Boletín xv6-3: El sistema de memoria de xv6. Reserva de páginas bajo demanda

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2021/2022

El Esquema de Paginación y Segmentación en xv6

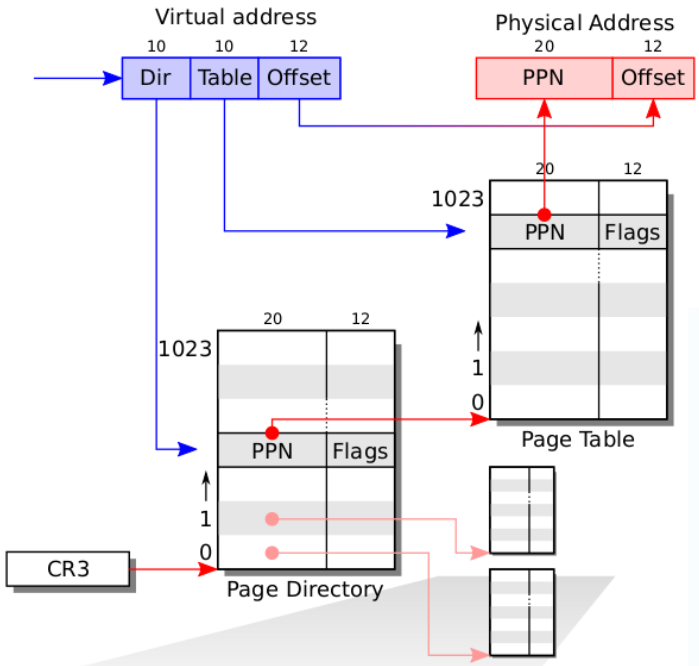
- xv6 usa la paginación para multiplexar el espacio de direcciones y como mecanismo de protección
- También utiliza algunos trucos:
  - Mapea la misma memoria (el kernel) en varios espacios de direcciones
  - Mapea la misma memoria más de una vez en un espacio de direcciones (cada página de usuario también se mapea en la visión que el kernel tiene de la memoria física)
  - Y protege contra desbordamientos de la pila de usuario con una página no mapeada
- En el resto de la sección explicaremos el uso que xv6 hace del esquema de paginación y segmentación proporcionado por el hardware
- Xv6 define en el fichero `mmu.h` todas las estructuras relacionadas con la gestión de la memoria
  - Segmentos (incluido el *task state segment*), directorio y tabla de páginas y manejo de interrupciones

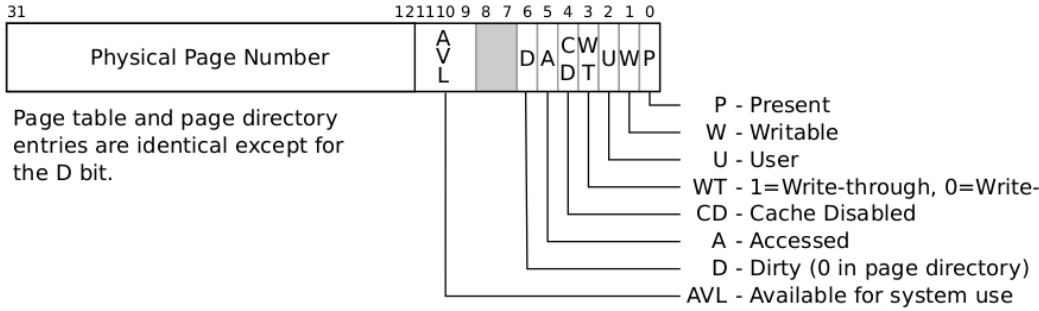
El Esquema de Paginación y Segmentación en xv6 (cont.)

- Se definen los siguientes segmentos (`mmu.h`):

```
#define SEG_KCODE 1 // kernel code
#define SEG_KDATA 2 // kernel data+stack
#define SEG_UCODE 3 // user code
#define SEG_UDATA 4 // user data+stack
#define SEG_TSS   5 // this process's task state
```

- En cuanto a la paginación, el esquema utilizado es el ya visto anteriormente:





El Esquema de Paginación y Segmentación en xv6 (cont.)

