Compilando el kernel de linux

Mg. Ing. Gonzalo E. Sanchez
MSE - 2022

Implementación de Sistemas Operativos II

Compilando el kernel

- Configuración y compilación
- Booting

Configuración y compilación

Configuración y compilación

- El kernel contiene miles de device drivers, drivers de filesystems, protocolos de networking, etc.
- Todo esto es altamente configurable, por lo que se da lugar a miles de opciones y combinaciones.
- Usualmente se utiliza esta flexibilidad para compilar partes del kernel (las que serán utilizadas).
- El set de opciones utilizado dependerá de:
  - La arquitectura del dispositivo final (target) y hardware asociado.
  - Las capacidades que se deseen implementadas en el kernel.

Configuración y compilación

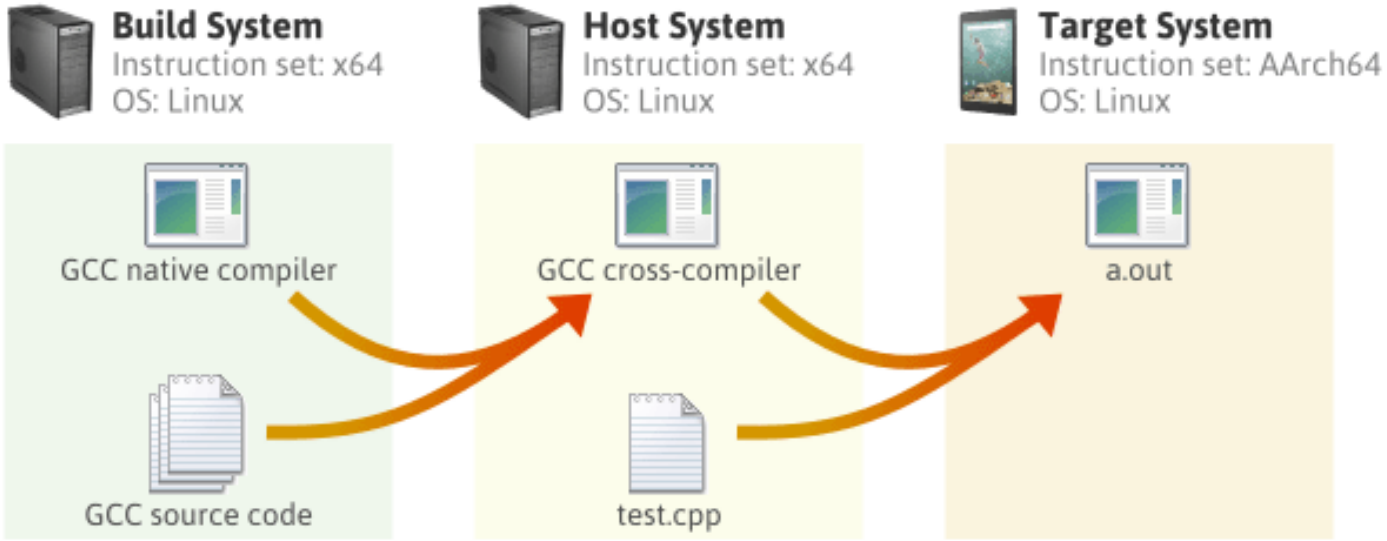
- Antes de compilar, se debe especificar qué arquitectura tiene el target.
- Corresponde a los nombres de directorio dentro de `arch/`
- En nuestro caso, se debe exportar la variable correspondiente.

```
export ARCH=arm
```

- Esto es porque por defecto se asume que al compilar, se lo hace para la arquitectura HOST.

Configuración y compilación

- **IMPORTANTE:** que significan BUILD, HOST, y TARGET



Configuración y compilación

- Retomando, al establecer arquitectura ARM, el sistema BUILD utiliza esta información para:
  - Utilizar las configuraciones correspondientes a esta arquitectura.
  - Compilar el kernel a partir de archivos fuentes y headers correspondientes.
- Toda la configuración de compilación para el kernel está organizada en múltiples makefiles.
- El desarrollador solo interactúa con el Makefile principal (top level - top directory).

