Sistemas Operativos. Práctica: Invalid Memory Access

Ricchy Alain Pérez Chevanier
José Emiliano Cabrera Blancas

1. Objetivo

Que el alumno implemente acceso seguro a memoria realizado por procesos de usuario, ya sea en contexto de usuario o en contexto de una *system call*.

2. Introducción

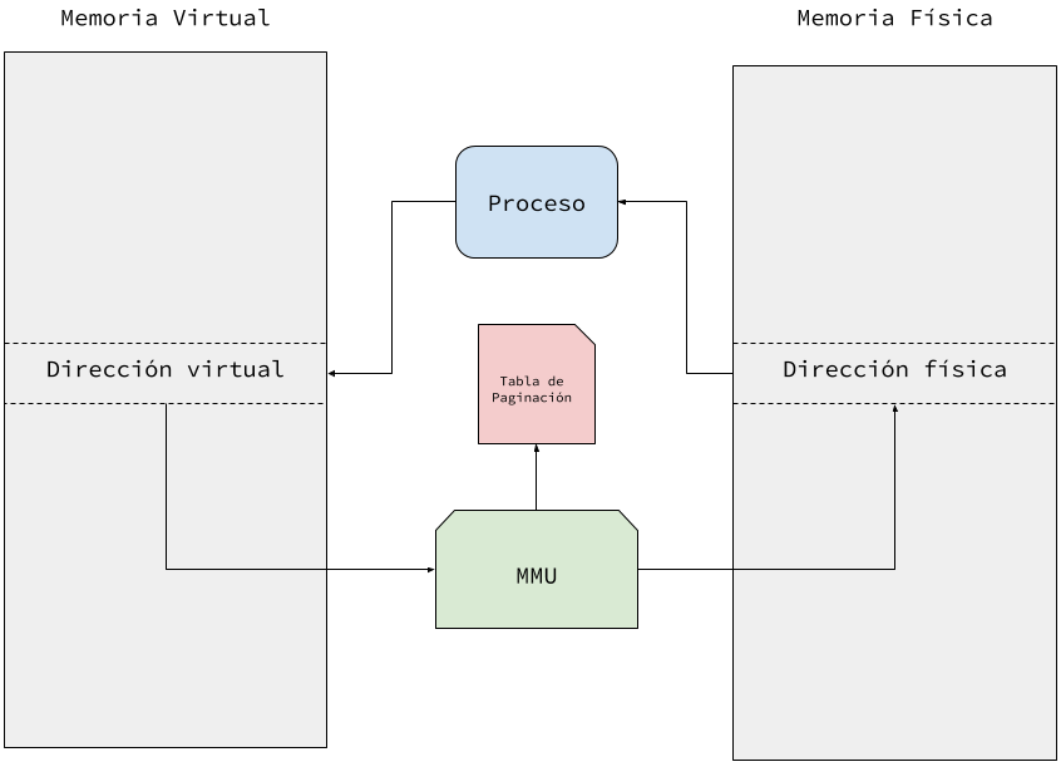
2.1. Memoria Virtual

Una parte indispensable de todo sistema operativo moderno es la memoria virtual. La memoria virtual es un mecanismo del sistema operativo que permite a los procesos, tanto de usuario como de kernel, utilizar espacios de direcciones diferentes al espacio de direcciones reales de la memoria principal. La finalidad de que los procesos no accedan directamente a la memoria física tiene que ver principalmente con dos propósitos:

1. **Seguridad**: Es posible aislar el espacio de memoria de los procesos, de tal forma que no puedan acceder a la memoria de otros procesos.
2. **Administración de memoria**: Es más fácil administrar la memoria, pues se tiene en todo momento control de que regiones de la memoria están ocupadas y por cual proceso.

Las arquitecturas basas en Intel x86 tienen dos modos de uso de la memoria, el modo virtual y el modo real, usualmente el modo real solo se utiliza para inicializar el sistema operativo y una vez realizada dicha tarea el procesador siempre trabaja en modo virtual. Cuando el procesador está en modo virtual, si un proceso trata de acceder a una dirección de memoria, el procesador necesita traducir la dirección (virtual) en la dirección física (real), para ello se utiliza un dispositivo llamado *MMU* (*Memory Managment Unit*).

Para que el dispositivo *MMU* sea capaz de traducir una dirección virtual en una dirección física, necesita una estructura que le permita realizar dicha traducción. La estructura que utiliza el MMU se llama tabla de paginación, el nombre hace referencia al concepto de paginación el cual se explicará a detalle más adelante. La parte importante de la tabla de paginación es que la *MMU* la utiliza como un diccionario/mapeo para poder realizar la traducción.



La memoria virtual en Pintos esta divida en dos regiones: la memoria virtual de procesos de usuario y memoria virtual del kernel.

