xv6: Un SO simple para educación

xv6 es un sistema operativo simple, tipo UNIX, desarrollado en el curso *Operating System Enginnering* del **M.I.T** con el objetivo de usar un SO simple de entender pero realista en cursos de grado y posgrado.

En su versión mas actual se ha realizado un *port* a la arquitectura **risc-v**. En la página de *xv6* se encuentran las instrucciones para la descarga del código fuente, manuales técnicos y herramientas (toolchain) necesarias para su compilación y ejecución.

Para su compilación hay que instalar el *cross-compiler*[1](*risc-v toolchain*) tal como se indica en **xv6 tools**. Las herramientas usadas son básicamente **qemu** para usar como máquina virtual para plataforma risc-v, el GNU toolchain para risc-v: GCC (cross-compiler y linker), gdb (debugger) y *binutils* (objdump, y otros).

xv6: Descripción general

Es un SO simple, escrito en C con pocas líneas de assembly. El objetivo de su diseño e implementación es la facilidad de lectura y comprensión más que el rendimiento.

Sus principales características son:

- Diseño monolítico
- Es mono usuario, multitarea (*preemtive*).
- Soporta arquitecturas *multicore* o *multiprocesamiento simétrico (SMP)*.
- Ofrece una API que es un suconjunto del UNIX v6 tal como se describe en el capítulo **APIs**.

Diseño

En ésta sección se describen los distintos subsistemas y sus implementaciones (archivos fuente). En la carpeta `kernel` se encuentra la implementación del núcleo de xv6. En la carpeta `user` está el código de los *procesos de usuario* básicos con que viene acompañado, básicamente un conjunto mínimo de comandos, un *shell* de línea de comandos y un archivo de textos `README` .

En el directorio `kernel` el núcleo se implementa en varios módulos.

- **Parámetros del sistema**: Constantes sobre tamaños máximos de las estructuras de datos usadas (`param.h`).
- **Gestión de procesos**: Archivos `proc.c` , `exec.c` .
- **Gestión de la memoria**: Archivos `kalloc.c` , `vm.c`
- **Dispositivos de entrada-salida**: Soporta básicamente 2 dispositivos.

1. *Terminal*: (teclado+pantalla), conectada al puerto de comunicaciones serie **Universal Asynchronous Receiver/Transmitter**, implementado en `console.c` y `uart.c`.

2. Disco que soporta la interfaz *virtio* (`virtio_disk.c`).

   En el archivo `plic.c` se implementa el control del *riscv Platform Level Interrupt Controller (PLIC)* el cual permite controlar y *rutear* interrupciones de los dispositivos.

- **Sistema de archivos**: Basados en *i-nodes* y es *transaccional*. Es un diseño modular en capas con los siguientes sub-módulos:

  1. *Buffer caché* y entrada/salida de bloques: `bio.c`
  2. *Logging*: Gestión de transacciones (`log.c`)
  3. *I-nodes*, *directory* y *file paths*: `fs.c`
  4. *File descriptors*: `file.c`

- **Arquitectura**: Soporte para la arquitectura *risc-v* `riscv.h` . Allí se encuentran funciones y macros de bajo nivel para

  1. Control de CPUs y protección (niveles de ejecución)
  2. Protección y memoria virtual
  3. Control de interrupciones y excepciones

Build system

El *build system* se basa en `make` . El archivo Makefile contiene reglas para *construir* los diferentes componentes del sistema que básicamente consiste de dos productos:

- El ejecutable `kernel` , en formato ELF, es la imagen del ejecutable del *kernel* de xv6.

- La imagen del *sistema de archivos* `fs.img` en el formato del sistema de archivos implementado. Internamente se encuentran los archivos de usuario (programas y datos) y representa la *imagen del disco* para la máquina virtual *qemu*.

Al ejecutar el comando `make qemu` , se recompilan los componentes necesarios, se generan las imágenes y se invoca a `qemu-system-riscv64 kernel/kernel fs.img` . En este comando además se pasan opciones que indican el número de CPUs (cores) y la cantidad de memoria (RAM) a usar, entre otras (ver regla `qemu: $K/kernel fs.img` del `Makefile`).