Configuración y compilación

- Toda la interacción se da a partir de la herramienta make.
- Se utilizan distintas opciones para indicar qué acciones debe tomar el compilador.
- Las configuraciones que se hacen mediante opciones de make se almacenan en un archivo ***.config***
- Este archivo es de tipo texto plano y contiene las opciones de la siguiente manera:

```
CONFIG_PARAM=value
```

Configuración y compilación

- Frecuentemente las opciones tienen dependencias.
- No se acostumbra a editar a mano el archivo **.config**
- Se hace mediante herramientas gráficas o por consola

```
//Opciones gráficas
make xconfig
make gconfig
```

```
//Opciones de consola
make menuconfig
make nconfig
```

Configuración y compilación

- Estas herramientas se pueden utilizar indistintamente.
- Todas editan el mismo archivo **.config**
- Todas muestran el mismo set de opciones.
- Para el dictado de clases se utiliza texto/consola

```
make menuconfig
```

- El cursante puede utilizar la que prefiera, no altera la compilación.

Configuración y compilación

- Es difícil determinar qué set de opciones funciona para el hardware objetivo.
- Es altamente recomendado comenzar con uno que funcione.
- Para sistemas embebidos, existen configuraciones por defecto para muchas plataformas.
- Se encuentran en **arch/<arch>/configs/**
- Solo son archivos **.config** mínimos, que establecen configuraciones distintas a la por defecto general.

Configuración y compilación

- Para determinar si una plataforma está disponible:

```
ls arch/<arch>/configs
```

- Se listan todos los archivos defconfig disponibles.
- Tendremos dos opciones, según la placa que utilicen:
  - **sunxi_defconfig** para los que utilicen Orange Pi
  - **omap2plus_defconfig** para los que utilicen BeagleBone Black

Configuración y compilación

- Para cargar la configuración por defecto:

```
make <plataforma>_defconfig
```

- NOTA: este comando debe ser ejecutado en el directorio top level.
- Si no se ejecuta en el top level, devuelve un mensaje de error.
- Cada vez que se ejecute esto se sobreescribe el archivo **.config**
- A partir de la configuración base, se modifica el archivo

Configuración y compilación

- Al ser compilado, el kernel es un único archivo, resultante de linkear los archivos objeto creados a partir de la configuración.
- Este único archivo es el que se carga en memoria por el bootloader.
- Esto implica que todas las funcionalidades seleccionadas están disponibles al momento de iniciar el kernel.
- Algunas funcionalidades pueden ser compiladas como módulos.

Configuración y compilación

- Cada módulo se guarda como un archivo separado en el filesystem.
- No se puede acceder a un módulo sin acceso a un filesystem.
- Esto excluye a los módulos de estar disponibles en la etapa temprana de booteo (no hay acceso a filesystem)

Configuración y compilación

- Existen distintos tipos de opciones.
- Opciones ***bool***: true o false dependiendo se desee la funcionalidad en el kernel.
- Opciones ***tristate****:*
  - *true o false* como en el caso anterior
  - *module* para que la funcionalidad se compile en un módulo.
- Opciones tipo ***int***, tipo ***hex*** y tipo ***string***.

Configuración y compilación

- Como se mencionó anteriormente, hay dependencias entre funcionalidades.
- Ejemplo: Habilitar un driver de red requiere el network stack habilitado.
- Existen dos tipos de dependencias:
  - **depends on:** Si la opción B depende de A, no es visible hasta tanto A sea seleccionada.
  - **select:** Si la opción B depende de A, al habilitarse A se selecciona automáticamente B.

Configuración y compilación

HANDS ON

1. Seleccionar una defconfig
2. Explorar distintas configuraciones posibles.

Configuración y compilación

- Un problema muy frecuente es que al cambiar la configuracion, el kernel deja de funcionar.
- Lo usual es no recordar los cambios que se hicieron.
- Para volver a la configuración anterior se utiliza un archivo generado automáticamente.

```
cp .config.old .config
```

- NOTA: solo sirve para la actualización próxima anterior.

Configuración y compilación

- Para compilar el kernel también se utiliza la herramienta make, a partir del archivo Makefile.
- Es necesario setear la variable **CROSS_COMPILE** para que se utilice el compilador correcto.
- Luego de especificar CROSS_COMPILE, el comando invocado será **$(CROSS_COMPILE)gcc**