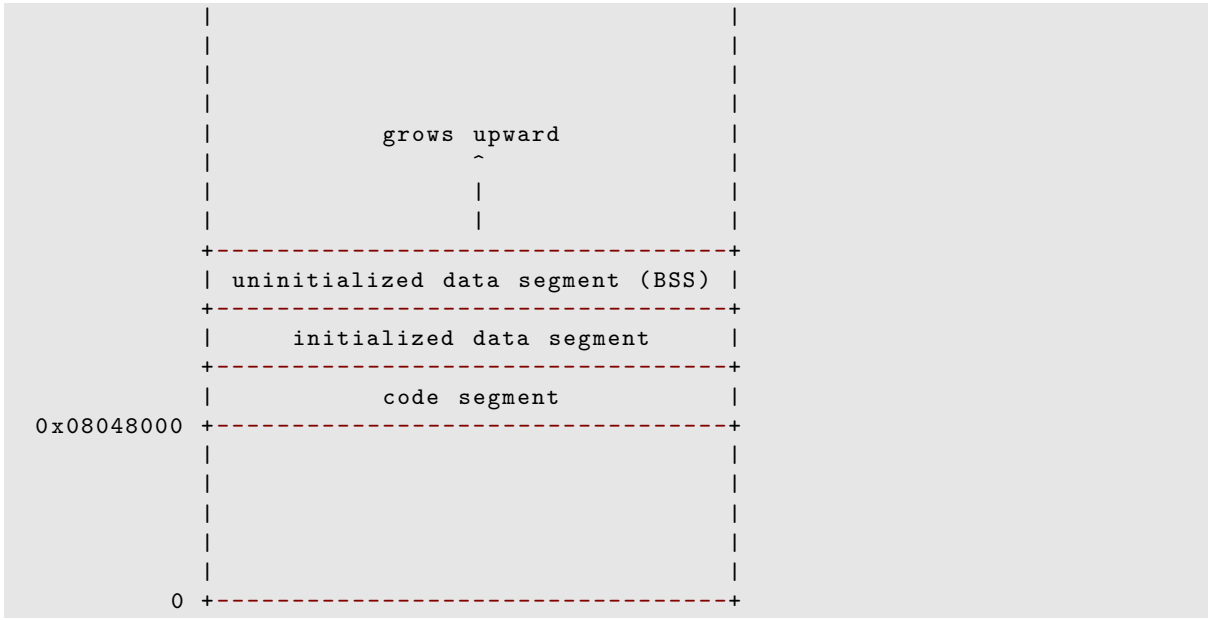
La memoria virtual del proceso de usuario tiene un rango que empieza desde *0* y termina en `PHYS_BASE`, constante que esta definida en el archivo `threads/vaddr.h`, y por valor predeterminado tiene *0xc0000000* (*3 GB*). El resto de las direcciones virtuales que ocupa el kernel tiene como rango `PHYS_BASE` hasta *4 GB*.

La memoria virtual del kernel es global. Por lo que siempre esta mapeado de la misma forma, esto sin importar el proceso o el *thread* que este corriendo en el procesador. En Pintos, la memoria virtual del kernel esta mapeada una a una con la memoria física, empezando en la posición `PHYS_BASE`. Es decir, que las direcciones de kernel comienzan desde *0*, por lo que el acceder a la dirección *0x1234*, se mapea a *0 + 0x1234*, la cual es *0x1234*.

Un programa de usuario sólo puede acceder a su propio espacio de memoria virtual. Si un proceso de usuario intenta acceder a una dirección de kernel resulta en una interrupción(*trap*) llamada *page fault*, el cual es manejado por la función `page_fault()` definida en `userprog/exception.c`. El proceso de usuario sera terminado como resultado. Los *threads* del kernel por otro lado, pueden acceder tanto a la memoria virtual del kernel como a la memoria del proceso de usuario, sin embargo, incluso si el kernel intenta acceder a memoria que no esta mapeada a un proceso de usuario este fallara con un *page fault*.

Conceptualmente, cada proceso es libre de utilizar su memoria de usuario como lo desee. En la practica, la vista de la memoria virtual del proceso de usuario se ve de la siguiente manera:

```
1   PHYS_BASE +----------------------------------+
2             |             user stack           |
3             |                  |               |
4             |                  |               |
5             |                  V               |
6             |          grows downward          |
```



En el proyecto 2 de Pintos, el *stack* del proceso de usuario tiene un tamaño fijo, pero en el proyecto 3 se le permitirá crecer a este *stack*. Tradicionalmente, el tamaño de la sección *uninitialized data segment* puede ser ajustado por una llamada al sistema (*system call*), aunque tú no tendrás que implementar esta funcionalidad.

El segmento que contiene el código del proceso de usuario en Pintos empieza en la dirección virtual *0x08048000*, aproximadamente a 128MB desde la última dirección física. Este valor esta definido en el manual del procesador SySV-i386 y no tiene mayor significado.

