

Procesos

Un *proceso* representa un *programa de usuario en ejecución*. Generalmente se representa con al menos los siguientes atributos:

- Un identificador (*process id* o *pid*).
- Su estado: Ver la [figura 3](#).
- Razón o recurso por el cual está en estado `SLEEPING`.
- Su *mapa* (áreas) de memoria a las cuales puede acceder en modo usuario.
 - Código (*text segment*).
 - Datos globales (*estático*): Área de datos inicializados (*data section*) y no inicializados (*bss section*).
- Al menos un *thread* de ejecución, representado por
 - Stack en modo usuario. Stack que usa el *thread* mientras ejecuta en modo usuario.
 - Stack en modo kernel. En este stack se almacena el *trapframe*: Valores de los registros de la CPU al ocurrir una interrupción en éste thread. Esta pila también se usa para el control de invocación a funciones dentro del kernel ejecutándose en el *contexto del proceso*. Finalmente, también se usa para guardar el *contexto* (*caller saved CPU registers*) del thread cuando se produzca un *context switch*.
- Conjuntos de recursos adquiridos: Archivos/pipes/sockets abiertos, semáforos, timers, áreas de memoria compartida, manejadores de señales y otros.
- El *comando* o *programa* del cual es una instancia.

Linux

GNU-Linux representa a un proceso por medio de una `struct task_struct`, el cual contiene los atributos básicos del thread y punteros a estructuras de datos que representan los demás recursos (archivos, memoria, etc). En un proceso *multithread* cada `struct task_struct` apuntan a los mismos recursos, ya que pertenecen al mismo proceso y tienen el mismo *thread group=PID* aunque diferente *thread id*.

Cuando un proceso se carga en memoria, mediante el syscall `exec(path, args)`, sus secciones de código (*text*) y datos (*data*) se cargan desde el archivo ejecutable (en formato ELF, por ejemplo), se le asigna espacio para el stack del thread principal, se inicializa con un *stackframe* que al ser planificado se ejecutará desde su *punto de entrada* (función `_start` de la biblioteca estándar). Esta función comúnmente se define como:

c

```
void _start(void)
{
    exit(main());
}
```

La siguiente figura muestra el esquema (*layout*) en memoria de un proceso.

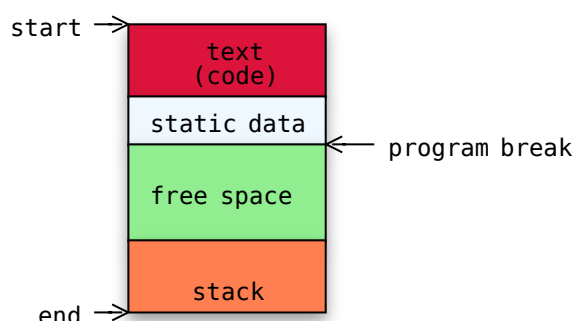


Figura 1: *Layout (virtual) de memoria de un proceso.*

El **espacio de direcciones** de un proceso es $(start, end)$. En muchos sistemas, $start = 0$.

En los SO tipo UNIX, la llamada al sistema `sbrk(n)` permite extender/reducir el segmento de datos del proceso asignando o liberando memoria y actualiza el *program break*.

El subsistema de gestión de procesos se encarga de la creación, cambios de estado, cambios de contexto, sincronización y finalización de procesos. Mas abajo se describe en detalle el uso de los atributos mencionados. También se encarga de mantener el registro de los recursos adquiridos por cada proceso.

Llamadas al sistema de gestión de procesos

Algunas de las llamadas al sistema para la gestión de procesos se describen a continuación.

- `fork()` : Crea un nuevo proceso (hijo) el cual es una copia del proceso invocante. El proceso hijo *hereda* los descriptores de los archivos abiertos del padre.

