

Multimedia embebida con Yocto

Carlos Gutierrez Mata, Daniel Chassoul Orellana, Jorge Schofield Carvajal

26 de septiembre de 2025

Índice

1. Equipo Host	3
1.1. Características	3
1.1.1. Hardware	3
1.1.2. Software	3
2. Toolkit Seleccionado	3
2.1. Características del toolkit	3
2.2. Herramientas de compilación y utilidades	3
2.3. Lenguaje de programación	4
2.4. Bibliotecas y frameworks específicos	4
2.5. Requerimientos del Toolkit	4
2.5.1. Hardware	4
2.5.2. Software	4
2.6. Tabla resumida de dependencias	4
3. Estructura y flujo de trabajo del toolkit seleccionado	4
3.1. Flujo de trabajo para generar la imagen	4
3.2. Diagrama del flujo de trabajo	5
4. Aplicación de ejemplo seleccionada	6
4.1. Descripción general	6
4.2. Diagrama CONOPS	6
4.3. Componentes de la aplicación	7
4.4. Dependencias y librerías requeridas	7
4.5. Flujo interno de la aplicación	7
4.6. Pipeline de GStreamer	8
5. Recetas de Yocto	8
5.1. Estructura básica de una receta	8
5.2. Ejemplo de receta para la aplicación	9
5.3. Dependencias de software	9
6. Selección adecuada del Target Machine	10
7. Proceso de síntesis de la imagen	10
7.1. bitbake-layers create-layer	10
7.2. bitbake-layers add-layer	11
7.3. bitbake nombre-receta	11
7.4. bitbake -c cleanall nombre-receta	11
7.5. bitbake core-image-minimal	12
8. Instalación de la imagen sintetizada sobre el sistema de máquinas virtuales	12

1. Equipo Host

1.1. Características

1.1.1. Hardware

- **Procesador:** Procesadores x86 de 4 a 6 cores.
- **Memoria RAM:** se utilizó de 12 GB a 16 GB.
- **Espacio en disco:** 100 GB o más.

1.1.2. Software

- **Sistema operativo:** Ubuntu 22.04/24.04 LTS.
- **Poky:** versión Walnascar.

2. Toolkit Seleccionado

2.1. Características del toolkit

El toolkit seleccionado para este proyecto es Yocto, que permite generar imágenes embebidas personalizadas de forma reproducible y eficiente. Entre sus principales características destacan:

- Modularidad: permite agregar o eliminar paquetes y capas según las necesidades del proyecto.
- Compatibilidad con múltiples arquitecturas de hardware.
- Integración con herramientas de compilación y gestión de dependencias.
- Flexibilidad para incluir aplicaciones y bibliotecas de terceros.

2.2. Herramientas de compilación y utilidades

Para que Yocto pueda generar imágenes correctamente, es necesario instalar un conjunto de paquetes esenciales en el sistema host. En distribuciones basadas en Ubuntu, estos paquetes se instalan con el siguiente comando:

```
sudo apt-get install build-essential chrpath cpio debianutils diffstat file  
gawk gcc git iputils-ping libacl1 liblz4-tool locales python3 python3-git  
python3-jinja2 python3-pexpect python3-pip python3-subunit
```

Estas herramientas permiten:

- Compilación de recetas y programas (`gcc`, `build-essential`).
- Manipulación de archivos y sistemas de empaquetado (`cpio`, `file`).
- Automatización y scripting (`gawk`, `diffstat`).
- Gestión de dependencias en Python (`python3`, `python3-pip`, `python3-jinja2`).
- Control de versiones (`git`).

2.3. Lenguaje de programación

La aplicación principal está implementada en Python3, por lo que es necesario incluir este intérprete en la imagen. En `local.conf` se añadió la línea:

```
IMAGE_INSTALL:append = " python3"
```

2.4. Bibliotecas y frameworks específicos

La aplicación requiere bibliotecas adicionales para visión por computadora y procesamiento multimedia: OpenCV, PyGObject y GStreamer con sus plugins base, good, bad y ugly. Estas dependencias se declararon en `local.conf` de la siguiente manera:

```
IMAGE_INSTALL:append = " python3-opencv python3-pyobject \  
                        gstreamer1.0 gstreamer1.0-libav \  
                        gstreamer1.0-plugins-base \  
                        gstreamer1.0-plugins-good \  
                        gstreamer1.0-plugins-bad \  
                        gstreamer1.0-plugins-ugly "
```

2.5. Requerimientos del Toolkit

2.5.1. Hardware

- **Procesador:** CPU moderna de 4 núcleos o superior.
- **Memoria RAM:** mínimo 8 GB, recomendado 16 GB.
- **Espacio en disco:** al menos 100 GB libres para compilaciones y cachés.

2.5.2. Software

- **Sistema operativo:** Cualquier distribución de Linux, recomendado si es una versión estable.
- **Git:** versión 1.8.3.1 o mayor.
- **Tar:** versión 1.28 o mayor.
- **Python:** versión 3.90 o mayor.
- **Gcc:** versión 10.1 o mayor.
- **GNU:** versión 4.0 o mayor.