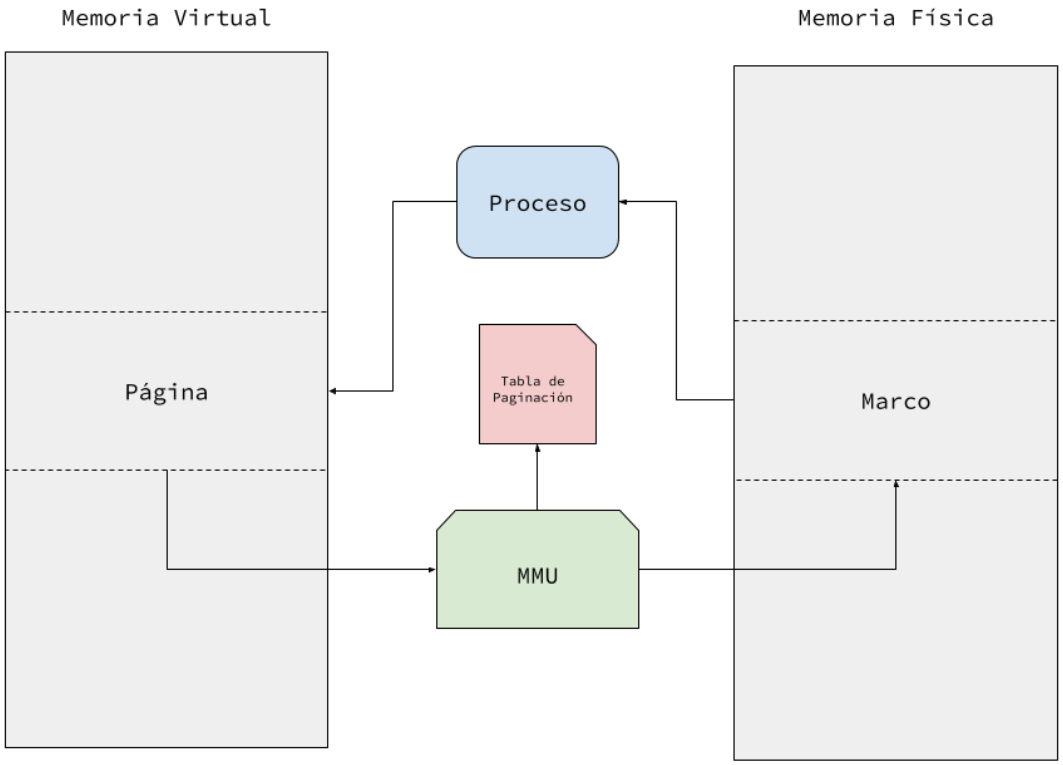
2.2. Paginación

La estrategia mas sencilla para crear la tabla de paginación sería tener una entrada por cada dirección virtual, la cual apuntaría a la dirección física. El problema radica en que necesitamos un espacio igual al tamaño del espacio virtual para almacenar la tabla, por ejemplo, si estamos en una arquitectura de 32 bits, necesitaríamos al menos 4GB para almacenar la tabla de paginación de un solo proceso. Es indispensable utilizar un mecanismo más eficiente, el problema con el esquema anteriormente planteado es que se están haciendo dos suposiciones:

1. Un proceso siempre tiene asociado todo su espacio virtual en direcciones físicas.
2. Una dirección virtual se puede asociar a cualquier dirección física.

El primer punto en realidad si puede ocurrir, pero lo que se puede hacer es considerar un esquema en el que las entradas solamente se agreguen cuando las direcciones se activen. El segundo punto es el más importante, si restringimos la forma de asignar direcciones virtuales con direcciones físicas se puede ahorrar espacio de almacenamiento. Es aquí donde la paginación surge como una solución.

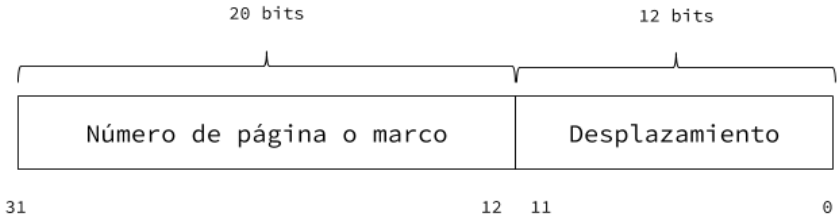
La paginación es un mecanismo de manejo de memoria virtual, consiste principalmente en dividir los espacios de memoria, tanto virtual como física, en bloques de memoria de tamaño fijo. En lugar de asociar dirección por dirección, la idea es asociar un bloque de memoria virtual en un bloque de memoria física. A los bloques de memoria virtual los llamaremos páginas (*page*), mientras que a los bloques de memoria física los llamaremos marcos (*frame*).



La principal ventaja de emparejar bloques de memoria en lugar de direcciones es que la tabla de paginación se reduce significativamente. La estrategia es guardar en la tabla de paginación los emparejamientos entre páginas y marcos, después, para obtener la dirección final se puede suponer que la posición relativa de la dirección en la página es la misma que en el marco. Si suponemos que estamos en un espacio de direcciones de 32 bits, en lugar de necesitar 2^{32} entradas en la tabla, solo necesitamos 2^{32} / N entradas, donde N es el tamaño de la página en bytes.