- `exit(exitstatus)` : Finaliza el proceso.
- `wait(&childstatus)` : Espera (se bloquea) hasta que finalice un proceso hijo.
- `getpid()` : Retorna el identificador del proceso corriente.
- `kill(pid)` : Envía una señal (terminación) al proceso con id `pid` .
- `sleep(n)` : Espera (se bloquea) por `n` segundos.
- `exec(filename, args)` : Reemplaza la imagen de código y datos del proceso corriente por el código y datos de `filename` . Al final de esta sección se describe en mayor detalle.
- `sbrk(n)` : Asigna `n bytes` de memoria extra al proceso invocante.

Estados de un proceso

Un proceso puede estar en uno de los siguientes estados:

- *RUNNING*: Está ejecutando actualmente, tiene asignada una CPU.
- *RUNNABLE* o *READY*: Está esperando a que el kernel le asigne una CPU.
- *SLEEPING* o *WAITING*: Está *bloqueado* hasta la ocurrencia de un evento que lo despertará.

La [figura 3](#) muestra los cambios de estado que puede tener un proceso y las funciones de un kernel involucradas en las transiciones.

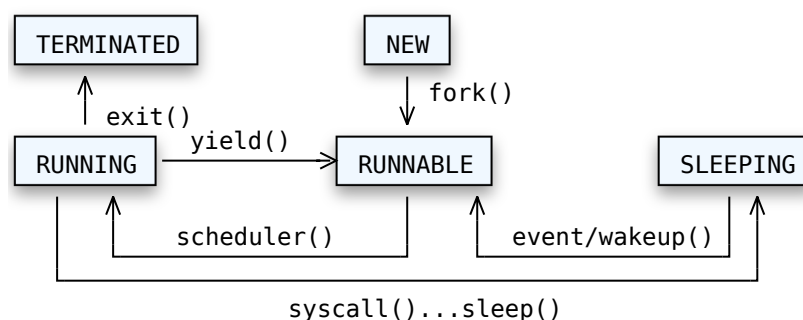


Figura 2: Cambios de estado de procesos.

Pueden existir otros estados. Por ejemplo en los sistemas tipo UNIX, un proceso que ha terminado su ejecución (vía `exit()`) pero aún existe en el sistema se denomina *zombie*. Esto ocurre comúnmente cuando el proceso *padre* aún no ha ejecutado `wait(&status)` y se requiere mantener su estado de terminación en el descriptor del proceso hasta que el proceso padre ejecute el `wait()` correspondiente.

En los sistemas operativos tipo UNIX, cada proceso, excepto `init` (el primer proceso lanzado por el kernel luego del *booting*) tiene un padre. El kernel debe mantener esta invariante.

Nota

¿Qué sucede si un proceso padre termina antes que sus hijos? En UNIX, los procesos hijos son *adoptados* por `init`.

Representación de procesos

Una de las estructuras de datos principales es el conjunto de *process control blocks* (PCB) o *descriptores de procesos*. Este conjunto puede estar implementado como un arreglo (tabla), diccionario o listas.

Un *descriptor* de proceso es una estructura o registro que contiene campos para representar los atributos ya mencionados.

Linux

Los descriptores de procesos están representados en la estructura de datos

Las operaciones que realizan los cambios de estado de un proceso se muestran en la [figura 3](#). No confundir estas funciones con aquellas que implementan las llamadas al sistema con el mismo nombre, como `sleep` o `fork`. Las funciones que implementan los *syscalls* comúnmente invocan a estas funciones.

En *xv6*, los *syscalls* están implementados en las funciones `sys_fork()`, `sys_exec()`, ...) en `sysproc.c` y `sysfile.c`.

En los sistemas tipo UNIX, cada proceso (excepto *init*) tiene un padre: El proceso que invocó el *syscall* `fork` correspondiente. Esto induce a una estructura de árbol como representación de todos los procesos del sistema.

Sobre la tabla de procesos comúnmente también se representan colas o listas para representar sub-conjuntos de procesos. Por ejemplo, un *scheduler* que implementa una política *FIFO*, implementará el conjunto de procesos `RUNNABLE` como una cola, lista o red-black tree.

Un ejemplo interesante es la implementación de [listas doblemente encadenadas intrusivas](#) en GNU-Linux.

Xv6

`scheduler()` recorre toda la tabla de procesos hasta encontrar un proceso `RUNNABLE`. Obviamente, esto tiene un impacto negativo en rendimiento.