2.6. Tabla resumida de dependencias

A continuación se resumen las dependencias del proyecto, su propósito y cómo se integran:

3. Estructura y flujo de trabajo del toolkit seleccionado

3.1. Flujo de trabajo para generar la imagen

El flujo de trabajo seguido para generar imágenes embebidas con Yocto se describe a continuación:

1. **Preparación del host:** Instalación de paquetes esenciales para compilación y gestión de dependencias.

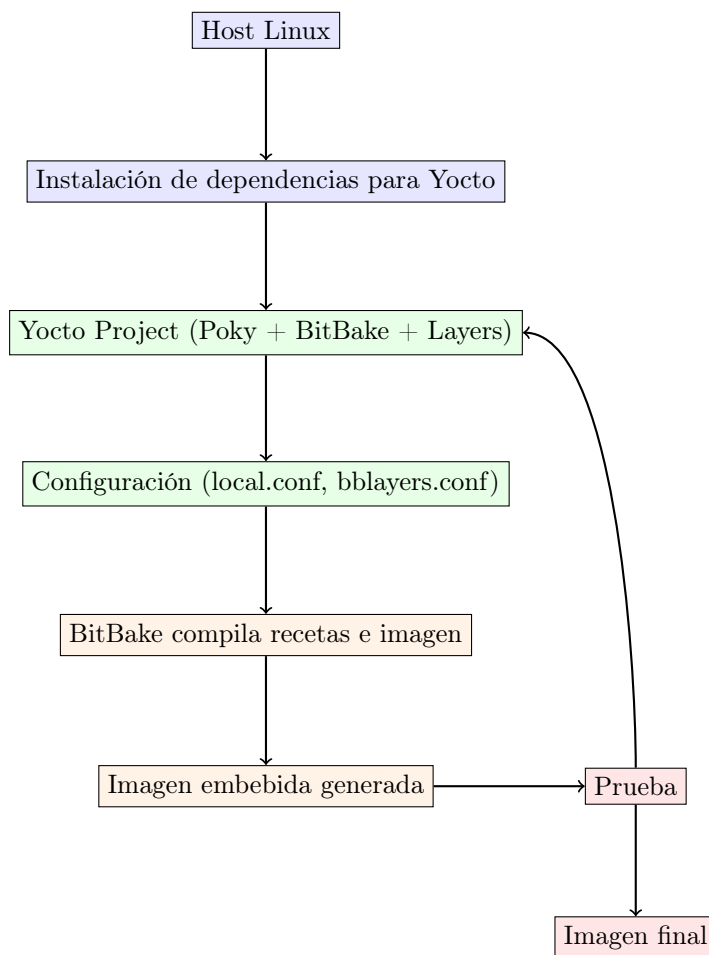
Dependencia	Propósito	Origen / Integración
Herramientas de compilación	Construcción de recetas Yocto	Host
Python3	Lenguaje de la aplicación	local.conf
OpenCV	Visión por computadora	local.conf
PyGObject	Enlace Python-GStreamer	local.conf
GStreamer + plugins	Procesamiento multimedia	local.conf

Tabla 1: Resumen de dependencias del proyecto

2. **Configuración del ambiente de compilación:** Ejecución de `source oe-init-build-env` para establecer variables de entorno y directorios de trabajo.
3. **Configuración de capas:** Modificación del archivo `bblayers.conf` para incluir capas necesarias que contienen recetas y configuraciones adicionales.
4. **Configuración de la imagen:** Modificación de `local.conf` para definir paquetes generales y dependencias requeridas por la imagen.
5. **Creación de recetas personalizadas:** El toolkit permite definir nuevas recetas para incluir paquetes o aplicaciones específicas dentro de la imagen.
6. **Construcción de recetas:** Uso de `bitbake <nombre-de-receta>` para compilar cada receta individualmente.
7. **Construcción de la imagen:** Ejecución de `bitbake <nombre-de-imagen>` para generar la imagen completa.
8. **Verificación de la imagen generada:** Inspección de los archivos de imagen generados en el directorio de despliegue del `/poky/build/tmp/deploy/images/qemux86-64/`.

3.2. Diagrama del flujo de trabajo

Este flujo describe cómo se utiliza Yocto para generar imágenes embebidas de manera modular y reproducible, permitiendo agregar o actualizar recetas y paquetes según los requerimientos del proyecto. En el siguiente flujo de trabajo se puede representar gráficamente de forma general:



4. Aplicación de ejemplo seleccionada

4.1. Descripción general

La aplicación de ejemplo implementada dentro de la imagen embebida tiene como objetivo principal la detección de rostros en un video utilizando visión por computadora. Está diseñada para procesar secuencias de video en bloques, detectar rostros y ojos en cada frame, y empaquetar la cantidad de rostros detectados en un archivo de resultados `.txt`.

4.2. Diagrama CONOPS

El diagrama 1 representa como la aplicación al detectar un límite de rostros dispara una alerta indicando la cantidad actual de rostros detectados.