Es necesario señalar que el tamaño de las páginas (que es el mismo tamaño que el de los marcos), es muy importante. Si el valor es muy pequeño la tabla de paginación tendrá más entradas, si es muy

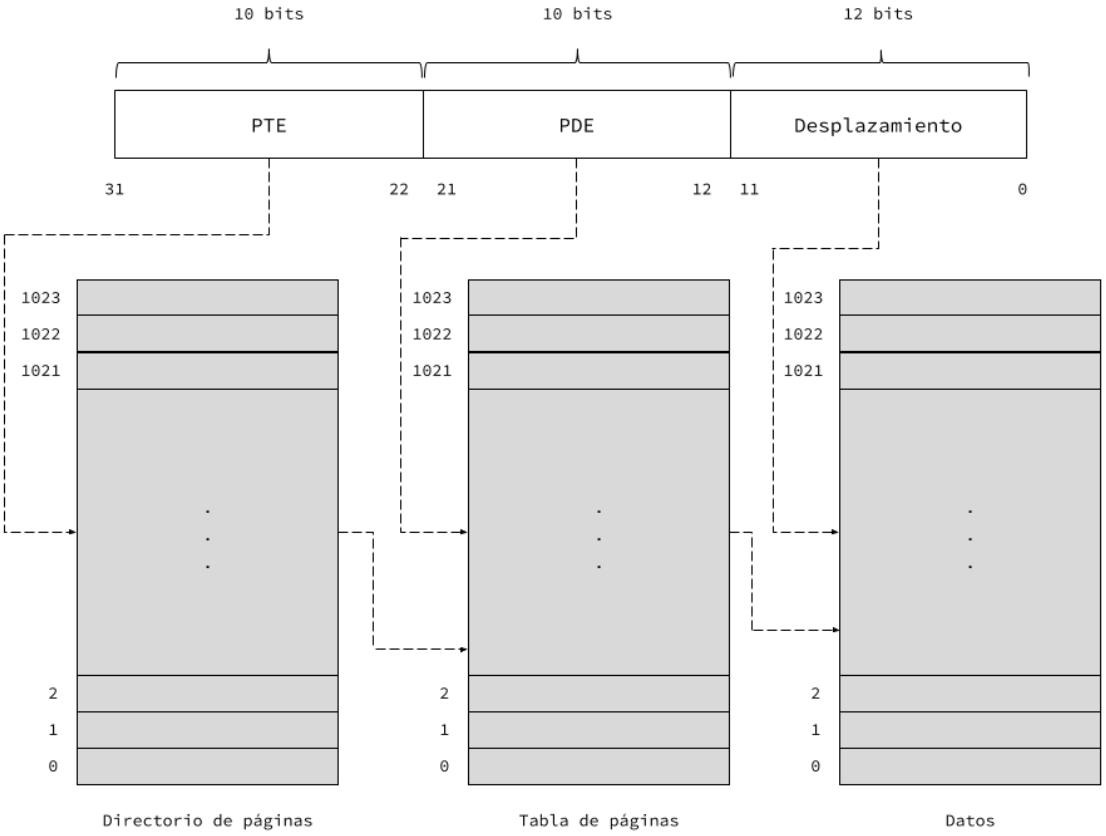
grande se corre el riesgo de desperdiciar memoria (*framentation*), pues uno de los inconvenientes de la paginación es el mínimo de memoria que se puede asignar a un proceso es una página. En la mayoría de los sistemas operativos de 32 bits el tamaño de la pagina es de 4096 bytes (4KB), lo mismo ocurre en el caso de Pintos. Como se puede notar, el tamaño de la página es divisor de 2^{32}, con lo cual garantizamos que el espacio virtual se cubra sin dejar huecos. Con un tamaño de página de 4 KB una dirección virtual se puede segmentar en dos secciones: el número de la pagina (20 bits) y el desplazamiento (12 bits). Al igual que la dirección virtual, la dirección física tiene el mismo formato: el número de marco (20 bits) y el desplazamiento (12 bits).



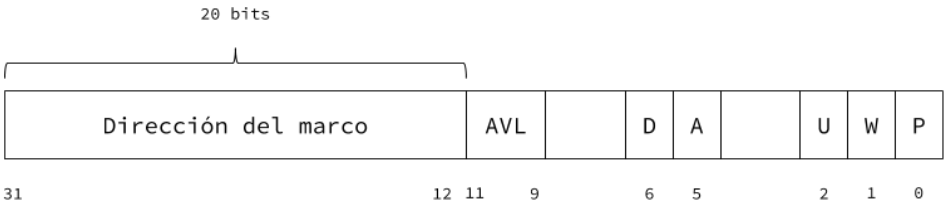
2.3. Tabla de paginación

Como se mencionó anteriormente, la tabla de paginación es una estructura que permite al dispositivo MMU traducir direcciones virtuales en direcciones físicas. La tabla de paginación se debe almacenar en la memoria principal y debe contener una entrada por cada una de las páginas que el proceso tiene activas.