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
```

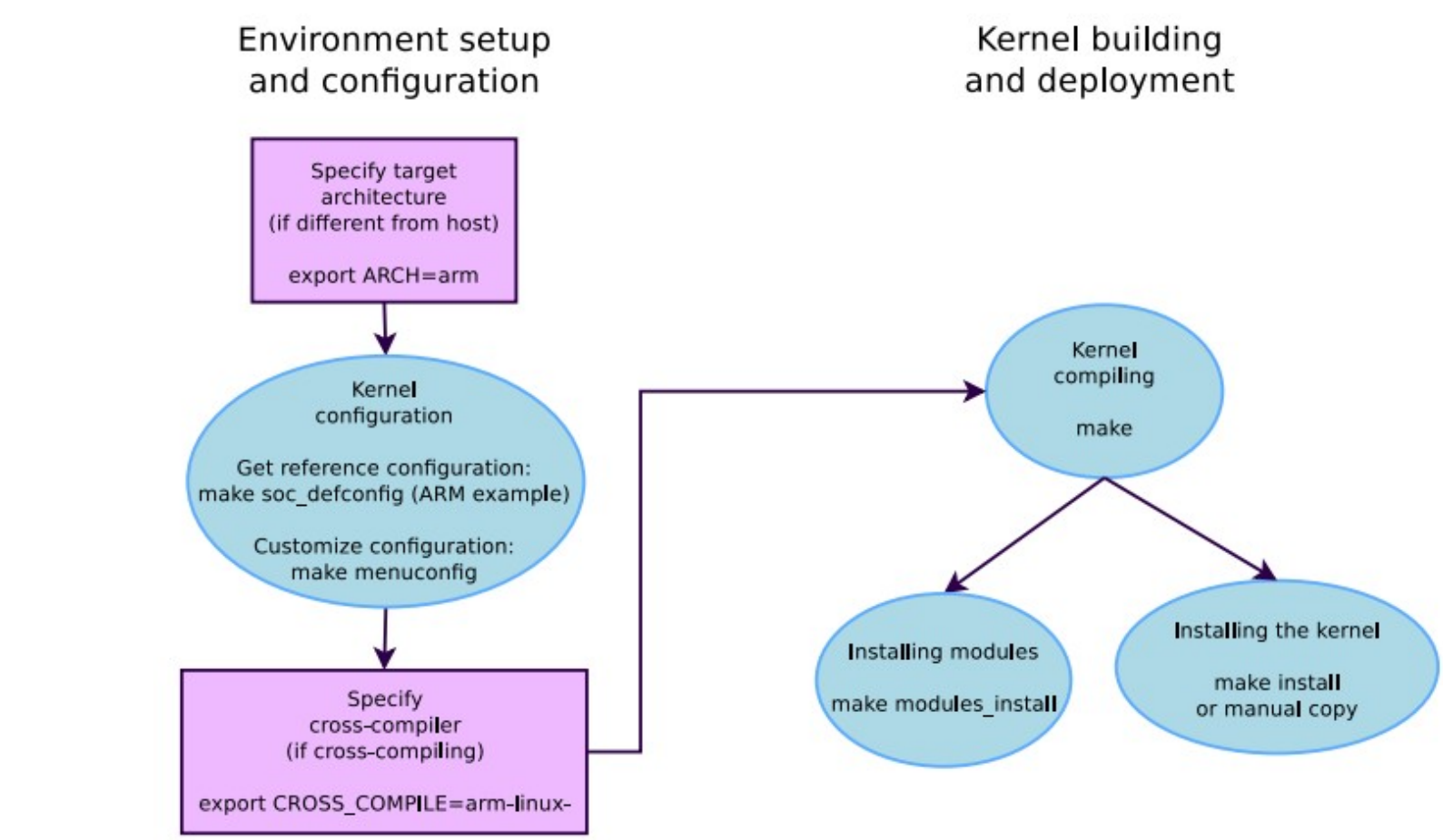
Configuración y compilación

- Si se dejara **CROSS_COMPILE** sin especificar, se utiliza el compilador del sistema HOST (corriendo actualmente).
- **CROSS_COMPILE** se especifica para que los binarios salida puedan ejecutarse en el sistema TARGET.
- Los compiladores para diferenciarse de nativos o del tipo cruzado utilizan prefijos.
- Ese prefijo es lo que se indica al setear **CROSS_COMPILE.**

Configuración y compilación

- La compilación del kernel entonces se hace invocando a make.
- Es usual especificar más de un thread para disminuir el tiempo de compilación.
- Se presume dos hilos por núcleo.
- La compilación genera dos archivos importantes:
  - **zImage** en el directorio **arch/arm/boot/**
  - Device tree blob: **arch/arm/boot/dts/*.dtb**
- Además genera todos los modulos de kernel (***.ko**)

Configuración y compilación



Configuración y compilación

- NOTAS:
  - En sistemas embebidos, no es común ejecutar make install, porque esto instala los archivos en el sistema HOST.
  - Kernel y DTB se copian a mano o mediante scripts.
  - No se puede hacer directamente una instalación de módulos, hay que modificar el path de instalación.