Los programas de usuario (en el directorio `user`) se enlazan con los archivos objeto correspondientes a los archivos fuentes

- `ulib.c` : Funciones de manejo de strings y otros.
- `usys.S` : Llamadas al sistema.
- `printf.c` : Implementación de `printf()`
- `umalloc.c` : Manejador del heap (funciones `malloc(), free()`) en modo usuario.

Estos módulos serían la *biblioteca estándar* de xv6.

En las reglas de cada *target*, además de los archivos objetos, se usan las utilidades `objcopy` y `objdump` para extraer información como la tabla de símbolos y el código *assembly* de cada archivo objeto para su análisis.

Procesos

La estructura `struct proc` (definida en `kernel/proc.h`) representa un proceso de usuario como se muestra en el siguiente diagrama.

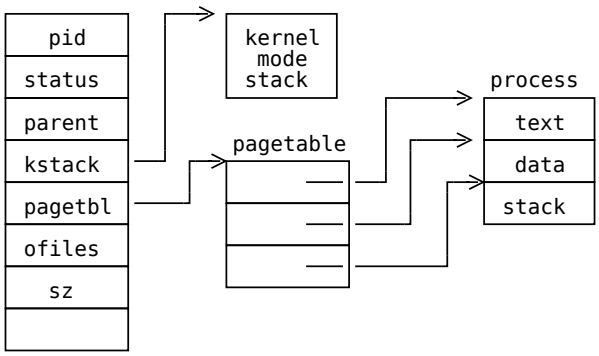


Figura 1: *Representación de un proceso.*

Cada proceso, excepto *init* tiene un padre, representado por el puntero `parent`.

La *pila en modo kernel* se usa cuando ocurre un *trap* (interrupción o excepción) mientras está el proceso está `RUNNING` o cuando realiza una llamada al sistema). Ante este evento, el sistema *salta* al *trap handler* del kernel correspondiente y éste hace que la cpu cambie de pila.

La *tabla de páginas* define el mapa de memoria del proceso, es decir las áreas (código, datos globales y la pila en modo usuario) a las que tiene acceso.

El arreglo `ofiles` contiene punteros a los descriptores de archivos abiertos por el proceso.

El campo `sz` determina el *tamaño* del espacio (dirección máxima) de memoria usada por el proceso.

Xv6 representa a los procesos en un arreglo de dimensión fija. Ver `proc.c`.

La arquitectura risc-v

Es un *instruction set architecture (ISA)* abierta basada en una arquitectura *Reduced Instruction Set Computer (RISC)*. El proyecto fue iniciado en 2010 en la Universidad de California. Actualmente existen varias implementaciones en *chips* y *boards*. El objetivo es que sea una arquitectura que permita el desarrollo de cpus compactas, con buen rendimiento y bajo consumo.

Se basa en un diseño modular, partiendo de un sistema básico con múltiples extensiones posibles. La siguiente tabla describe el ISA básico.

Nombre	Descripción	Instrucciones
RV32I	Ops enteros 32bits	40
RV64I	Ops enteros 64bits	15

Nombre	Descripción	Instrucciones
RV128I	Ops enteros 128bits	15

Algunas extensiones:

Nombre	Descripción	Instrucciones
M	Integer multip./division	8 (RV32), 16 (RV64)
A	Atomic instructions	11 (RV32), 13 (RV64)
F	Floating point	26 (RV32), 30 (RV64)
H	Hypervisor instructions	15
S	Supervisor instructions	4

Xv6 corre sobre una plataforma *Sv39*, una arquitectura de 64 bits con extensiones *M*, *A y *S* con unespacio de direcciones de 39 bits.

Una cpu de este tipo tiene tres *modos de ejecución*: *machine mode* (el modo más privilegiado), en el cual inicia, *supervisor mode*, generalmente usado como *kernel mode* y *user mode*.

Para ver más detalles sobre la arquitectura *risc-v*, ver

1. **Risc-v ISA unprivileged**

   El capítulo 25 describe las instrucciones en *assembly language* y los registros de la cpu y sus convenciones de uso.

2. **Risc-v ISA provileged**. En particular ver la descripción de los modos de operación y los registros especiales de control y estado (*CSRs*), que están disponibles en los modos *machine* y *supervisor*.