Con el objetivo de reducir aun más el tamaño de la tabla de paginación y considerando que en el caso promedio los procesos usualmente tienen pocas páginas activas, es posible segmentar la dirección virtual en un nivel más. La idea es tener dos niveles de paginación al primero lo llamaremos directorio de páginas (*Page Directory (PD)*), el cual es un arreglo de 1024 (10 bits) entradas llamadas PDE. Cada entrada redirige a una nueva estructura llamada tabla de páginas (*Page Table (PT)*), la cual también es un arreglo de 1024 (10 bits) entradas llamadas PTE. Estas entradas son las que apuntan al marco físico.



Podemos observar que el último nivel de la estructura es la que debe direccionar al marco (memoria física), con este valor y el desplazamiento podemos obtener la dirección física completa. Podemos observar que el número de marco se puede direccionar con 20 bits, como cada entrada del ultimo nivel tiene 32 bits, tenemos 12 bits sobrantes. Estos 12 bits se utilizan para almacenar información adicional de la página y el marco.



1. **AVL**: Espacio disponible si el sistema operativo necesita almacenar más información.
2. **D**: Es conocido como *Dirty bit*, este bit tiene valor 1 cuando se ha realizado una operación escritura en el marco.
3. **A**: Es conocido como *Accessed bit*, este bit tiene valor 1 cuando se ha realizado una operación de lectura o escritura en el marco.
4. **U**: Cuando es 1, el proceso de usuario tiene acceso a la página/marco. Cuando es 0, solo el kernel tiene acceso.
5. **W**: Cuando es 1, la página/marco es de lectura/escritura, cuando es 0 es solo lectura.
6. **P**: Cuando es 0, cualquier intento de acceso a la página/marco genera un fallo de página.

2.4. Tabla de paginación en Pintos

Una vez que se tiene definido de forma abstracta como funciona la memoria virtual y la paginación, el siguiente paso es llevar a cabo una implementación. Un punto muy importante a tener en cuenta al implementar paginación es que el sistema operativo (el kernel) también se ejecuta en modo virtual. Es decir el kernel tampoco tiene acceso directo al espacio de direcciones físicas. Por lo cual se tienen que tener dos consideraciones:

1. Tener una forma de administrar la memoria física de forma indirecta.
2. La inicialización del sistema operativo debe crear un entorno de memoria virtual para el kernel.

En el caso de Pintos ambos puntos se solucionan al inicializar el sistema. Para el segundo punto se hace en dos momentos de la inicialización, primero, cuando el procesador aún esta en modo real, se crea una tabla de paginación temporal donde se hace un mapeo uno a uno de la memoria física y al virtual. La tabla temporal es solo para que el kernel tenga acceso a toda la memoria principal una vez que el procesador cambia a modo virtual, el código encargado de crear la tabla temporal se encuentra en el archivo *src/threads/start.S*.

Una vez que el procesador está en modo virtual se procede a crear la tabla de paginación definitiva, lo cual se implementa en el archivo *src/threads/init.c* en la función *paging_init*, que se muestra a continuación:

```
1  static void
2  paging_init (void)
3  {
4    uint32_t *pd, *pt;
5    size_t page;
6    extern char _start, _end_kernel_text;
7
8    pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
9    pt = NULL;
10   for (page = 0; page < init_ram_pages; page++)
11     {
12       uintptr_t paddr = page * PGSIZE;
13       char *vaddr = ptov (paddr);
14       size_t pde_idx = pd_no (vaddr);
```

```
15        size_t pte_idx = pt_no (vaddr);
16        bool in_kernel_text = &_start <= vaddr && vaddr < &_end_kernel_text;
17
18        if (pd[pde_idx] == 0)
19          {
20            pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
21            pd[pde_idx] = pde_create (pt);
22          }
23
24        pt[pte_idx] = pte_create_kernel (vaddr, !in_kernel_text);
25      }
26
27    asm volatile ("movl %0, %%cr3" : : "r" (vtop (init_page_dir)));
28  }
```

El primer paso importante ocurre en la línea 8, donde se aparta espacio para el directorio de páginas (4096 bytes o 1024 entradas de 32 bits). Luego se inicia un ciclo para cada una de la páginas o marcos en la memoria física (línea 10). En la líneas 20 y 21 se crea la tabla de paginación (PT) y se inicializa la entrada en el directorio de paginación (PD). Luego, se agrega la dirección virtual a la tabla de paginación (PT) en la línea 24, por medio de la función *pte_create_kernel*, la cual marca la página como accesible únicamente por el kernel. Por último, en la línea 27 le indicamos al procesador que utilice la nueva tabla de paginación, lo cual se hace escribiendo la dirección de la tabla en el registro CR3.

Otra parte que se resuelve en el código anterior es el acceso del kernel a toda la memoria física por medio de la memoria virtual. La idea es mapear toda la memoria física en el espacio virtual del kernel que comienza en la dirección *0xc0000000* (`PHYS_BASE`). Lo anterior se hace al calcular la dirección virtual de la memoria física con la función *ptov* en la línea 13. La función se encuentra definida en el archivo *src/threads/vaddr.h*:

```
1 static inline void *
2 ptov (uintptr_t paddr)
3 {
4   ASSERT ((void *) paddr < PHYS_BASE);
5
6   return (void *) (paddr + PHYS_BASE);
7 }
```

Como podemos observar lo único que hacemos es sumar `PHYS_BASE` (*0xc0000000*) a la dirección física para obtener la dirección virtual. De esta forma estamos mapeando toda la memoria física en el espacio virtual del kernel. Este mecanismo es muy sencillo de implementar, sin embargo, solo funciona cuando el tamaño de la memoria física es pequeño, pues solo podemos mapear hasta 1GB de memoria física.