Cuando un proceso está en estado `SLEEPING`, el sistema deberá conocer su motivo. Un diseño e implementación común es asociar el conjunto de procesos que esperan al evento que los despertará.

Xv6

Esto se representa en el campo `chan` (*channel*) del descriptor del proceso, el cual apunta a un dato (heterogéneo) que representa el motivo. Por ejemplo, un proceso `p` esperando por el syscall `sleep(seconds)`, `p->chan` apunta a la variable global `ticks`.

Otro aspecto a tener en cuenta es qué sucede si el *scheduler* no encuentra un proceso `RUNNABLE` (situación muy común). Básicamente se pueden implementar alguna de las siguientes alternativas:

1. Hacer que `scheduler()` quede en un ciclo hasta encontrar uno.
2. Usar un proceso `idle`, creado inicialmente por el sistema y que ejecute `while (true) yield();`. Este proceso al obtener la CPU la libera inmediatamente.

En cualquier caso ésto es una oportunidad para *contabilizar* la carga de trabajo (o *idleness*) del sistema para realizar acciones como por ejemplo, ahorro de energía^[1] o disparar operaciones diferidas (como procesar paquetes de red recibidos).

Recordemos que el kernel tomará el control cada vez que ocurra un *trap*.

Traps

Un *trap* es una abstracción de los siguientes eventos:

- El proceso ejecutó un *syscall* (instrucción `ecall` en *risc-v*), o
- la CPU disparó una excepción porque el proceso intenta ejecutar una instrucción inválida (privilegiada, división por cero, etc) o porque intenta acceder a un área de memoria inválida, o
- un dispositivo generó una *interrupción*. Un kernel con *preemptive multi-tasking* usa un *timer* o *clock* que genera interrupciones periódicamente.

Otra estructura de datos importante es la representación de la cpu, la cual comúnmente contiene comúnmente los siguientes atributos:

- Identificador de la CPU (por si hay más de una)

- Una referencia al proceso corriente.
- Un *contexto salvado*: Estado de la CPU del último *cambio de contexto*.
- Estado de las interrupciones (habilitadas/deshabilitadas) y otros.

En una arquitectura *smp* (*multicore*) se deberá mantener un conjunto de estas estructuras.

Para ver en detalle las estructuras de datos para la gestión de procesos en xv6, ver `proc.h` .

Llamadas al sistema

Comúnmente, a cada llamada al sistema se le asigna un identificador único. Es común asignarles números naturales (en xv6, ver `kernel/syscall.h`).

Las llamadas al sistema se implementan en dos partes:

- Invocación desde un proceso (*user mode*).

Generalmente se implementan como funciones de biblioteca. Cada función se implementa comúnmente poniendo el código de *syscall* en un registro designado de la cpu (`a7` en *risc-v*) para que el kernel lo pueda obtener y luego ejecutar una instrucción de *trap* (`int` en x86 o `ecall` en *risc-v*).

En xv6 están implementadas en `user/usys.S` .

- Funciones de servicios en el kernel.

Al ejecutarse una instrucción de *trap* correspondiente a un *syscall*, se dispara el mecanismo de manejo de interrupciones y se ejecuta el *trap handler* correspondiente en el kernel. En este caso, se invocará a un *dispatcher* que analiza el código del *syscall* (que está en un registro de la cpu) e invoca a la función de servicio correspondiente.

En xv6, éstas funciones están definidas en `sysproc.c` y `sysfile.c` . Por cada *syscall* hay una función `sys_<syscall>()` que implementa el servicio. El *dispatcher* es la función `syscall()` (en `syscall.c`), que invoca a la rutina apuntada por `syscalls[n]` , donde `n` es el código del *syscall*.

Generalmente las *syscalls* retornan algún valor o código de error a los procesos. En el caso de una API *POSIX*, los valores negativos son códigos de error.

Los syscalls `exit` y `wait`

Estas dos llamadas al sistema POSIX están relacionadas.