- De los segmentos anteriormente indicados, el `SEG_TSS` denominado **segmento de pila del proceso**, es de especial importancia
- Este segmento es accesible a través del registro de tarea (`tr`), almacenando la siguiente información acerca de la tarea:
  - Estado de los registros del procesador
  - Permisos sobre los puertos de E/S
  - Punteros de pila a usar en los modos inferiores
  - Enlace al TSS anterior
- Por razones de seguridad, el TSS debe estar en una zona de memoria que sólo sea accesible por el kernel
- El TSS contiene 6 campos que especifican el nuevo puntero de pila cuando se produce un cambio de nivel de privilegio
  - El campo `ss0` contiene el selector de segmento de pila para PLC=0
  - Cuando se produce una interrupción en modo protegido, la CPU busca el el TSS el valor de `ss0` y `esp0` y carga dichos valores en `ss` y `esp`, respectivamente

El Esquema de Paginación y Segmentación en xv6 (cont.)

  - Esto permite al kernel usar una pila diferente a la del programa en modo usuario
  - Esta pila es única para cada programa
- En xv6 dicha estructura está definida en `mmu.h`:

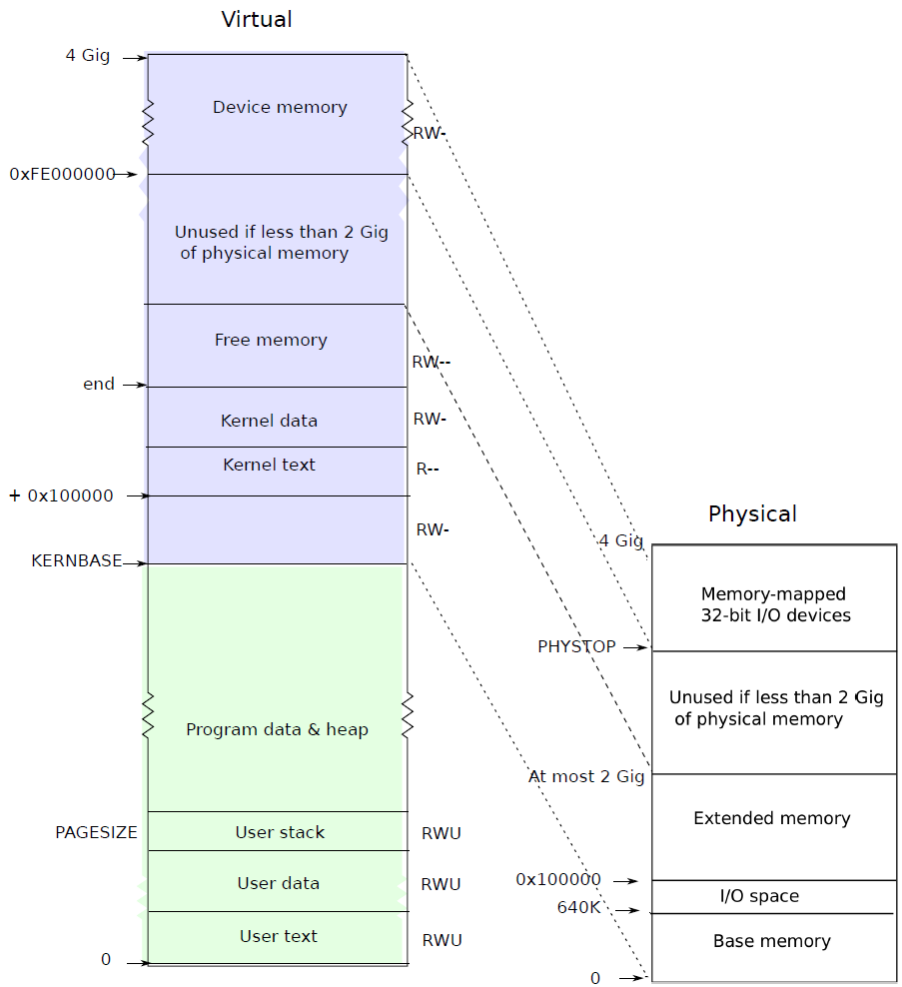
```
// Task state segment format
struct taskstate {
  uint link;         // Old ts selector
  uint esp0;         // Stack pointers and segment selectors
  ushort ss0;        //   after an increase in privilege level
  ushort padding1;
  uint *esp1;
  ushort ss1;
  ushort padding2;
  uint *esp2;
  ushort ss2;
  ushort padding3;
  void *cr3;         // Page directory base
  uint *eip;         // Saved state from last task switch
  uint eflags;
  uint eax;          // More saved state (registers)
  uint ecx;
  uint edx;
  uint ebx;
  uint *esp;
  uint *ebp;
  uint esi;
```