El *risc-v virtIO board* emulado por qemu tiene una arquitectura básica como se muestra en el siguiente diagrama:

Memory Maped,
LA entrada y salida esta en memoria.

Si quiero escribir algo, es en el frame buffer.

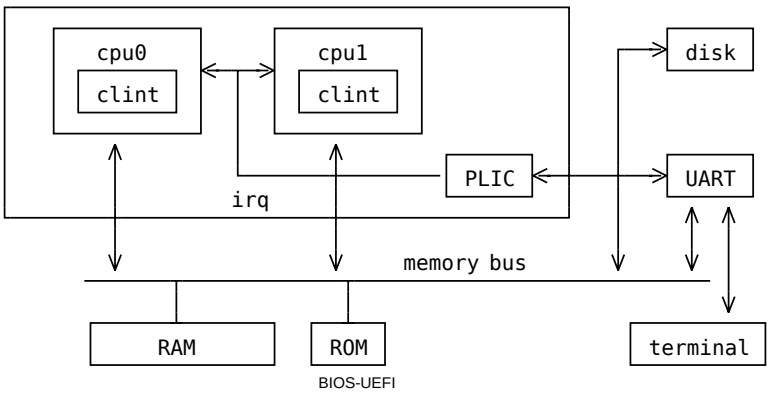


Figura 2: *Esquema de la placa risc-v a usar.*

Los dispositivos de entrada-salida (UART y el disco) generan interrupciones que son ruteadas por el *Plataform-Level Interrupt Controller (PLIC)* hacia las cpus.

Traps y modos de ejecución

La arquitectura *riscv* usada soporta 3 modos de ejecución o privilegio:

- *Machine*
- *Supervisor*
- *User (unprivileged)*

En cada modo, existen registros de control y estado (*CSRs*) e instrucciones que sólo pueden accederse o ejecutarse en el modo correspondiente o superior.

Una cpu *riscv* inicia en *machine mode* (el de mayor privilegio). Un sistema operativo debe configurar cómo *atrapar* los *traps*: interrupciones (asíncronas de dispositivos), excepciones (*bad address/instruction*, *división por cero*, ...) y llamadas al sistema (instrucción `ecall`).

Los dos últimos tipos de *traps* son *síncronas* porque las genera la CPU en la ejecución de la instrucción `ecall`.

En cada *trap* la cpu ejecuta los siguientes pasos:

1. Si es una interrupción de un dispositivo y `sstatus | SIE == 0` (las interrupciones están deshabilitadas), ésta es ignorada. Sino,
2. Desahilita interrupciones (0 en el bit `SIE` de `sstatus`).
3. Salva el *program counter* `pc` en el registro de control `sepc`
4. Salva el *modo corriente* en el bit `SPP` de `sstatus`.
5. Setea `scause` con el motivo (código) de la interrupción.
6. Pasa a modo supervisor
7. Setea `pc = stvec`. En xv6 `stvec` apunta a las rutinas `uservec` o `kernelvec`, dependiendo si estaba en modo usuario o en modo supervisor, respectivamente.
8. Esto produce que la CPU *salte* al *interrupt handler* (`uservec` por ejemplo).

En cada nivel de privilegio existen registros de control y estado (*CSRs*). Sólo se analizan en base a nuestro interés los registros en los modos privilegiados (*supervisor* y *machine*).

En *supervisor mode* existen los siguientes CSRs:

Nombre	Descripción (uso)
sstatus	Registro de estado
sedeleg	Registro de delegación de excepciones
sideleg	Registro de delegación de interrupciones
sie	Registro de habilitación de interrupciones
stvec	Dirección del *trap handler*
sscratch	Puntero a datos (parámetros para el *trap handler*)
sepc	Valor del *program counter* previo al trap
scause	Causa del trap
stval	Bad address or instruction (que causó la excepción)
sip	Interrupt pending
stp	Cpu thread (or hart) id
satp	Address translation and protection (pointer to page table)

Tabla 1: *CSRs en modo supervisor.*

En *machine mode* los registros de la tabla de arriba están prefijados por *m* en lugar de *s* y además existen los siguientes registros:

Nombre	Descripción
mhartid	Identificador de la cpu (core)
pmpcfg0-15	Physical memory protection configuration registers
pmpaddr0-63	Physical memory protection address registers

Tabla 2: *CSRs de interés en modo machine.*

Los registros de configuración de protección y direcciones de memoria, *pmpcfg* y *pmpaddr* permiten configurar *áreas de memoria* y sus modos de acceso en *machine mode* (ver la sección 3.5 del volumen II del manual de riscv *privileged architecture*).