- `exit(exitcode)` : Finaliza el proceso corriente con código de salida `exitcode` .
- `wait(&exitcode)` : El proceso corriente espera (se bloquea) hasta que finalice un hijo. El código de salida del hijo se *copia* en el parámetro de salida `exitcode` .

Esto ofrece un mecanismo de sincronización y comunicación básico entre procesos. El proceso padre puede saber cómo terminó el hijo.

Por convención se asume que el código o *status* de salida 0 es *terminación normal* (o exitosa). Un código de salida diferente a cero representa una finalización anormal y el código indica el motivo. Cada aplicación define el significado de los códigos de salida anormales.

Esto motiva el estado `ZOMBIE` de un proceso que ejecutó `exit()` antes que el padre ejecute `wait()` , ya que al menos se debe mantener el *código de salida* hasta que el padre ejecute `wait()` .

En el caso que el proceso padre finalice sin hacer un `wait()` , los sistemas tipo UNIX comúnmente hacen un *reparent* del proceso hijo, comúnmente haciendo que lo *adopte* `init` .

En un shell el código de salida del último proceso queda en la *variable de ambiente* `$?` .

Hilos de ejecución (threads)

Un proceso tiene al menos un *hilo* o *thread* de ejecución que representa la secuencia de instrucciones en ejecución por la cpu.

Un *thread* generalmente está representado por:

- Registros de la cpu:
 - *Program counter* (`pc`): Dirección de la próxima instrucción a ejecutar.
 - *Stack pointer* (`sp`): Dirección del *tope* de la pila.
 - *Frame (o base) pointer* (`fp`): Dirección base del registro de activación del tope.
- Un *stack*: Pila de *registros de activaciones* de funciones.

La pila contiene *registros de activación* para el control de la secuencia de invocaciones a funciones y sus retornos. Además comúnmente contienen los *datos automáticos* (variables locales y argumentos) de las funciones activas. De ese modo se crean y eliminan automáticamente los *ambientes locales*.

La representación de un *registro de activación* se basa en el *calling convention* de la *ABI* de la plataforma.

El *frame pointer* se usa en las instrucciones del programa como dirección base para acceder en forma relativa a las variables locales y argumentos dentro del registro de activación.

Los pasos en una invocación a una función *f* son los siguientes:

1. Invocante: Reserva espacio para el valor de retorno, si es que se dejará en la pila. Si se retorna en un registro, este paso se omite.
2. Invocante: Apila (o pone en registros) los valores de los argumentos.
3. Invocante: Ejecuta la instrucción `call f`, que comúnmente apila el valor del *program counter* y luego le asigna la dirección de la primera instrucción de *f*.
4. Invocada (*f*): Apila (salva) el valor el *frame pointer* y le asigna el valor del *stack pointer*.
5. Invocada (*f*): Reserva espacio para las *variables locales*, restando un valor al *stack pointer*. Estos pasos (4 y 5) se conocen como el *prólogo*.
6. Invocada (*f*): Ejecuta las instrucciones del *cuerpo* de *f*.
7. Invocada (*f*): Comienza el retorno (*epílogo*). Se eliminan las *variables locales* y *temporarios* (`sp=fp`).
8. Invocada (*f*): Recupera el *base pointer* salvado (`pop fp`).
9. Invocada (*f*): Retorna (ejecuta la instrucción `ret`).
10. Invocante: Desapila los argumentos, si hubiera, y toma el valor retornado.

Esta secuencia de operaciones deja un *activation record* con el siguiente formato:

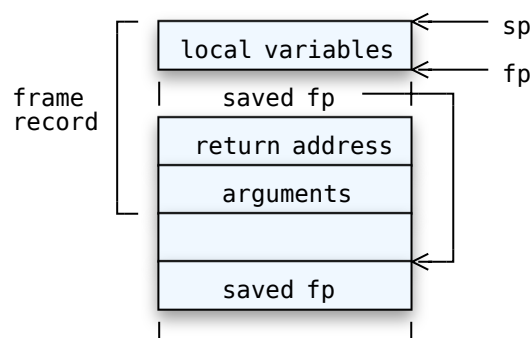


Figura 3: Pila de registros de activación de funciones.