El Esquema de Paginación y Segmentación en xv6 (cont.)

```
  uint edi;
  ushort es;          // Even more saved state (segment selectors)
  ushort padding4;
  ushort cs;
  ushort padding5;
  ushort ss;
  ushort padding6;
  ushort ds;
  ushort padding7;
  ushort fs;
  ushort padding8;
  ushort gs;
  ushort padding9;
  ushort ldt;
  ushort padding10;
  ushort t;           // Trap on task switch
  ushort iomb;        // I/O map base address
};
```

El Esquema de Paginación y Segmentación en xv6 (cont.)

- La tabla de páginas creada en `entry` permite empezar la ejecución del kernel tal como se vió en la práctica sobre el arranque de xv6
- En `main()` se cambia a una nueva tabla de páginas llamando a `kvmalloc()`
- Cada proceso tiene una tabla de páginas separada. El fichero `memlayout.h` describe el *layout* de la memoria en xv6
  - Dicha tabla mapea todo el kernel, lo que permite no cambiar de tabla de páginas al cambiar de modo



Creación del espacio de direcciones

- `main()` llama a `kvmalloc()` (en `vm.c`) que crea una tabla de páginas con los mapeos necesarios por encima de KERNBASE para que se ejecute el kernel

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

- La mayor parte del trabajo ocurre en `setupkvm()`
  - Primero reserva una página para guardar el directorio de páginas

```
pde_t* setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  ...
```

Creación del espacio de direcciones (cont.)

- Después llama a `mappages()` que instala las traducciones que necesita el kernel

```
  ...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

- Dichas traducciones son:

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

Creación del espacio de direcciones (cont.)

- `setupkvm()` no realiza ningún mapeo para la memoria de usuario (esto ocurrirá después)

- El código de la función `mappages()` es el siguiente:

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

Creación del espacio de direcciones (cont.)

- donde la función `walkpgdir()` devuelve la entrada en la tabla correspondiente a la dirección virtual indicada reservando, si `alloc!=0`, las páginas físicas necesarias:

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

Creación del espacio de direcciones (cont.)

- Por último, cambiamos a la tabla de páginas del kernel:

```
// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}
```

- La creación del espacio de direcciones de usuario se realiza a través de la llamada `exec()` tal como veremos más adelante

Reserva de la memoria física

- Xv6 usa la memoria física entre el fin del kernel y `PHYSTOP` para la reserva de páginas (de 4KB) en tiempo de ejecución
- Utiliza una lista enlazada para llevar la cuenta de que páginas están libres
- La inicialización de la gestión de la memoria física ocurre en dos fases:
  1. `main()` llama a `kinit1()` cuando todavía está utilizando `entrypgdir` para añadir a la lista de libres las páginas mapeadas por `entrypgdir` y no usadas tras la carga del kernel:

```
// En main.c (función main):
kinit1(end, P2V(4*1024*1024)); // phys page allocator

// En kalloc.c
void kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
```

Reserva de la memoria física (cont.)

2. `main()` llama a `kinit2()` con el resto de páginas físicas después de instalar la tabla de página definitiva y mapearlas en todos los *cores*

```
// En main.c (función main):
kinit2(P2V(4*1024*1024), P2V(PHYSTOP));