2.5. Espacio virtual de los procesos de usuario

Como se mencionó anteriormente, una de la principales ventajas de la memoria virtual es poder aislar a los procesos de usuario. El aislamiento es con fines de seguridad, por ello, es necesario que el espacio virtual de los procesos de usuario este aislado entre los mismos procesos de usuario, así como también del espacio virtual del kernel. El espacio virtual de un proceso de usuario empieza en la dirección 0 y termina en la dirección `PHYS_BASE` (*0xc0000000*), a partir de dicha dirección comienza el espacio del kernel.

Para poder aislar los espacios virtuales de los procesos de usuario se requiere una tabla de paginación diferente para cada procesos de usuario. Sin embargo como los hilos de kernel necesitan tener acceso a toda la memoria, la tabla de paginación debe ser diferente para los procesos de usuario pero no para los hilos de kernel. El problema radica en que el espacio virtual de un proceso es contiguo al espacio del kernel, lo cual aplica para todos los procesos de usuario.

Pintos soluciona el problema anterior utilizando una tabla de paginación base, que contiene el espacio de direcciones del kernel. Cada que se crea un nuevo proceso de usuario se copia la tabla y se agregan las páginas necesarias para el proceso nuevo. En el archivo *userprog/process.c*, en la función *load()* se crea la nueva tabla de paginación usando la función *pagedir_create*, definida en el archivo *userprog/pagedir.c*:

```
1 uint32_t *
```

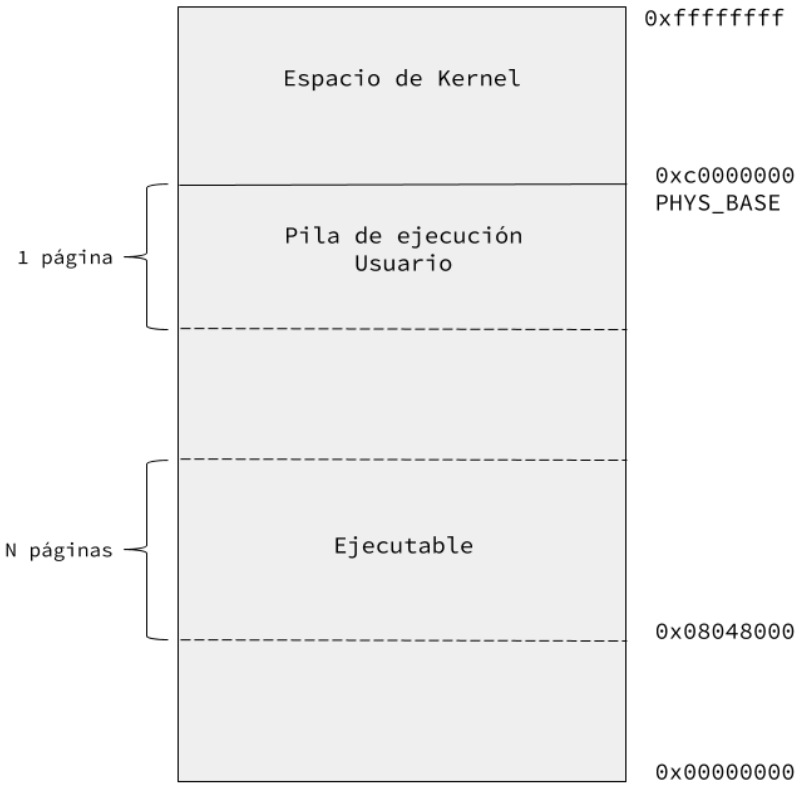
```
2  pagedir_create (void)
3  {
4    uint32_t *pd = palloc_get_page (0);
5    if (pd != NULL)
6      memcpy (pd, init_page_dir, PGSIZE);
7    return pd;
8  }
```

A grandes rasgos, lo que se hace es solicitar memoria para el nuevo directorio de paginación (línea 4), para después copiar el contenido del directorio de páginas base (línea 6). Al copiar la tabla base garantizamos que el nuevo hilo de kernel tenga acceso a todo el espacio físico (que está mapeado uno a uno en el espacio del kernel).