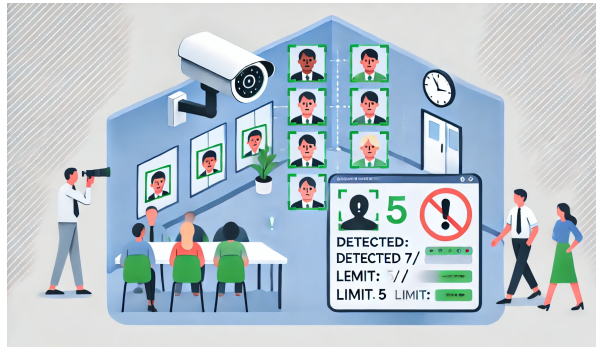


Figura 1: Diagrama CONOPS de la aplicación

4.3. Componentes de la aplicación

La aplicación está compuesta por los siguientes elementos:

- **Código fuente:** `reconoceRostros.py`, que implementa la lógica de captura de video, procesamiento de frames y detección de rostros.
- **Clasificadores Haar:** Archivos XML (`haarcascade_frontalface_default.xml` y `haarcascade_eye.xml`) utilizados para detectar rostros y ojos.
- **Entrada multimedia:** Video de prueba (`video.mp4`) sobre el cual se realiza el análisis y procesamiento.
- **Salida de resultados:** Archivo de texto (`rostros_totales.txt`) que registra el número de rostros detectados por frame.

4.4. Dependencias y librerías requeridas

Para su correcto funcionamiento, la aplicación requiere:

- Python3 como lenguaje de ejecución.
- OpenCV para procesamiento de imágenes y detección de rostros y ojos.
- GStreamer y PyGObject para la captura y conversión del video.
- Librerías estándar de NumPy para manipulación de matrices de pixels.

4.5. Flujo interno de la aplicación

El procesamiento interno de la aplicación se puede resumir en los siguientes pasos:

1. Ubicación e inicialización de los detectores Haar para rostros y ojos.
2. Configuración del pipeline de GStreamer para leer el video de entrada.
3. Lectura de frames en bloques (según `BLOCK_SIZE`) para optimizar el procesamiento.
4. Conversión de cada frame a escala de grises y detección de rostros.
5. Para cada rostro detectado, verificación de la presencia de al menos dos ojos mediante el detector correspondiente.

6. Registro de la cantidad de rostros válidos detectados por frame en un arreglo interno.
7. Escritura de los resultados finales en un archivo de texto (`rostros_totales.txt`).
8. Liberación de recursos y cierre del pipeline de video.

4.6. Pipeline de GStreamer

El pipeline configurado es el siguiente:

```
filesrc location=/usr/share/video2.mp4 ! decodebin ! videoconvert !  
video/x-raw,format=BGR ! appsink name=sink emit-signals=true
```

Descripción de los elementos del pipeline:

- `filesrc location=...`: Fuente de archivos que especifica la ruta del video de entrada.
- `decodebin`: Elemento automático que detecta el formato del video y aplica la decodificación adecuada.
- `videoconvert`: Convierte los frames a un formato de pixel estándar compatible con OpenCV.
- `video/x-raw, format=BGR`: Define el formato de video de salida como imagen raw en color BGR, compatible con OpenCV.
- `appsink name=sink emit-signals=true`: Permite que la aplicación reciba los frames uno por uno desde Python, habilitando su procesamiento.

Integración con OpenCV: Cada frame extraído del pipeline se convierte a un array de NumPy, lo que permite utilizar las funciones de OpenCV para:

- Convertir la imagen a escala de grises.
- Detectar rostros y ojos utilizando clasificadores Haar.
- Contabilizar y almacenar los resultados de detección por frame.

Manejo de buffers: Para evitar la acumulación de frames en memoria y asegurar el procesamiento completo del video:

- Se establece `max-buffers=1` en `appsink`, limitando el buffer a un solo frame.
- Se activa `drop=True`, lo que permite descartar frames antiguos si la aplicación no puede procesarlos a tiempo.

5. Recetas de Yocto

5.1. Estructura básica de una receta

Las recetas (documentos `.bb`) son archivos con instrucciones para el Bitbake que le indican cómo configurar, compilar, empaquetar y buscar otros archivos. Las recetas se encuentran dentro de las capas y siguen una estructura básica, por ejemplo:

- Metadata
- Búsqueda de archivos fuente

- Instrucciones de BitBake
- Empaquetamiento

También, se puede agregar especificaciones para el versionamiento de los programas y herencias de otras recetas. Por otro lado, para que el BitBake pueda encontrar las recetas, es necesario agregar un archivo extra llamado *layer.conf*, donde se especifica la dirección de las recetas.

5.2. Ejemplo de receta para la aplicación

Para agregar el archivo de Python de la aplicación se utilizó la receta siguiente, donde se establece su dirección y se le indica al BitBake dónde instalarlo:

```
SUMMARY = "Rostros"
LICENSE = "CLOSED"

SRC_URI += "file://reconoceRostros.py"

S = "${UNPACKDIR}"

do_install() {
    install -d ${D}${bindir}
    install -m 0755 ${S}/reconoceRostros.py ${D}${bindir}/reconoceRostros
}
```

Se utilizó esta misma estructura de la receta para agregar los archivos xml y el video de prueba, todos necesarios para la ejecución del programa.