// En kalloc.c
void kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

- Ambas funciones se construyen sobre `freerange`, que libera un rango de direcciones virtuales, devolviendo las mismas a la lista de libres:

```
void freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

Reserva de la memoria física (cont.)

- La reserva y liberación de una página física se realizan a través de `kalloc()` y `kfree()`. Ambas utilizan una lista de páginas físicas libres en memoria protegidas por un candado:

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

Reserva de la memoria física (cont.)

- El código de **kalloc()** es:

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

Reserva de la memoria física (cont.)

- Mientras que el código de **kfree** es el siguiente:

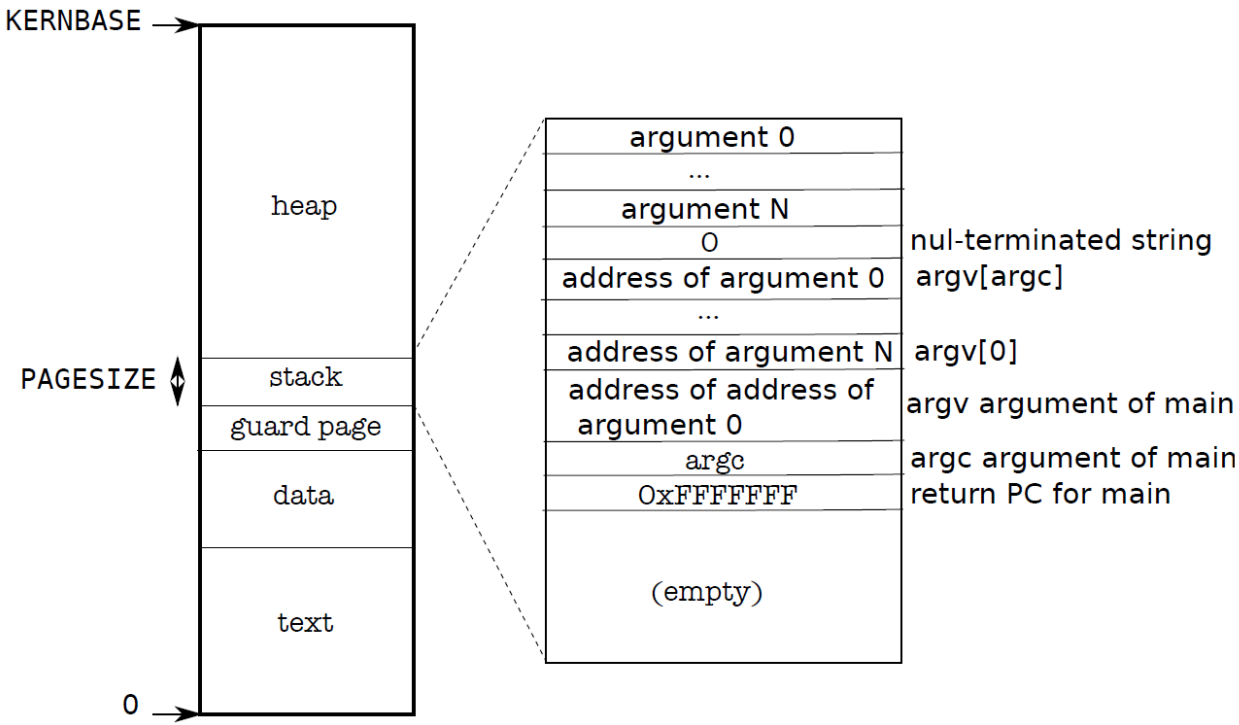
```
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

Reserva de la memoria física (cont.)

- Los propios marcos de página se utilizan como lugar de almacenamiento de la lista enlazada de marcos libres. Al hacer la asignación del puntero a una estructura `struct run`, los primeros bytes del marco se utilizan para puntero al siguiente marco libre

La llamada al sistema exec()

- La llamada al sistema **exec()** es la encargada de crear la parte de usuario de un espacio de direcciones
- En la siguiente figura se muestra el *layout* de la memoria de usuario de un proceso en ejecución



La llamada al sistema `exec()` (cont.)

- La zona *heap* está encima de la pila y puede expandirse llamando a `sbrk()`, mientras que la pila consta de una única página
- xv6 pone una página de guarda (mapeada pero no accesible) debajo de la pila
  - Así si la pila se desborda, el hardware generará una excepción
- La llamada `exec()` inicializa dicho espacio de usuario a partir de un fichero almacenado en el sistema de ficheros que se abre usando `namei`

```
int exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;

  begin_op();
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ...
```

La llamada al sistema `exec()` (cont.)