El *call convention* en una ABI de un SO o plataforma de hardware especifica los detalles de estos pasos. En particular, cómo se pasan los argumentos (en registros, pila o ambos) y el valor retornado (en un registro o en la pila). Por ejemplo, Linux para x86-64 usa la especificación [System V ABI](#).

riscv

Una cpu *risc-v* tiene 32 registros de propósitos generales más el *program counter* (`pc`). Algunos de sus registros se usan según la siguiente convención:

- `x1` o `ra` : Return address (set by `call` or `jmp` instructions)
- `x2` o `sp` : Stack pointer.
- `x8` o `fp` : Frame pointer.
- `x10` o `a0` : Argumento 0 / valor de retorno.
- `x11` (`a1`)- `x17` (`a7`): Argumentos 1..7. El resto de los argumentos se deben apilar en el stack.

La instrucción `call` no salva el `pc` en el stack. Se debe hacer en el prólogo > de una función. `ret` no desapila del stack (hacerlo en el apílogo).

Un hilo puede estar ejecutando en dos modos:

1. *User mode*: La CPU está ejecutando instrucciones del proceso.
2. *Kernel mode*: Se ha alterado el flujo de ejecución normal del proceso debido a la ocurrencia de un *trap*:

El kernel, en su inicialización, definió un *vector de interrupciones*, el cual define un *trap handler* para cada tipo de *trap*. Para más detalles sobre manejo de interrupciones y modos de ejecución ver el capítulo de [taller de xv6](#).

Al ocurrir un *trap*, el hardware y el *trap handler* correspondiente realizan:

1. La CPU pasa a *kernel mode*.
2. Se *salta* al *trap handler* correspondiente.
3. Si el trap se produjo en *user mode*, cambia al *mapa de memoria* del kernel y cambia al *stack en modo kernel* del proceso corriente.
4. Salva el estado de la CPU (valores de los registros) en el *trapframe* correspondiente.
5. Procesa el *trap*: Llama a diferentes funciones del kernel.
6. En el retorno, recupera el estado de la CPU salvado en el *trapframe*.

7. Retorna del *trap* ejecutando una instrucción de *retorno de interrupción* o *retorno del modo supervisor* (ej: `iret` en x86, `sret` en risc-v).

Xv6

El campo `kstack` de `struct proc` (definida en `proc.h`) apunta a la base del stack en modo kernel (de 4KB de tamaño).

En el caso el *trap* ocurra en *user mode*, el uso de una pila diferente impide que un proceso de usuario pueda *filtrar* datos del kernel. Un proceso ejecutando en modo usuario no tiene acceso al *kernel mode stack*.

Cambios de contexto

Es posible ver a los threads de cada proceso y el scheduler como *corrutinas*.

Definición

Una *corrutina* es un componente en un programa que pueden ser *suspendido* por medio de las operaciones `yield` o `sleep` y luego continuar (`resume`) desde su punto anterior preservando su estado local. Comúnmente se usan para implementar *multitarea cooperativa*.

En xv6, la función `sched()` (mediante `swtch`) implementa una operación del tipo `yield()` o `resume(scheduler)` mientras que el `swtch` en `scheduler()` implementa un `resume(p)` donde `p` es el proceso (corrutina) seleccionado como se muestra en el siguiente diagrama.

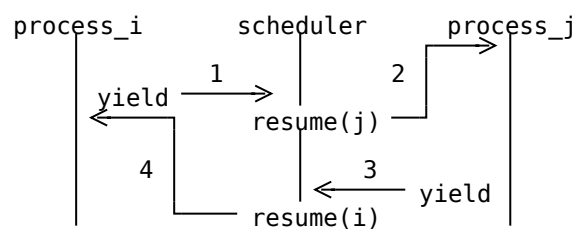


Figura 4: *Visión de threads como corrutinas.*

La figura anterior muestra que el *proceso (corrutina)* *i* ejecutándose *abandona la cpu* mediante `yield` lo que hace que el control *continúe* en *scheduler*, el cual decide hacer *continuar* al *process j*. Este último luego abandona la cpu y *scheduler* hace que el *proceso i* continúe o *reasuma* desde el punto a continuación de `yield`.