En xv6, al inicio en *machine mode* se configura un área con toda la memoria accesible. Ver la función `start()` en `kernel/start.c` .

Por omisión, todas las interrupciones y excepciones se *atrapan* en *machine mode* pero se puede configurar la cpu para que las *delegue* a otro modos mediante los registros *medeleg* y *mideleg*. La función `start()` en `start.c` , delega los traps a *modo supervisor*, excepto las interrupciones del *timer* que sólo pueden manejarse en *machine mode*.

Para pasar a un modo menos privilegiado se debe configurar el registro *status* correspondiente (*mstatus* o *status*) para determinar a qué modo retorna una instrucción `mret` (retorno desde *machine mode*) o `sret` (retorno desde *supervisor mode*).

En xv6, `start()` , ejecutándose en *machine mode*, setea `mepc = main` y `mstatus = MSTATUS_MPP_S` para que al ejecutar `mret` salte a la función `main()` en `main.c` en modo supervisor.

En cada transición del kernel a un proceso de usuario se configuran los CSRs *sstatus*, *sepc* y *satp* para que la instrucción `sret` retorne a *modo usuario*. Ver `usertrapret()` en `trap.c` .

Xv6 abstrae las interrupciones, excepciones y syscalls en *traps (trampas)*. Los *trap handlers* son las rutinas `kernelvec` , `timervec` (definidas en `kernelvec.S`) que atrapan las interrupciones o excepciones cuando la cpu está en `supervisor mode` y `uservec` (definida en `trampoline.S`) que maneja *traps* cuando ocurren en *user mode*. El registro `stvec` contiene la dirección de `kernelvec` o `uservec` según corresponda.

Los *trap handlers* `uservec` y `kernelvec` salvan el estado de la CPU; `kernelvec()` lo hace en la pila actual mientras que `uservec()` en `trapframe` : un área reservada para cada proceso. Luego invocan a las funciones `kerneltrap()` y `usertrap()` definidas en `trap.c` .

Por lo explicado arriba, podemos considerar que, luego del inicio del sistema, las funciones `kerneltrap()` y `usertrap()` son los *puntos de entrada al kernel*.

Mapas de memoria y protección

Las arquitecturas modernas proveen mecanismos de memoria virtual y protección. Risc-v particiona la memoria *lógicamente* en *páginas*: bloques de igual tamaño de 4KB. Esto se conoce como *paginado*.

Este mecanismo permite definir *espacios de memoria* para un proceso o el kernel.

Un *espacio* es un área de memoria *lógicamente contigua* (aunque físicamente sus páginas no necesariamente estén contiguas).

Con paginado, un hilo de ejecución sólo puede acceder a sus espacios de memoria. Además permite definir permisos de acceso (lectura, escritura, ejecución y otros) para cada página.

Estos espacios de memoria se representan en *tablas de páginas* que conceptualmente definen una función $mem(virtual\_address) \rightarrow physical\_address$.

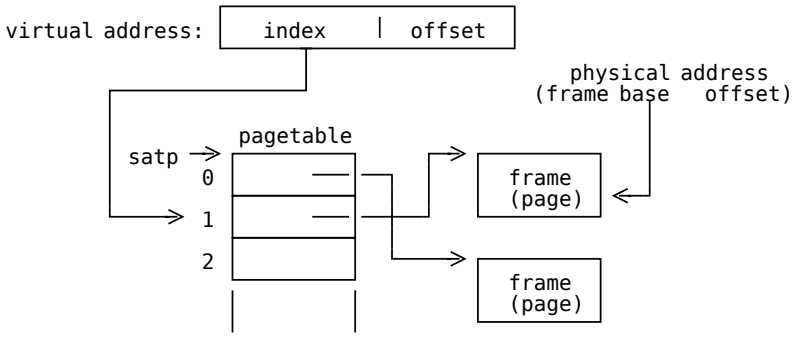


Figura 3: *Esquema de una page table.*

Las tabla de páginas se configuran en memoria y son *arreglos* de *punteros* a direcciones base de *frames* (o *pages*).