Una vez que se tiene esta base, se procede a agregar las páginas requeridas para el nuevo proceso de usuario. En el caso de Pintos son las páginas necesarias para cargar el ejecutable del proceso de usuario y las páginas necesarias para la pila de ejecución del mismo proceso. Ambas regiones se activan utilizando la función *install_page()*, la cual es prácticamente un envoltorio para *pagedir_set_page()*:

```
1  bool
2  pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
3  {
4    uint32_t *pte;
5
6    ASSERT (pg_ofs (upage) == 0);
7    ASSERT (pg_ofs (kpage) == 0);
8    ASSERT (is_user_vaddr (upage));
9    ASSERT (vtop (kpage) >> PTSHIFT < init_ram_pages);
10   ASSERT (pd != init_page_dir);
11
12   pte = lookup_page (pd, upage, true);
13
14   if (pte != NULL)
15     {
16       ASSERT ((*pte & PTE_P) == 0);
17       *pte = pte_create_user (kpage, writable);
18       return true;
19     }
20   else
21     return false;
22 }
```

En las primeras líneas (6 a 10), verificamos que los argumentos son correctos. En la línea 12 buscamos la entrada (PDE) para la página UPAGE, en el directorio de paginación, el tercer argumento (TRUE) indica que si la entrada es vacía (apunta a NULL), se debe crear una tabla de paginación (PT) y asociarla a la entrada (PDE). Por último en la línea 17 asociamos el marco (KPAGE) a la entrada y marcamos todas la direcciones con permiso de escritura.



Una vez instaladas la páginas en el directorio de paginación, se necesita hacer que el nuevo proceso de usuario utilice la nueva tabla de paginación. Para ello se utiliza la función *process_activate()*, la cual invoca la función *pagedir_activate()*, que esencialmente cambiar el valor del registro CR3 para que apunte a la nueva tabla de paginación:

```
1 void
2 pagedir_activate (uint32_t *pd)
3 {
4   if (pd == NULL)
5     pd = init_page_dir;
6   asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
7 }
```

3. Forma de Trabajo

Pasos a seguir para trabajar en esta práctica:

1. Debes de tener una solución funcional para la práctica anterior.
2. A partir de la rama que contiene la solución de la práctica anterior crea una nueva rama llamada `proyecto02/invalid-memory-access` y posiciónate en ella.
3. Reemplaza el archivo `.github/workflows/run-tests.yml` con el que entregamos junto con este documento.
4. Agrega los archivos `validate-user-memory.*` en el directorio `src/userprog`.
5. Reemplaza el archivo `src/Makefile.build` con el que entregamos junto con este documento.
6. Agrega los archivos `execute-tests-invalid-memory-access` y `execute-tests-rox` en el directorio `src/userprog`. Dale permiso de ejecución a estos archivos con `chmod +x <PATH_TO_FILE>`.

7. Agrega al repositorio en el directorio raíz el archivo `designdoc-invalid-memory-access.md`.

8. Crea un *commit* de estos cambios y sube (*push*) la rama al repositorio.

9. Crea un *Pull Request* para la rama `proyecto03/invalid-memory-access` hacia la rama que contiene la solución a la práctica anterior. Este *Pull Request* debe de contener 7 archivos en la sección de *Files changed*, que son exactamente los que introdujiste en los pasos anteriores.

10. En la sección de *Actions* debes de seguir pasando las pruebas de *Argument Passing* y *Syscalls Processes*, *Syscalls File System* y *File System*.

11. Una nota importante es que para pasar las pruebas vamos a utilizar la máquina virtual `qemu`, pues `boch` puede tener algunos problemas.

En el archivo `readme.md` del repositorio se incluyen instrucciones de cómo compilar el código y de cómo ejecutar algunas pruebas. Para ejecutar todas las pruebas de esta práctica, puedes acceder al contenedor, posicionarte en el directorio `src/userprog` y ejecutar la instrucción
`bash execute-tests-invalid-memory-access` y `bash execute-tests-rox`.

Tu solución tiene que ser incremental, es decir, debes de pasar las pruebas de la práctica anterior y también las pruebas específicas de esta práctica, de hecho en la sección de *Actions* se van a ejecutar como pasos independientes.

4. Actividades

4.1. Accediendo a la Memoria de Usuario

En la mayoría de las llamadas al sistema, el kernel tiene que acceder a la memoria del proceso de usuario a través de apuntadores que vienen en el *interrupt frame*. Hay que tener cuidado con la manipulación de dichos apuntadores, el proceso de usuario podría pasar un apuntador nulo, un apuntador sin mapear en la memoria virtual, o un apuntador hacia la memoria virtual del kernel (mayor a `PHYS_BASE`). Todos estos tipos de apuntadores deberán ser tratados como inválidos y no afectar la memoria del kernel u otros procesos de usuario. El proceso que intente hacerlo deberá ser terminado y sus recursos liberados.

Existen 2 posibles soluciones para evitar apuntadores inválidos en llamadas al sistema. La primera es verificar y validar cada apuntador que el proceso de usuario provea y después desreferenciarlo, si escoges este camino, tendrás que revisar las funciones dentro de `userprog/pagerdir.c` y `threads/vaddr.h`. Esta es la solución más simple.

El segundo método es solo verificar que el apuntador de un proceso de usuario sea mayor a `PHYS_BASE`, y después desreferenciarlo. Un apuntador inválido causará un *page fault* que será manejado por la función `page_fault()` en `userprog/exception.c`. Esta técnica es normalmente más rápida porque usa la *unidad de manejo de memoria* (*MMU*), y por lo tanto es la técnica más usada en *kernels* en el mundo real (esto incluye a Linux).