Como ya se describió arriba, el kernel toma el control al ocurrir un *trap*.

Xv6

Los *trap handlers* (o *puntos de entrada*) de xv6 son:

- `uservec()` en `kernel/trampoline.S` para *traps* que ocurren en *user mode*.
- `kernelvec()` en `kernel/kernelvec.S` : para *traps* que ocurren en *kernel mode*.
- `timervec()` en `kernel/kernelvec.S` : para interrupciones del *timer*. En *risc-v* ejecuta en *machine mode*.

Estos *handlers* de bajo nivel terminan invocando a `usertrap()` o `kerneltrap()` , definidas en `trap.c` . Estas funciones determinan la acción a realizar.

Un *handler* se ejecuta en *modo kernel* y dependiendo en qué contexto ocurrió el evento puede pertenecer al *hilo de ejecución* del proceso interrumpido (*process context*) o en *kernel context* si estaba ejecutando código del kernel (ejemplo, ciclando en el *scheduler*).

Luego, en el kernel generalmente ocurren dos tipos de cambios de contexto (*threads switch*):

1. Del contexto del proceso al contexto del kernel (al *scheduler*).
2. Del contexto del kernel al contexto de un proceso (desde el *scheduler*).

El primer caso se da cuando un proceso abandona el estado `RUNNING` . El segundo se da cuando el *scheduler* selecciona un proceso `RUNNABLE` y lo pasa a `RUNNING` , asignándole la cpu.

El kernel decide que el proceso corriente debe abandonar la CPU en base a:

1. El proceso completó su ***quantum*** o ***time slice*** de uso de CPU. Esto se produce cuando el kernel recibe un *timer interrupt*. En este caso el kernel invoca a una función del tipo `yield()` . Esto es típico de un SO con *preemptive multitasking*.
2. El proceso realizó una llamada al sistema, como `read()` , `write()` , `sleep()` u otra, cuyo procesamiento puede llevar demasiado tiempo por lo que conviene suspender el proceso (pasar al estado `SLEEPING`). El kernel invoca a una función del tipo `sleep(wait_queue)` o similar e invoca al *scheduler* para que éste asigne la CPU a otro proceso.

Un proceso *sleeping* se *despertará* (pasará a `RUNNABLE`) cuando ocurra un evento como por ejemplo, la finalización de la operación de entrada-salida por la que está esperando.

Cuando un proceso deja el estado `RUNNING` , el kernel realiza los siguientes pasos:

1. Salva el estado del thread corriente (*contexto*). Al menos debe guardar el *program counter* y el *stack pointer*.

2. Recupera el contexto del *scheduler* (salvado previamente).

En *xv6* a ésto lo realiza `swtch(&p->context, &cpu->context)` en `sched()` .

Esto hace que la *cpu salte* al punto de continuación de `scheduler()` .

En *xv6*, este punto es debajo de la invocación a `swtch(&cpu->context, &p->context)` . Ver el [listado 1](#).

En este punto el kernel está ejecutando en el contexto del *scheduler*, no de un proceso.

3. El *scheduler* selecciona un proceso `RUNNABLE` `p` .

4. Se salva el contexto actual (del *scheduler*) y se cambia al *contexto* de `p` (en *xv6*, `swtch(&cpu->context, &p->context)`).

Esto hace que la CPU *continúe* su ejecución en el punto en el que abandonó la CPU en el pasado (en *xv6*, salta al punto 2 del [listado 1](#)).

La implementación de una función como `swtch(old, new)` es dependiente de la arquitectura y comúnmente se implementa en *assembly*.

Xv6

La función `swtch(old, new)` , está definida en `swtch.S` . El control (*resume*) transiciona entre (las *corrutinas*) `sched()` y `scheduler()` como se muestra en la siguiente figura.

c

```
scheduler()
{
    ...
    for(;;) {
        ...
        // to process context: resume(sched)
        swtch(&cpu->context, &p->context);
        ... (1)  <-----+
    }
}

sched() {
    ...
    // to scheduler context: resume(scheduler)
    swtch(&p->context, &mycpu()->context);
```

```
... (2) <-----+
}
```

Listado 1: Cambios de contexto en xv6.

