

BusyBox

Mg. Ing. Gonzalo E. Sanchez
MSE - 2022

Implementación de Sistemas Operativos II

Busybox

- Generalidades

Generalidades

BusyBox

- Un sistema linux necesita un conjunto básico de programas para funcionar:
 - Un programa **init**.
 - Un **shell**.
 - Utilidades básicas varias para manipulación de archivos y configuración de sistema.
- Para sistemas linux normales, estos programas son provistos por diferentes proyectos.

BusyBox

- Proyectos:

- **coreutils, bash, grep, sed, tar, wget, modutils**, etc.
- Implica muchos componentes distintos para integrar.
- Componentes no diseñados para sistemas embebidos (limitaciones).
- No son demasiado configurables y tienen un amplio rango de funcionalidades.

- Busybox es una solución alternativa, sumamente común en sistemas embebidos.

BusyBox

- Hay una re-escritura de comandos Unix muy útiles.
 - Todo integrado en un solo proyecto, haciendo fácil de trabajar con el.
 - Diseñado para sistemas embebidos, altamente configurable sin funcionalidades “innecesarias”.
- Al ser un solo proyecto, todas las utilidades están compiladas en un solo binario ubicado en **/bin/busybox**.
- Se crea un link simbólico para cada aplicación integrada en **busybox**.

BusyBox

- Busybox provee una implementación de **init**.
- Más simple que la implementación de un servidor o PC de escritorio.
- No posee **runlevel** implementado.
- Utiliza un archivo simple de configuración: **/etc/inittab**
- Cada línea tiene la forma **<id>::<action>:<process>**
- Permite que se corran servicios al inicio del sistema, y se asegura que otros estén corriendo siempre.

BusyBox

- Se puede agregar soporte para utilizar **vi**.
- Solo agrega 20K y puede seleccionarse exactamente qué funcionalidades agregar.
- El usuario promedio difícilmente detecta la diferencia entre **vi** y esta versión lightweight.

BusyBox

- El ejecutable de BusyBox puede actuar como si se tratara de varios programas.
- Este comportamiento es según el nombre utilizado para invocar el ejecutable.
- Para que esto funcione, dado que el **.bin** tiene un nombre definido y único, se crean varios symlinks apuntando a él.
- Estos symlinks son los que disparan los distintos comportamientos.

BusyBox

- El método de utilizar un solo binario es principalmente por una cuestión de almacenamiento necesario.
- Un único ejecutable con muchas funcionalidades ocupa menos espacio que muchos archivos pequeños.
- De esta manera BusyBox tiene solo un juego de headers ELF, y puede compartir código fácilmente entre aplicaciones.
- Además la eficiencia de compresión y empaquetamiento es mayor (espacios entre archivos, diccionarios de compresión)

BusyBox

- La ejecución de Busybox comienza con la función **main()** del archivo **libbb/appletlib.c**
- Allí se setea la variable global **applet_name** con el valor **argv[0]** y llama a la función **run_applet_and_exit()**.
- Esto hace uso del array **applets[]** definido en **include/applets.h**
- Se transfiere el control a la función **APPLET_main()** apropiada y ese applet sigue la ejecución.
- Ejemplos **cat_main()**, **sed_main()**, **vi_main()**.

BusyBox

- Así los distintos symlinks logran las distintas llamadas a las funciones que componen Busybox.
- De la misma forma, para agregar un applet se debe:
 - Seleccionar un nombre.
 - Definir el CONFIG_NAME acorde.
 - Buscar entre las fuentes donde podría situarse el código fuente a ser agregado.
 - Asegurarse que se utilice *APPLET_main()* en vez de *main()* donde APPLET es el nombre de la nueva función.

BusyBox

- Siguiendo los pasos anteriores, se debe:
 - Incorporar las directivas, entradas de tablas, instrucciones para make y comentarios para uso correspondientes.
- Una vez hecho todo esto, se debe correr menuconfig, habilitar la nueva funcionalidad y compilar (y debuggear claro....).

Desarrollo de módulos de kernel

HANDS ON

1. Compilar BusyBox.
2. Utilizar el sistema de archivos NFS para cargar Busybox con la SBC.



Gracias.