En cualquier caso, tu trabajo es asegurarte de que no existan fugas de memoria (*memory leaks*). Por ejemplo, supón que una llamada al sistema adquiere un *lock* o asigna memoria usando *malloc()*. Si durante el procesamiento de una llamada al sistema se encuentra con que un apuntador es inválido, es importante liberar el *lock* o la memoria recientemente asignada. Si escogiste la primera solución (verificar los apuntadores antes de comenzar con la llamada al sistema), este problema no sucederá. Pero si decidiste ocupar la segunda solución entonces manejar esta situación es un poco más difícil, por lo que se les proporcionan las siguientes funciones como ayuda:

```
1 /* Reads a byte at user virtual address UADDR.
2    UADDR must be below PHYS_BASE.
3    Returns the byte value if successful, -1 if a segfault
4    occurred. */
5 static int
6 get_user (const uint8_t *uaddr)
```

```
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

Cada una de estas funciones asume que las direcciones *uaddr* pertenecen a un proceso de usuario, es decir, que son menores a `PHYS_BASE`. También asumen que modificaste *page_fault()*, de tal forma que durante un *page fault*, la función asigna *0xffffffff* al registro *eax* y copia el valor anterior de *eax* en el registro *eip*.

4.1.1. Pruebas (5 pts)

- `sc`: sc-bad-sp sc-bad-arg sc-boundary sc-boundary-2 sc-boundary-3
- `bad`: bad-read bad-write bad-read2 bad-write2 bad-jump bad-jump2
- `syscalls`: create-bad-ptr open-null open-bad-ptr create-null read-bad-ptr write-bad-ptr exec-bound-2 exec-bound-3 exec-bad-ptr wait-killed

4.1.2. Pruebas Extra (1 pts)

- `robustness`: no-vm/multi-oom

Hay que descomentar la prueba en el archivo `execute-tests-invalid-memory-access`. En realidad si tus implementaciones son correctas deberías de pasar esta prueba casi de gratis, con cambiar unas cuantas lineas en la función `start_process` en `process.c` es suficiente.

4.2. Denegando Escrituras a Ejecutables

Añade código para denegar las escrituras a los archivos que están siendo usados como ejecutables. Es común en las implementaciones del Sistemas Operativos bloquear las escrituras a estos ejecutables en uso por los resultados impredecibles que pueda tener si un proceso intenta correr código que estaba siendo guardado en disco. Esto será especialmente importante cuando tengamos la memoria virtual implementada en el proyecto 3. Por ahora no dará ningún problema.

Puedes usar `file_deny_write()` para prevenir escrituras a un archivo abierto. Usando la función `file_allow_write()` en el archivo permitirá escrituras otra vez en este (a menos de que otro proceso bloque escrituras a un ejecutable). Si un archivo es cerrado también deberá activar otra vez las escrituras. Por lo tanto, para denegar escrituras a un archivo, deberás de mantenerlo abierto durante la vida del proceso.

4.2.1. Pruebas (2 pts)

- rox-simple
- rox-child
- rox-multichild

4.3. Documento de Diseño (3 pts)

Contestar todas las sección del documento de diseño que se entrega junto con este documento.

5. Consejos

Primero ajusta la función `page_fault()` en `userprog/exception.c` de tal manera que si ocurre un *page fault* en modo usuario el proceso es terminado con un `exit_code` de `-1`. Con esto deberías de poder pasar el conjunto de pruebas `bad`.

Luego procede a implementar la validación de los argumentos de las *system calls* una por una, siempre verificando que sigues pasando las pruebas de esa syscall de acuerdo a los scripts de las prácticas pasadas y también verificando que empiezas a pasar algunas pruebas de la nueva práctica. Por ejemplo empieza por las más simples que reciben solamente un único argumento y que dicho argumento no es un apuntador, por ejemplo exit, wait, etc. Para las syscalls que reciben apuntadores, debes de validar que dichos apuntadores sean válidos, es decir, que no sean nulos, que apunten a direcciones que se encuentren por debajo de `PHYS_BASE` en su totalidad (sobre todo si recibes strings o buffers) y que dichas direcciones estén mapeadas en el *pagedir* del proceso de usuario que las proporcionó. La recomendación es implementar las funciones del archivo `userprog/validate-user-memory.c` y que estas las utilices en `userprog/syscall.c` para validar todos los apuntadores y valores que recibes del usuario, incluído el *stack pointer* que recibes en el interrupt frame (aka `esp`).

Las pruebas `sc` estresan que tu implementación de syscalls valide el *stack pointer* que recibe el syscall handler.

El resto de las pruebas de syscalls estresan que valides los argumentos que estas reciben, en especial los apuntadores.

Para pasar multi-oom, es necesario que primero pases todas las demás pruebas, y luego que hagas unos ajustes mínimos en `start_process`, para que siempre un proceso ponga su nombre correcto antes de terminar con un error.

6. Entregables

En Google Classroom agrega el enlace del `Pull Request` que contiene tu solución y en donde se deben de observar los resultados de Github Actions al correr las pruebas. También marca la tarea como entregada en Google Classroom.