Además, se configuró el archivo de capa de la siguiente manera:

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
           ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-custom"
BBFILE_PATTERN_meta-custom = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-custom = "6"

LAYERDEPENDS_meta-custom = "core"
LAYERSERIES_COMPAT_meta-custom = "walnascar"
```

5.3. Dependencias de software

Como se ha descrito anteriormente, la aplicación creada depende de varios paquetes de software y estos no se encuentran incluidos en la capa principal de Poky. Por esta razón, se utilizan otros grupos de capas, donde se encuentran paquetes comúnmente usados. En este caso, se incluyeron varias capas de *meta-openembedded* y la capa de *meta-yocto-bsp*, que contienen las bibliotecas necesarias para el proyecto.

```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""
```

```

BBLAYERS += " \
    ${TOPDIR}/poky/meta \
    ${TOPDIR}/poky/meta-poky \
    ${TOPDIR}/poky/meta-yocto-bsp \
    ${TOPDIR}/poky/meta-openembedded/meta-oe \
    ${TOPDIR}/poky/meta-openembedded/meta-multimedia \
    ${TOPDIR}/poky/meta-openembedded/meta-python \
    ${TOPDIR}/poky/meta-custom \
"

```

6. Selección adecuada del Target Machine

Para este proyecto se seleccionó el Target Machine 'genericx86-64' con el fin de crear una imagen bootable para VirtualBox. Esta selección se realiza editando el archivo `local.conf` como se muestra en la Fig. 2. Es importante destacar como el resto de target machines se mantienen comentadas mediante símbolo de `#`, logrando así que el `local.conf` solo lea la máquina objetivo al ser la única que se encuentra sin comentar.

```

#
# Machine Selection
#
# You need to select a specific machine to target the build with. There are a selection
# of emulated machines available which can boot and run in the QEMU emulator:
#
#MACHINE ?= "gemuarm"
#MACHINE ?= "gemuarm64"
#MACHINE ?= "gemumips"
#MACHINE ?= "gemumips64"
#MACHINE ?= "gemuppc"
#MACHINE ?= "gemux86"
#MACHINE ?= "gemux86-64"
#
# There are also the following hardware board target machines included for
# demonstration purposes:
#
#MACHINE ?= "beaglebone-yocto"
#MACHINE ?= "genericarm64"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#
# This sets the default machine to be gemux86-64 if no other machine is selected:
MACHINE ?= "genericx86-64"

```

Fig. 2: Selección del target machine

7. Proceso de síntesis de la imagen

7.1. bitbake-layers create-layer

Este comando permite crear una capa dentro de la carpeta de poky con la estructura adecuada de directorios para una nueva capa. Para el caso de la imagen sintetizada se creó la capa llamada *meta-custom* utilizando el comando *bitbake-layers create-layer meta-custom*. En esta capa se incorporaron las

recetas necesarias para incorporar el script de python de la aplicación, el video en formato .mp4 y dos archivos .xml necesarios para el funcionamiento de la aplicación.

```
daniel@daniel-IdeaPad-3-15ITL05:~/poky$ bitbake-layers create-layer meta-custom
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer meta-custom'
```

Fig. 3: Creación de la capa meta-custom mediante bitbake

7.2. bitbake-layers add-layer

Al ejecutar el comando *bitbake-layers add-layer meta-custom* permitió añadir la capa ya creada anteriormente a la build actual, modificando de manera automática el archivo *bblayers.conf* para agregar dicha capa a la lista de capas a recorrer al momento de sintetizar la imagen.

```
daniel@daniel-IdeaPad-3-15ITL05:~/poky$ bitbake-layers add-layer meta-custom
NOTE: Starting bitbake server...
daniel@daniel-IdeaPad-3-15ITL05:~/poky$
```

Fig. 4: Adición de la capa meta-custom mediante bitbake

7.3. bitbake nombre-receta

Permite realizar el análisis y parsing de una receta en específico dentro de las capas configuradas. Facilita el proceso de compilación, ya que cada vez que se modifica una receta en particular se realiza el bitbake de esa receta en específico para observar si hay errores o no, en lugar de realizar el bitbake de la imagen total. Este proceso se realizó para todas las recetas incluidas dentro de la capa *meta-custom*. En la Fig. 5 se muestra el proceso para la receta correspondiente a reconoce-rostros, es importante mencionar que este proceso se realiza también para la receta del archivo xml y las dos recetas de los videos mp4.

```
Build Configuration:
BB_VERSION           = "2.12.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-22.04"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "genericx86-64"
DISTRO               = "poky"
DISTRO_VERSION        = "5.2.3"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp        = "walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"
meta-oe
meta-multimedia
meta-python           = "walnascar:dca497d728792e3cb655c78455c2d649af312ce8"
meta-custom           = "walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"
```

Fig. 5: Bitbake para la receta reconoce-rostros

7.4. bitbake -c cleanall nombre-receta

Cada vez que se modificaba una receta que ya había sido compilada anteriormente se hacía uso de este comando con el fin de eliminar todos los artefactos generados por la receta en específico, forzando la recompilación completa de la receta con el fin de evitar conflictos con versiones pasadas de la receta.

Nuevamente, en la Fig. 6 se muestra el proceso de limpieza para la receta reconoce-rostros, sin embargo, este proceso también se realizó en cada receta a la hora de realizar cambios.