- `begin_op()` y `end_op()` sirven para marcar el comienzo y el fin de las operaciones en el sistema de ficheros en la bitácora
- A continuación se lee la cabecera ELF (definida en `elf.h`)
- Si la cabecera es válida `exec()` reserva una nueva tabla de página sin mapeos de usuario con `setupkvm()`

```
...
ilock(ip);
pgdir = 0;

// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
  goto bad;
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pgdir = setupkvm()) == 0)
  goto bad;
```

La llamada al sistema `exec()` (cont.)

- A continuación, reserva memoria para cada segmento ELF con `allocuvm()` y carga cada segmento en la memoria con `loaduvm()`

```
...
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;
...
```

La llamada al sistema `exec()` (cont.)

- La función `allocuvm()` se encarga de reservar las entradas en la tabla de páginas y las páginas físicas necesarias para que el proceso crezca de tamaño. Su código, que se encuentra en `vm.c` es:

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
```

La llamada al sistema `exec()` (cont.)

```
      return 0;
    }
  }
  return newsz;
}
```

- Por otro lado, la función `loaduvm()` utiliza `walkpgdir()` para encontrar la dirección física en donde escribir cada página del segmento ELF y `readi()` para leer la página del fichero (no se muestra)
- A continuación `exec()` crea e inicializa la pila de usuario (una única página) copiando los argumentos y la dirección de retorno

```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
```

La llamada al sistema exec() (cont.)

```
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

- exec() coloca una página inaccesible justo debajo de la página de pila
  - Los programas que intenten usar más de una página fallarán
  - Esta página también permite manejar argumentos demasiado largos
    - La función **copyout()**, utilizada para copiar los argumentos en la pila, se dará cuenta de que la dirección destino no es accesible y devolverá -1
- Si exec() detecta error salta a **bad** que libera la nueva imagen y devuelve -1

La llamada al sistema exec() (cont.)

- exec() espera a asegurar que la llamada al sistema tendrá éxito antes de liberar la imagen antigua. Una vez completa la imagen, puede instalarla y liberar la antigua devolviendo 0:

```
...
oldpgdir = proc->pgdir;
proc->pgdir = pgdir;
proc->sz = sz;
proc->tf->eip = elf.entry;  // main
proc->tf->esp = sp;
switchuvm(proc);
freevm(oldpgdir);
return 0;
```

La llamada al sistema `exec()` (cont.)

- Obsérvese la llamada a `switchuvm()` que conmuta el TSS y la tabla de páginas a la del proceso correspondiente:

```
void switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  cpu->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

Reserva de Páginas Bajo Demanda en xv6

- Una de las muchas optimizaciones que los SS.OO. implementan es la reserva bajo demanda de páginas en la memoria *heap* (*lazy allocation*)
- En `xv6`, las aplicaciones solicitan memoria al kernel a través de la llamada al sistema `sbrk()`
- En el kernel que estamos utilizando, `sbrk()` reserva las páginas físicas necesarias y las mapea en el espacio de direcciones virtuales del proceso
- Existen programas que reservan memoria pero nunca la utilizan (p.e. para implementar arrays dispersos de gran tamaño)
- Por esta razón los *kernels* suelen retrasar la reserva de cada página de la memoria hasta que la aplicación intenta usarla

Reserva de Páginas Bajo Demanda en xv6 (cont.)

- **EJERCICIO 1**: Implementa esta característica de reserva diferida en xv6. Para ello:
  1. Elimina la reserva de páginas de la llamada al sistema `sbrk()` (implementada a través de la función `sys_sbrk()` en `sysproc.c`)
     - La nueva función debe incrementar el tamaño del proceso (`proc->sz`) y devolver el tamaño antiguo pero no debe reservar memoria
     - No se debe llamar a `growproc()` en caso de que el proceso crezca
  2. Realiza las modificaciones, arranca xv6, y teclea `echo hola` en el *shell*:

```
init: starting sh
$ echo hola
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

Reserva de Páginas Bajo Demanda en xv6 (cont.)