El primer proceso

A continuación se describen los pasos que realiza xv6 en la creación del primer proceso. En otros sistemas operativos, la secuencia es similar.

Durante el inicio de xv6, `main()` configuró el hardware con los *trap handlers* como se describe en detalle en el capítulo [taller de xv6](#) y crea el primer proceso invocando a `userinit()`. Esta función realiza los siguientes pasos:

1. Crea e inicializa un descriptor del proceso con estado `RUNNABLE`.
2. Reserva memoria para sus áreas de código, datos globales y stacks de usuario y para modo kernel.
3. Carga el código y datos de un ejecutable inicial (`initcode`) en la memoria asignada. Los datos en el arreglo `initcode` corresponden al binario resultante de ensamblar `user/initcode.S` y extraer su contenido con el comando `od -t xC initcode` el cual es equivalente a

c

```
char *args[] = {"init", 0};
exec("init", args);
```

4. Configura el *contexto salvado* y el *trapframe* del proceso para simular un *retorno de un trap a modo usuario*.

Luego, cuando el kernel desde `main()` invoque a `scheduler()`, éste encontrará este único proceso como `RUNNABLE` y hará el *context switch* por la cual la CPU *saltará* a un camino de *retorno de un trap* (simulada) hacia la función `trapret()`.

Luego del retorno a modo usuario, la CPU comenzará a ejecutar el código de `initcode`. Este código ejecuta la llamada al sistema `exec("init", args)` la cual *carga* el código y datos del programa `init` desde el disco, le crea un nuevo stack en modo usuario, libera la memoria usada por `initcode` y deja al proceso en estado `RUNNABLE`. Este proceso será planificado nuevamente y al pasar a modo usuario (por segunda vez) comenzará la ejecución de `init`.

El proceso `init` generalmente se vincula a las *terminales* y dispara el proceso `login` en cada una. Finalmente `login` creará un *shell* para que un usuario comience su interacción con el sistema. La siguiente figura muestra el árbol de procesos inicial.

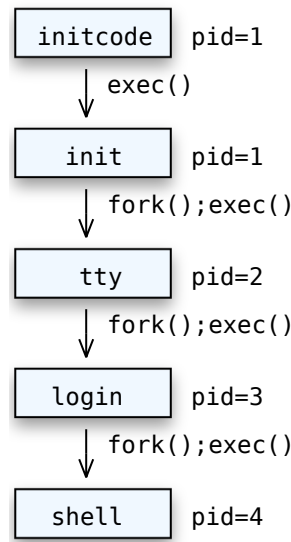


Figura 5: Inicio del sistema.

En xv6, existe una sola terminal y no soporta usuarios por lo que `init` dispara directamente el *shell*.

La llamada al sistema `exec`

El syscall `exec(cmd, args)` realiza básicamente los siguientes pasos:


1. Reserva memoria para el código, datos y *user-mode* stack y carga el código y los datos del programa (ejecutable) `cmd` en la memoria asignada. Este será el *nuevo mapa de memoria del proceso*.
2. Configura la pila en modo usuario (apilando los argumentos de `main`) y el *trapframe* para *simular* un retorno al *punto de entrada del programa* (dirección de memoria *entry* en el archivo ELF).
3. Libera la imagen de memoria anterior del proceso.
4. Configura en la cpu el nuevo *mapa de memoria* creado.

En xv6 esta función está implementada en `exec.c`.

El proceso, luego al ser planificado, retornará de modo *kernel* a modo *usuario* ejecutando la función correspondiente al punto de entrada (comúnmente `main()` o `start()`) del programa.

xv6

`exec()` asigna una *página* encima de la *página* del stack sin permisos de acceso en modo usuario para detectar *stack overflow*.

[1]  El ahorro de energía puede apagar o suspender subsistemas y bajar la frecuencia de operación de la cpu.

[< Anterior](#)

[Próximo >](#)

Herramientas de desarrollo **Planificación de uso de cpu**