```
daniel@daniel-IdeaPad-3-15ITL05:~/poky$ bitbake -c cleanall reconoce-rostros

Loading cache: 100% |#####| Time: 0:00:01
Loaded 4648 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "2.12.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "universal"
TARGET_SYS      = "x86_64-poky-linux"
MACHINE         = "qemux86-64"
DISTRO          = "poky"
DISTRO_VERSION  = "5.2.3"
TUNE_FEATURES   = "m64 core2"
TARGET_FPU      = ""
meta
meta-poky
meta-yocto-bsp  = "my-walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"
meta-oe
meta-multimedia
meta-python    = "walnascar:dca497d728792e3cb655c78455c2d649af312ce8"
meta-custom    = "my-walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"

Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)#####
#####| ETA: 0:00:00
Initialising tasks: 100% |#####| Time: 0:00:00
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3 tasks of which 0 didn't need to be rerun and all succeeded.
daniel@daniel-IdeaPad-3-15ITL05:~/poky$
```

Fig. 6: Bitbake para la limpieza de la receta reconoce-rostros

7.5. bitbake core-image-minimal

Comando que permite la compilación de la imagen mínima para su sintetización y posterior incorporación en virtual box.

8. Instalación de la imagen sintetizada sobre el sistema de máquinas virtuales

Una vez generada la imagen mínima mediante BitBake, es necesario correrla sobre el entorno VirtualBox. Para esto, primero se abre VirtualBox y se escoge la opción 'New' con el fin de crear una nueva máquina virtual. Una vez realizado esto saltará una ventana de configuración donde se escoge el nombre de la máquina, su sistema operativo y la versión de dicho sistema operativo. En la Fig 8 se muestran las configuraciones seleccionadas para la imagen.

Una vez realizada la configuración de la pestaña 'Virtual machine name and operating system' es necesario seleccionar la pestaña 'Specify virtual hard disk', al hacerlo se observa la ventana de configuración de la Fig. 9.

Dentro de esta pestaña se selecciona la opción 'Use an existing virtual hard disk file', donde se abrirá la ventana de la Fig. 10 y se tiene que añadir el archivo .vmdk generado al sintetizar la imagen para lo cual es necesario seleccionar el ícono de 'Add'.

Una vez seleccionada esta opción es necesario recorrer los directorios de la manera indicada en la Fig. 11 para encontrar el lugar donde se guardó el archivo .vmdk

Al llegar a la carpeta 'genericx86-64' se observa los archivos vmdk que han sido generados mediante la síntesis de la imagen, se escoge el adecuado y la configuración de la pestaña 'Specify virtual hard disk' queda como se muestra en la Fig. 12.

```

jschoft@jschoft:~/poky/build$ bitbake core-image-minimal
Loading cache: 100% |#####| ETA: --:--:--
Loaded 0 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:55
Parsing of 2672 .bb files complete (0 cached, 2672 parsed). 4649 targets, 109 skipped, 0 masked, 0 errors.
Removing 11 recipes from the allarch sysroot: 100% |#####| Time: 0:00:01
Removing 177 recipes from the core2-64 sysroot: 100% |#####| Time: 0:00:09
Removing 189 recipes from the genericx86_64 sysroot: 100% |#####| Time: 0:00:07
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "2.12.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "ubuntu-22.04"
TARGET_SYS      = "x86_64-poky-linux"
MACHINE         = "genericx86-64"
DISTRO          = "poky"
DISTRO_VERSION  = "5.2.3"
TUNE_FEATURES   = "m64 core2"
TARGET_FPU      = ""
meta
meta-poky
meta-yocto-bsp   = "walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"
meta-oe
meta-multimedia
meta-python     = "walnascar:dca497d728792e3cb655c78455c2d649af312ce8"
meta-custom     = "walnascar:5495d8b6ff75e625e78b94503acbb35d0247709c"

