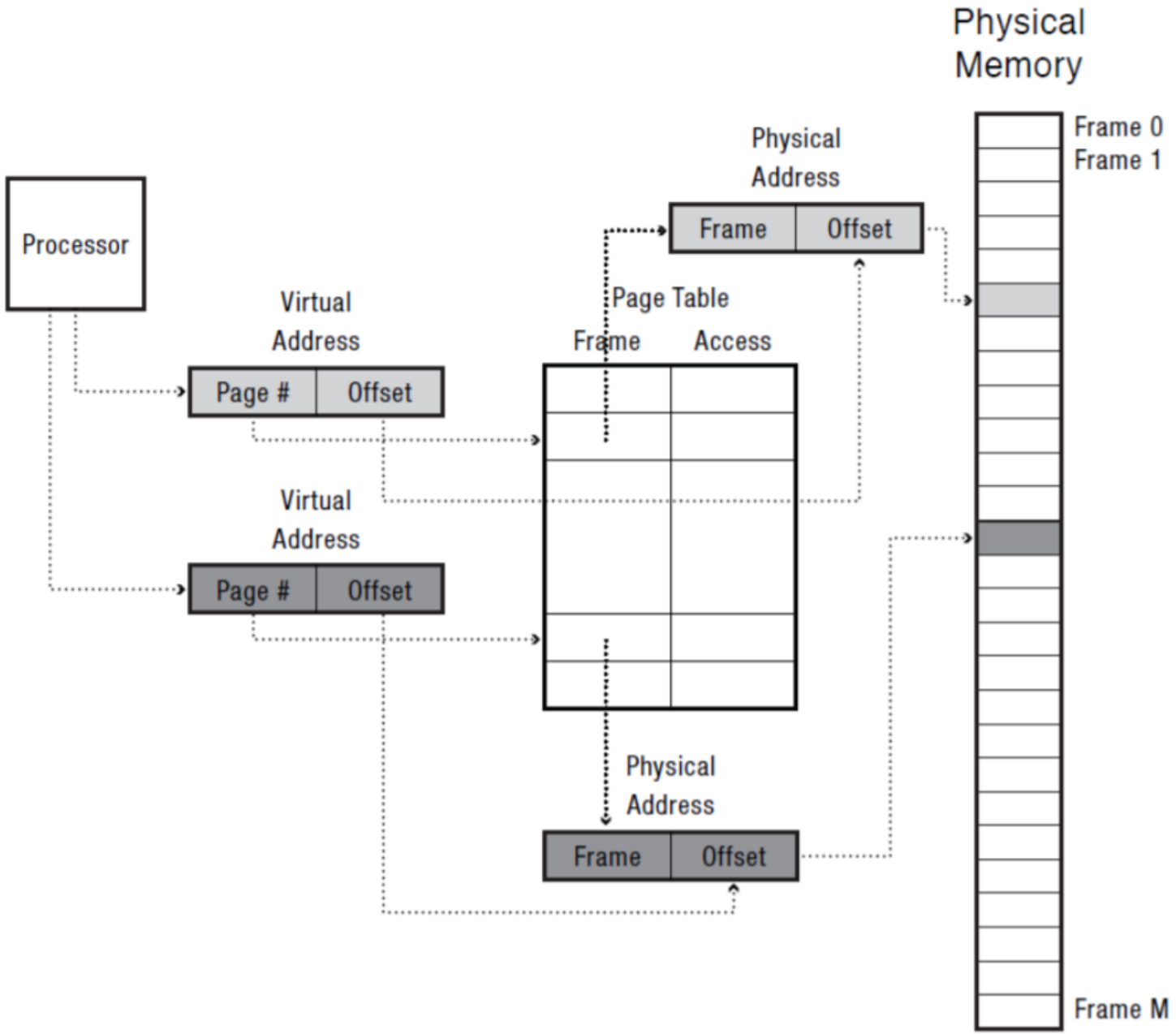
duplicate_pte()