Una dirección de memoria (*lógica* o *virtual*) se interpreta diviéndola en dos partes: $(index, offset)$. La disrección *física* (o real) se computa como $physical\_address = satp[index] + offset$, como se muestra en la **figura 4**.

Xv6, como otros SO, define una *page table* para el *kernel* y una para cada proceso.

La ROM, RAM y los puertos de los controladores de los dispositivos se *mapean en espacios de direcciones físicas de memoria* como se muestra en la siguiente figura.

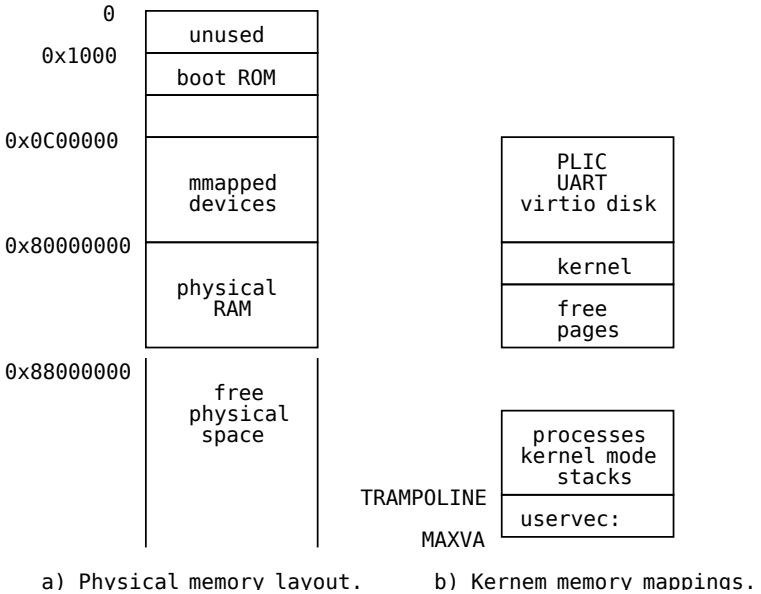


Figura 4: *Distribución (layout) de la memoria en qemu-RV-39.*

La tabla de páginas para el kernel mapea las direcciones físicas de la RAM y de los dispositivos de forma 1 a 1. El código y datos del módulo `trampoline.S` se mapean en la última página direccionable por la arquitectura RV-39 (`MAXVA`). Antes de `TRAMPOLINE` (direcciones menores) se mapean las páginas reservadas para el stack en modo kernel de cada proceso.

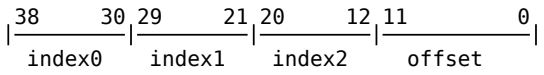
El script `kernel.ld` indica al linker que el código y datos de `trampoline.S` se enlace a partir de la dirección `TRAMPOLINE` .

La *page table* de un proceso describe el espacio de memoria para sus áreas de código, datos y stack a partir de la dirección lógica 0 (cero).

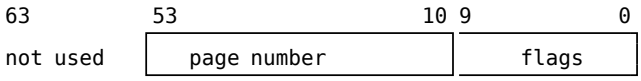
Implementación de tablas de páginas

Una tabla de páginas puede tener muchas entradas ya que un proceso podría necesitar un gran espacio de direcciones. Esto requiere que comúnmente se implementen como un árbol, donde cada nodo tiene un número de entradas (*ptes*) tal que el tamaño del nodo sea igual al tamaño de página.

El modelo riscv *Sv39* (64 bits con un bus de direcciones de 39 bits), con páginas de 4KB, usa un árbol de 3 niveles. Una dirección lógica se interpreta como se muestra en la siguiente figura.



Cada nodo del árbol es de 4 KB y contiene 512 *pte*s de 64 bits de la forma



donde el *page number (pn)* corresponde a la dirección física base de la página y los flags son bits de permisos de acceso:

- *U*: Permiso de acceso (si está en 1) en modo usuario.
- *W*: Permiso de escritura en la página.
- *V*: Si es 1 es una *pte válida*.
- otros...