3. El mensaje `pid 3 sh: trap...` proviene del manejador de *traps*, definido en `trap.c`
   - Ha capturado un fallo de página (trap 14 o `T_PGFLT`) que el kernel de xv6 no sabe cómo manejar
   - `addr 0x4004` indica la dirección virtual que generó el fallo de página
4. Modifica el código en `trap.c` para responder a un fallo de página en el espacio de usuario mapeando una nueva página física en la dirección que generó el fallo, regresando después al espacio de usuario para que el proceso continúe
   - Debes añadir tu código justo antes de la llamada a `cprintf()` que produce el mensaje
   - Para este primer paso, es suficiente con que permita al *shell* ejecutar comandos simples como `echo` y `ls`
   - Reutiliza código de `allocuvm()` en `vm.c` y usa `PGROUNDDOWN(va)` para redondear la dirección virtual a límite de página
   - Necesitarás llamar a `mappages()`. Para ello borra la declaración de función estática en `vm.c` y declara `mappages()` en `trap.c`

Reserva de Páginas Bajo Demanda en xv6 (cont.)

- La implementación realizada en el ejercicio 1 no es totalmente correcta ya que no contempla varias situaciones:
  - El caso de un argumento negativo al llamarse a `sbrk()`
  - Manejar el caso de fallos en la página inválida debajo de la pila
  - Verificar que `fork()` y `exit()/wait()` funciona en el caso de que haya direcciones virtuales sin memoria reservada para ellas
  - Asegurarse de que funciona el uso por parte del kernel de páginas de usuario que todavía no han sido reservadas (p.e., si un programa pasa una dirección de la zona de usuario todavía no reservada a `read()`)

NOTA: Para comprobar que todo funciona, los programas proporcionados `tsbrkX` deberán funcionar sin problemas, y también la ejecución de tuberías en el *shell* (p. ej. `ls | wc`)

Reserva de Páginas Bajo Demanda en xv6 (cont.)

- **EJERCICIO 2**: Modifica el código del primer ejercicio para que contemple las situaciones anteriormente descritas. Para ello tenga en cuenta la siguiente información extraida de
  `https://wiki.osdev.org/Exceptions#Page_Fault`

```
The Page Fault sets an error code:
 31              4               0
+---+--  --+---+---+---+---+---+---+
|   Reserved   | I | R | U | W | P |
+---+--  --+---+---+---+---+---+---+
        Length  Name    Description
P       1 bit   Present         When set, the page fault was caused by a page-protection
      violation. When not set, it was caused by a non-present page.
W       1 bit   Write   When set, the page fault was caused by a write access. When not
      set, it was caused by a read access.
U       1 bit   User    When set, the page fault was caused while CPL = 3. This does not
      necessarily mean that the page fault was a privilege violation.
R       1 bit   Reserved write  When set, one or more page directory entries contain
      reserved bits which are set to 1. This only applies when the PSE or PAE flags in
      CR4 are set to 1.
I       1 bit   Instruction Fetch       When set, the page fault was caused by an
      instruction fetch. This only applies when the No-Execute bit is supported and
      enabled.
```

Reserva de Páginas Bajo Demanda en xv6 (cont.)

- **EJERCICIO 3**: Modifica el código de la función `exec()` para que asigne a cada proceso un número de páginas de pila igual al número de páginas de código+datos de ese proceso. Compruebe que las anteriores modificaciones siguen funcionando

Reserva de Páginas Bajo Demanda en xv6 (cont.)

- **EJERCICIO 4**: Añade una llamada al sistema `freemem()` con la siguiente signatura:

```
1  int freemem(int type)
```

  - La llamada retornará:
    - Si `type` es igual a 0, el número de páginas de memoria física disponibles para asignar a los procesos
    - Si `type` es igual a 1, el número de bytes de memoria física disponibles para asignar a los procesos, que sería el número de la salida cuando `type` es igual a 0 multiplicado por el tamaño de página del sistema
  - También se definirán las constantes `F_PAGES` con valor 0, y `F_BYTES` con valor 1
  - La implementación se puede basar en las funciones `kalloc()` y `kfree()` del núcleo