NOTE: Fetching uninitative binary shim http://downloads.yoctoproject.org/releases/uninitative/4.9/x86_64-nativesdk-libc-4.9.tar.xz;sha25
6sum=4c03d1ed2b7b4e823aca4a1a23d8f2e322f1770fc10e859adcede5777aff4f3a (will check PREMIRRORS first)
Sstate summary: Wanted 3505 Local 2 Mirrors 0 Missed 3503 Current 422 (0% match, 10% complete)
Removing 750 stale sstate objects for arch x86_64: 100% |#####| Time: 0:00:02
NOTE: Executing Tasks
Setscene tasks: 3927 of 3927
Currently 3 running tasks (8386 of 8567) 97% |#####|
NOTE: Tasks Summary: Attempted 8567 tasks of which 425 didn't need to be rerun and all succeeded.

```

Fig. 7: Bitbake para la compilación de imagen mínima

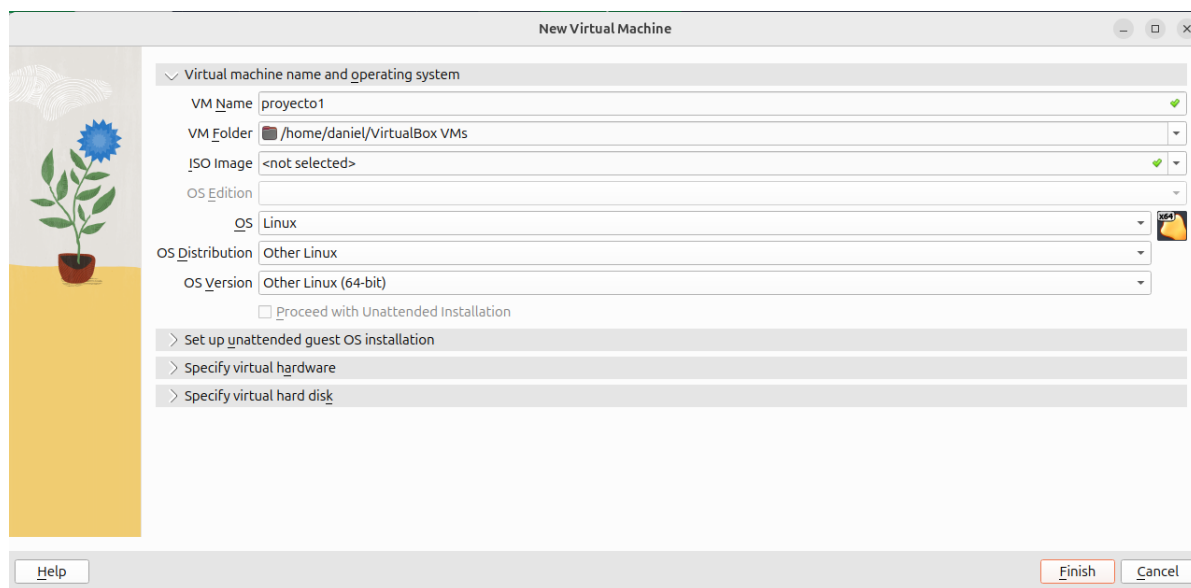


Fig. 8: Configuración inicial de la máquina virtual

Ya con esto simplemente se selecciona el botón de 'finish' y se creará la máquina virtual tal y como se observa en la Fig. 13. Antes de encender la máquina virtual