Este mecanismo representa una función de traducción de direcciones virtuales a físicas. El subsistema de la cpu que realiza esta traducción y verificación de acceso se conoce como la *memory management unit (MMU)*.

Los 9 bits (38-30) de la dirección lógica o virtual más significativos corresponden al *índice* en la raíz, los bits (29-21) el índice en el nodo hijo del nivel intermedio y los bits (20-12) el índice en el nodo hoja. Este índice refiere a la *pte* en el nodo hoja que contiene la *dirección física base* de la página. La dirección física final es la indicada por el *offset* dentro de la página (*pagenumber+offset*).

En este esquema, la *MMU* computa *pa=((root[va.index0].pn)[va.index1].pn)[va.index2]+offset*.

donde *pa* es la *physical address* y *va* es la *virtual (logical) address*.

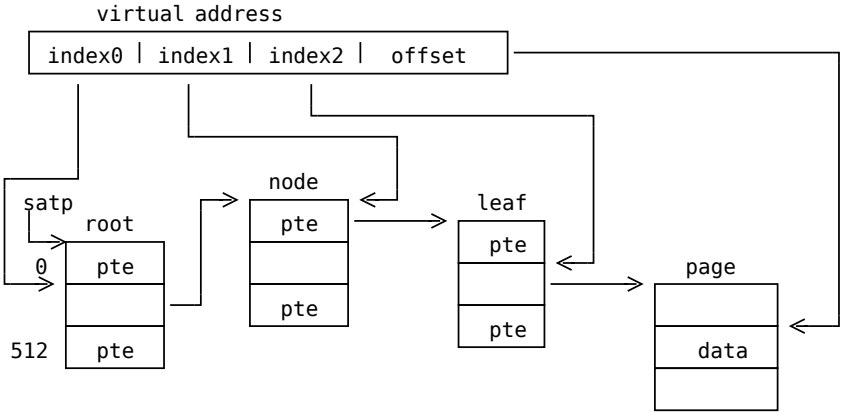


Figura 5: *Traducción de direcciones lógicas a físicas en RV-39.*

En *risc-v*, en modo supervisor, el registro ***Supervisor Address Translation and Protection (satp)*** apunta a la raíz del árbol de la tabla de páginas en uso por la CPU.

Xv6, en cada *context switch* el kernel deberá hacer que éste registro apunte a la tabla de páginas del proceso al que se le dé el control.

Esto se hace en la función en assembly `userret()` en `trampoline.S` la cual es invocada como `userret(p->pagetable)` desde `usertrapret()` (en `trap.c`). La función `userret(satp)` retorna desde el modo supervisor a modo usuario para continuar con la ejecución del proceso corriente o el nuevo seleccionado por `scheduler()` .

Traps en modo usuario

Al ocurrir un *trap* en *user mode*, la cpu salta a `uservec()` en `trampoline.S` . Esta función forma parte del código del kernel, pero la cpu está usando la *page table* del proceso de usuario.

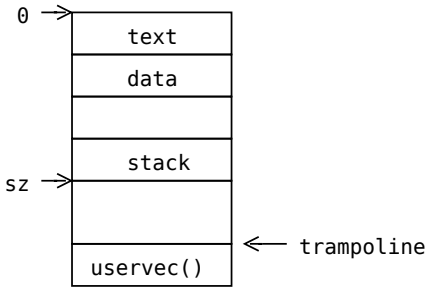


Figura 6: *Espacios de memoria de un proceso.*

Xv6 mapea la página física de *trampoline* (que contiene `uservec()` y `userret()`) en el espacio de memoria del proceso (cuando éste fue creado). Ver `alloc_proc()` y `proc_pagetable()` en `proc.c` .

Esto es necesario ya que cuando ocurra una interrupción, excepción o syscall en modo usuario la CPU no cambia automáticamente a la tabla de páginas del kernel (como ocurre en CPUs x86).

La CPU pasa a ejecutar código del kernel (`uservec()`)en modo supervisor pero en el espacio de memoria del proceso, así que éste código debe poder ser accesible, por lo que podemos decir que *trampoline.S* contiene código compartido entre el kernel y todos los procesos.

Así, al ocurrir un *trap*, se la CPU salta a `uservec()` y ésta hace que la CPU use la tabla de páginas del kernel.