```
make INSTALL_MOD_PATH=<dir>/ modules_install
```

Configuración y compilación

HANDS ON

1. Compilar el kernel.
2. Explorar archivos de salida.

Booting

Booting

- Muchas plataformas embebidas tienen hardware que no puede ser descubierto (non-discoverable).
- Dependiendo la arquitectura, esta clase de hardware se describe mediante:
  - Tablas escritas en la BIOS ACPI (x86).
  - Escrito en C directamente en el kernel.
  - Utilizando un lenguaje especial de descripción de hardware en un Device Tree.

Booting

- En su origen, el Device Tree fue creado para dar soporte a PowerPC.
- Luego fue adoptado para otras arquitecturas (ARM, ARC,....).
- Hoy en día linux tiene soporte en el Device Tree en la mayoría de las arquitecturas para el caso de hardware específico.
- Los archivos fuentes de un Device Tree se compilan en un Device Tree Blob.
- Este archivo DTB es requerido por el kernel para bootear.

Booting

- La placa BeagleBone Black viene precargada con software.
- Contiene U-Boot, que utilizaremos muy frecuentemente.
- U-Boot proviene de *universal bootloader*.
- Es un software muy flexible, utilizado en las plataformas con arquitectura ARM.
- Un bootloader es un programa que ejecuta los pasos necesarios para iniciar el sistema completo.

Booting

- Es el encargado de tomar el kernel y cargarlo en memoria.
- Para esta materia, el kernel y el DTB estarán almacenados en el disco duro del sistema HOST.
- Se hacen disponibles al sistema TARGET a través de NFS (network file system).
- Por medio de comandos de U-Boot, se carga el kernel y el DTB en memoria.
- Por último se ejecuta un comando de U-Boot para iniciar el kernel.

Booting

- Los pasos para bootear entonces son:
  - Cargar el archivo **zImage** en la dirección de memoria X.
  - Cargar el archivo ***.dtb** en la dirección de memoria Y.
  - Iniciar el kernel mediante el comando **bootz X - Y**.
- NOTAS:
  - La dirección de carga de zImage y del archivo *.dtb dependen del hardware.
  - El guión medio entre las direcciones X e Y indica que se inicia sin *initramfs.*

Booting

- Una característica importante del kernel es que pueden pasarse argumentos en tiempo de ejecución.
- La línea de comandos del kernel es un string que define algunas configuraciones:
  - **root=** para el filesystem.
  - **console=** para el direccionamiento de mensajes de kernel.
  - **ip=** para determinar el ip que debe utilizar.
  - **nfsroot=** para indicar la ruta donde esta el filesystem a utilizar.

Booting

- Existen tres maneras de pasar argumentos al kernel:
  a. Mediante el bootloader. U-Boot presenta un string llamada **bootargs** que luego se pasa como argumentos al kernel.
  b. Especificados en el device tree.
  c. Compilados directamente en el kernel utilizando la opción **CONFIG_CMDLINE**.

Booteando desde U-Boot

Boot desde U-boot

- Versiones actuales de U-boot pueden bootear el binario **zImage** producto de la compilación del kernel.
- En versiones más antiguas se requería de un formato especial de imagen: **uImage**.
- **uImage** puede ser generado a partir de **zImage** utilizando la herramienta **mkimage.**
- Puede hacerse de manera automática utilizando el target **make uImage**.

Boot desde U-boot

- En versiones antiguas de algunas plataformas ARM, make uImage requiere una variable LOADADDRR.
- Esta variable de ambiente marca en qué dirección física se ejecutará el kernel.
- Además del kernel, U-boot pasa un device tree blob.
- Booteo tipico:
  - Carga de **zImage** o de **uImage** en la dirección X de RAM
  - Carga de **<board>.dtb** en la dirección Y de RAM.
  - Inicio de kernel con **bootz X - Y**

Boot desde U-boot

- El comportamiento del kernel puede ser modificado en tiempo de ejecución utilizando la línea de comandos del kernel.
- Es una cadena de caracteres definiendo argumentos que utiliza el kernel al iniciar.
- Es muy importante para la configuración del sistema
- **root=** para marcar el root filesystem
- **console=** para indicar dónde irán los mensajes del kernel

Boot desde U-boot

- La línea de comandos puede ser pasada de dos formas:
  - Por el bootloader: En U-Boot se pasan los contenidos de la variable de ambiente **bootargs**
  - Puede estar construida en el mismo kernel, utilizando la opción **CONFIG_CMDLINE**

Boot desde U-boot

HANDS ON

1. Compilar el kernel con el toolchain generado anteriormente.
2. Transferir zImage y *.dtb por TFTP
3. Bootear el kernel.

Gracias.