es necesario ir hacia 'Settings' y en la pestaña de 'System' se marca la opción de UEFI con el fin de poder correr correctamente la imagen generada, esto se muestra en la Fig. 14.

Finalmente, se realiza un doble click sobre la máquina virtual y esta empezará a recorrer en el entorno de VirtualBox. La máquina virtual funcional se observa en la Fig. 15.

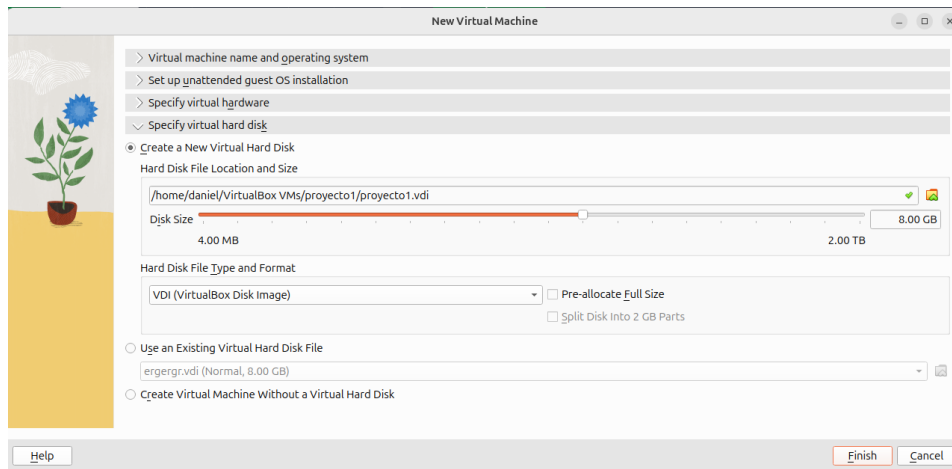


Fig. 9: Pestaña de configuración para el disco duro virtual

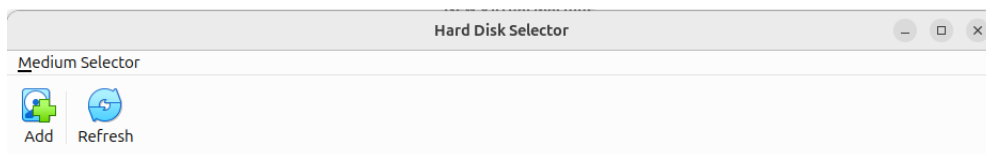


Fig. 10: Pestaña de configuración para el disco duro virtual



Fig. 11: Recorrido de directorios

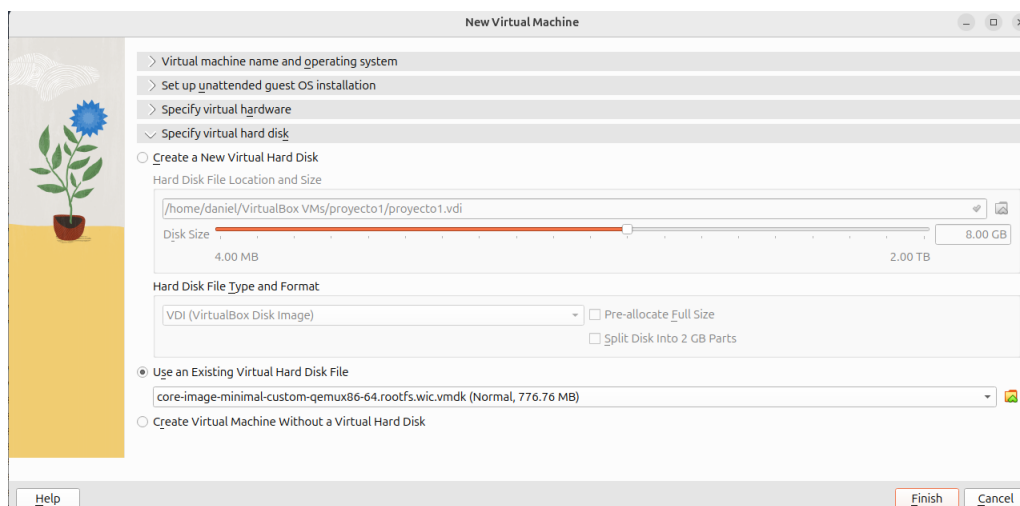


Fig. 12: Configuración final de la pestaña Virtual hard disk

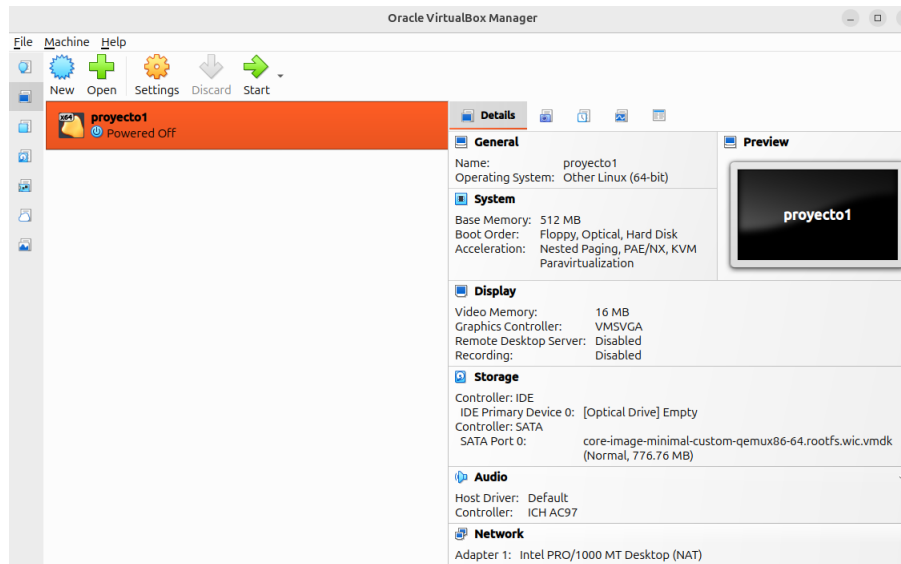


Fig. 13: Ventana con la máquina virtual creada

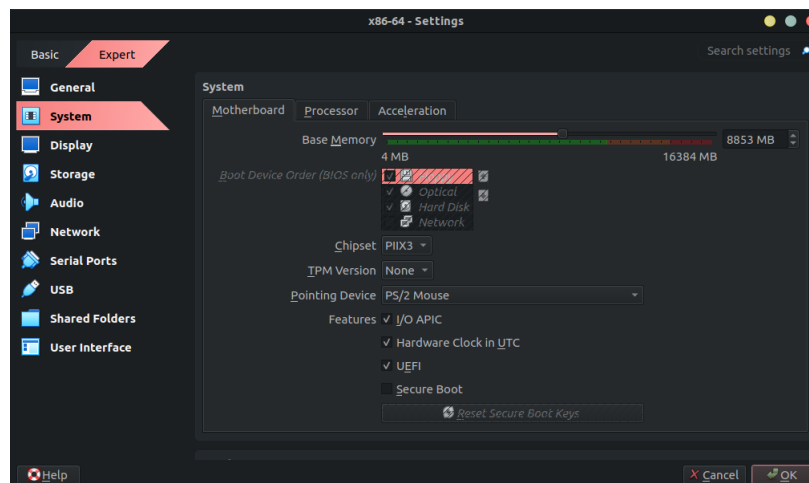
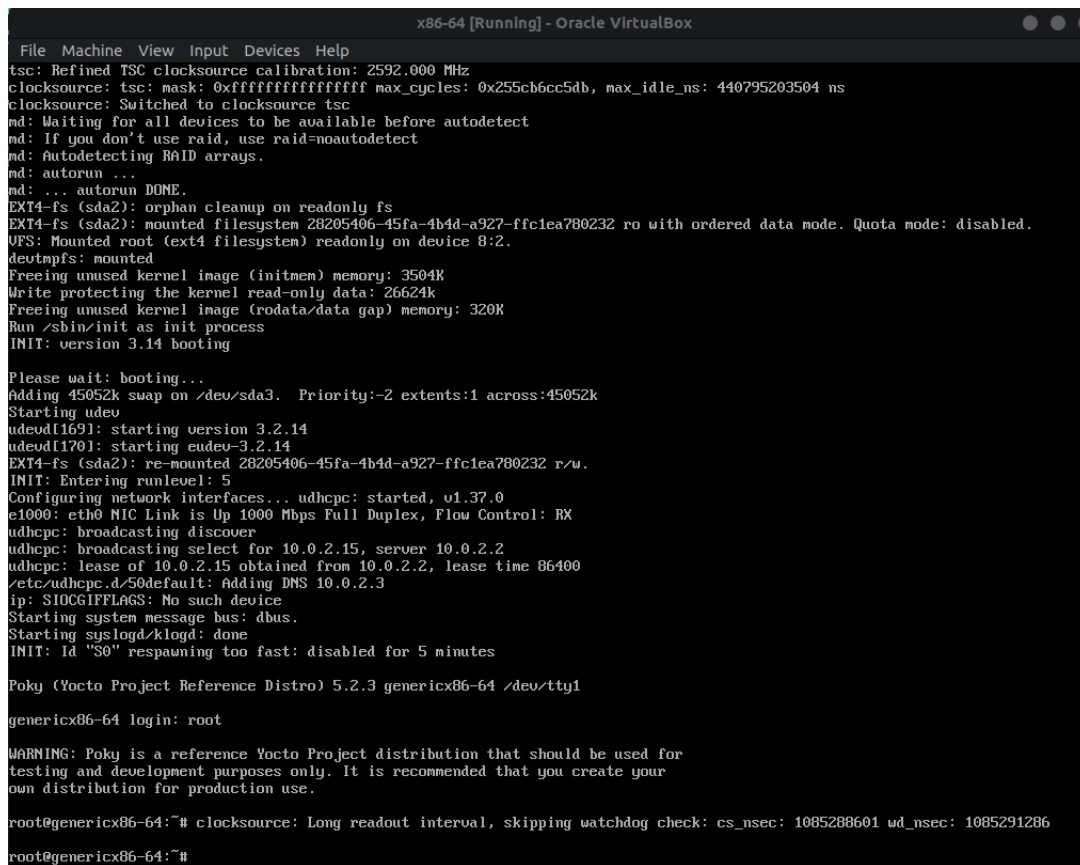


Fig. 14: Activación del UEFI en Settings