- Parent and child must start with the same physical memory

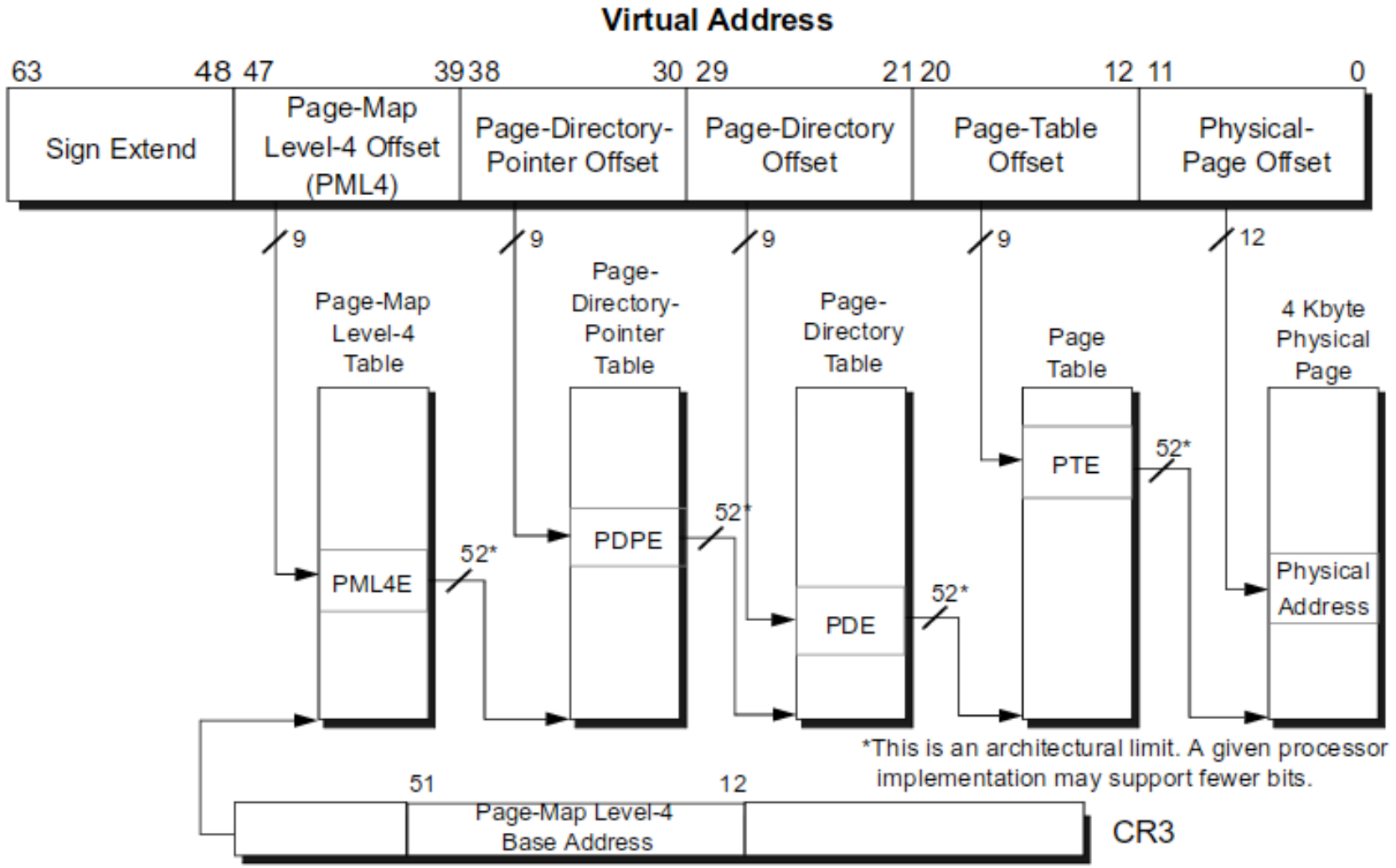
TODOs:

- Parent inherits file resources (e.g., opened file descriptor) to child

Paging


Physical Memory
Processor
Physical Address
Frame
Offset
Frame 0
Frame 1
Page Table
Virtual Address
Page #
Offset
Frame
Access
Virtual Address
Page #
Offset
Physical Address
Frame
Offset
Frame M

Intel X86 page table



Duplicate_pte()

```
static bool
duplicate_pte (uint64_t *pte, void *va, void *aux) {
  struct thread *current = thread_current ();
  struct thread *parent = (struct thread *) aux;
  void *parent_page;
  void *newpage;
  bool writable;

  /* 1. TODO: If the parent_page is kernel page, then return immediately. */

  /* 2. Resolve VA from the parent's page map level 4. */
  parent_page = pml4_get_page (parent->pml4, va);

  /* 3. TODO: Allocate new PAL_USER page for the child and set result to
   *    TODO: NEWPAGE. */

  /* 4. TODO: Duplicate parent's page to the new page and
   *    TODO: check whether parent's page is writable or not (set WRITABLE
   *    TODO: according to the result). */

  /* 5. Add new page to child's page table at address VA with WRITABLE
   *    permission. */
  if (!pml4_set_page (current->pml4, va, newpage, writable)) {
    /* 6. TODO: if fail to insert page, do error handling. */
  }
  return true;
}
```

What API you have to use?

How to duplicate memory a content of parent (parent_page) to the new child page?