Antes de retornar del *trap* a modo usuario, se restaura el valor de `satp` para que apunte nuevamente a la tabla de páginas del proceso correspondiente.

Ejecución de xv6 sobre qemu

La regla `make qemu` del `Makefile` lanza qemu `kernel` como la imagen del kernel a cargar, `fs.img` como la *imagen* del disco, con `CPUS` cores y con 128MB de RAM.

Qemu carga el `kernel` en la dirección `0x80000000` (dirección de comienzo de la RAM física) y pasa el control a esa dirección. El código del kernel está compilado y enlazado (ver `kernel/kernel.ld`)para ejecutar a partir de esa dirección. La primera función enlazada en el kernel (en `0x80000000`) es `_entry()` , definida en assembly en `entry.S` .

En *risc-v*, cada cpu inicia en paralelo en *machine mode*, o sea que cada cpu comienza a ejecutar `entry()` .

A partir de allí, el código del kernel de xv6 realiza los siguientes pasos:

1. `entry()` : Prepara cada cpu (*hart* en terminología de *risc-v*) para usar un *stack inicial* e invoca a `start()` , definida en `start.c` .

2. `start()` : Configura cada cpu para saltar a `main()` (definida en `main.c`) en *modo supervisor*. Deshabilita *paginado* (por el momento) y configura las cpus para que las *interrupciones* (del disco y del UART) y las *excepciones* se atrapen en modo *supervisor*. Aquí también se configura el *timer* (`CLINT`) para cada cpu y el *timer interrupt handler* (`timervec` en `kernelvec.S`). Las interrupciones del *timer* se deben atrapar *machine mode*.

3. Luego `start()` retorna del *machine mode* saltando a `main()` en modo *supervisor*.

4. `main()` hace que la cpu 0 inicialice los subsistemas y cree el primer proceso (ver `userinit()` en `proc.c`), el cual básicamente crea un proceso con el código `initcode` (binario extraído desde `initcode.S` y *hardcoded* en el arreglo de bytes `initcode[]`). Esa secuencia de instrucciones de máquina es equivalente a `exec("init", ["init", 0])` . Finalmente, `main()` hace que cada cpu invoque a `scheduler()` .

5. En `scheduler()` las cpus *compiten* por seleccionar un proceso en estado `RUNNABLE` y lo pasa a estado `RUNNING` (le asigna la cpu corriente), efectuando el primer *context switch* a modo usuario. Una de las CPUs encontrará al primer proceso con el código de `initcode` , el cual tiene un *contexto* salvado de interrupción (creado durante su creación). La función `scheduler()` *salta/cambia al contexto del proceso* ejecutando `swtch(&c->context, &p->context))` .

6. La CPU (`c`) continúa ejecutando `forkret()` (indicado por el valor del `pc` en `p->context`) y finalmente alcanza `usertrapret()` la cual ejecuta un *retorno de una interrupción* de modo supervisor a modo usuario, previamente restaurando los valores de los registros de la CPU salvados en el *trapframe* en el stack en modo kernel del proceso. En el primer proceso el `pc` queda apuntando a la primera instrucción del programa `initcode` .

7. El código de `initcode` ejecuta el syscall `exec("init",...)`. Esta llamada al sistema *reemplaza* la imagen de memoria del código y datos del proceso corriente, carga del disco (*filesystem*) las instrucciones de código y datos del programa (archivo ejecutable en formato *ELF*) `init`, inicializa el stack en modo usuario (argumentos de la función `main`) y configura el *trapframe* para que al retorno de la interrupción `init` comience por su *entry point* (main).

8. El proceso `init` (con *pid=1*) abre los archivos de entrada y salida estándar y lanza el *shell* hacendo `fork()` y `exec("sh",...)`. Este último permite la interacción del usuario con el sistema.

A partir de aquí, el usuario al introducir un comando, el shell realizará las llamadas al sistema correspondientes para lanzar nuevos procesos.

Demo 1: Compilación y ejecución (sobre qemu) de xv6.

Para más detalles de los aspectos de diseño e implementación de vx6, ver **xv6 book**.

---

[1]↩️ Cross-compiler: Compilador que corre en una plataforma pero genera código para otra.

Usuarios y seguridad