```
File Machine View Input Devices Help
tsc: Refined TSC clocksource calibration: 2592.000 MHz
clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x255cb6cc5db, max_idle_ns: 440795203504 ns
clocksource: Switched to clocksource tsc
md: Waiting for all devices to be available before autodetect
md: If you don't use raid, use raid=noautodetect
md: Autodetecting RAID arrays.
md: autorun ...
md: ... autorun DONE.
EXT4-fs (sda2): orphan cleanup on readonly fs
EXT4-fs (sda2): mounted filesystem 28205406-45fa-4b4d-a927-ffc1ea780232 ro with ordered data mode. Quota mode: disabled.
VFS: Mounted root (ext4 filesystem) readonly on device 8:2.
devtmpfs: mounted
Freeing unused kernel image (initmem) memory: 3504K
Write protecting the kernel read-only data: 26624k
Freeing unused kernel image (rodata/data gap) memory: 320K
Run /sbin/init as init process
INIT: version 3.14 booting

Please wait: booting...
Adding 45052k swap on /dev/sda3. Priority:-2 extents:1 across:45052k
Starting udev
udevd[169]: starting version 3.2.14
udevd[170]: starting eudev-3.2.14
EXT4-fs (sda2): re-mounted 28205406-45fa-4b4d-a927-ffc1ea780232 r/w.
INIT: Entering runlevel: 5
Configuring network interfaces... udhcpc: started, v1.37.0
e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
udhcpc: broadcasting discover
udhcpc: broadcasting select for 10.0.2.15, server 10.0.2.2
udhcpc: lease of 10.0.2.15 obtained from 10.0.2.2, lease time 86400
/etc/udhcpc.d/50default: Adding DNS 10.0.2.3
ip: SIOCGIFFLAGS: No such device
Starting system message bus: dbus.
Starting syslogd/klogd: done
INIT: Id "S0" respawning too fast: disabled for 5 minutes

Poky (Yocto Project Reference Distro) 5.2.3 genericx86-64 /dev/tty1
genericx86-64 login: root

WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

root@genericx86-64:~# clocksource: Long readout interval, skipping watchdog check: cs_nsec: 1085288601 wd_nsec: 1085291286
root@genericx86-64:~#
```

Fig. 15: Máquina virtual funcionando en VirtualBox