Unidad Didáctica 2:
TRABAJO EN SEGUNDO PLANO

Programación Dirigida por Eventos

UAX Universidad Alfonso X el Sabio

ÍNDICE

Contenido

1. Introducción

¡Bienvenido a la asignatura Programación Dirigida por Eventos!

En esta unidad didáctica, nos enfocaremos en el "Trabajo en segundo plano". Aprenderás a manejar tareas que no deben ejecutarse en el hilo principal de la aplicación para no bloquear la interfaz de usuario. Esto incluye el uso de AsyncTask y Loaders para ejecutar tareas en segundo plano de manera eficiente y segura. También se explorará el manejo de hilos, la utilización de servicios para tareas prolongadas y la implementación de receptores de broadcast.

Ten en cuenta:

Es crucial entender la importancia de mantener el hilo principal libre de operaciones intensivas para garantizar una experiencia de usuario fluida. Aprenderás a utilizar herramientas y técnicas que permiten delegar tareas complejas a hilos de fondo, mejorando así la eficiencia y la capacidad de respuesta de tus aplicaciones.

Temas que se tratarán en esta unidad:

1. **AsyncTask y AsyncTaskLoader en Android:** Cómo ejecutar tareas en segundo plano y actualizar la interfaz de usuario al completar estas tareas.
2. **Hilos (Threads):** Conceptos básicos y avanzados sobre el uso de hilos en Android para realizar tareas simultáneas.
3. **El Hilo Principal:** Comprensión del ciclo de vida del hilo principal y cómo evitar su bloqueo.
4. **Ejecución de Tareas en Hilos de Fondo:** Estrategias para mover tareas pesadas fuera del hilo principal.
5. **Loaders:** Implementación y ventajas de usar Loaders para la carga asíncrona de datos.
6. **Broadcasts en Android:** Envío y recepción de mensajes del sistema y aplicaciones mediante broadcasts.
7. **Alarmas y Programadores:** Uso de AlarmManager para programar tareas en momentos específicos.
8. **Transferencia Eficiente de Datos y JobScheduler:** Técnicas para la transferencia de datos eficiente y programación de tareas con JobScheduler.

Al finalizar esta unidad, deberías ser capaz de implementar tareas en segundo plano de manera eficaz, garantizando una experiencia de usuario óptima en tus aplicaciones Android.

2. Objetivos

Objetivo general:

- Proporcionar a los estudiantes las habilidades necesarias para gestionar tareas en segundo plano en aplicaciones Android, utilizando técnicas y herramientas adecuadas para mantener la eficiencia y la capacidad de respuesta de la aplicación.

Objetivos específicos:

En esta unidad se establecen tres objetivos específicos:

1. **Comprender los fundamentos del trabajo en segundo plano en Android:** Aprender a utilizar AsyncTask y Loaders para ejecutar tareas sin bloquear el hilo principal.
2. **Implementar técnicas de manejo de hilos y servicios:** Desarrollar la capacidad de usar hilos y servicios para tareas prolongadas, asegurando que las aplicaciones sigan siendo receptivas.
3. **Optimizar la transferencia de datos y programación de tareas:** Aplicar técnicas de JobScheduler y AlarmManager para programar y ejecutar tareas de manera eficiente, minimizando el consumo de recursos y mejorando el rendimiento de las aplicaciones.

3. Tareas en segundo plano

3.1. AsyncTask y AsyncTaskLoader en Android

Hilos (Threads)

Hilo Principal

El hilo principal, también conocido como hilo de UI, es el hilo en el que se ejecuta la aplicación Android:

- **Ejecuta el código línea por línea**
- **Dibuja la interfaz de usuario (UI) en la pantalla**
- **Responde a las acciones del usuario manejando eventos de UI**

Importancia de la Velocidad del Hilo Principal

El hardware actualiza la pantalla cada 16 milisegundos, por lo que el hilo de UI tiene 16 ms para realizar todo su trabajo. Si el hilo principal tarda más, la aplicación se entrecorta o se cuelga, lo que puede llevar a que los usuarios desinstalen la aplicación.

Tareas de Larga Duración

Las tareas de larga duración incluyen:

- Operaciones de red
- Cálculos extensivos
- Descarga o carga de archivos
- Procesamiento de imágenes
- Carga de datos

Estas tareas deben ejecutarse en un hilo de fondo para evitar bloquear el hilo de UI.

Ejecución de Tareas en Hilos de Fondo

Las tareas de larga duración deben ejecutarse en hilos de fondo. Para ello, se pueden usar AsyncTask, el marco de Loader y los servicios.

Reglas para Hilos en Android

- **No bloquear el hilo de UI**: Realiza el trabajo no relacionado con la UI en un hilo de fondo.
- **No acceder a la UI desde hilos de fondo**: Realiza el trabajo de UI solo en el hilo de UI.

AsyncTask

¿Qué es AsyncTask?

AsyncTask se utiliza para implementar tareas básicas en segundo plano. Simplifica la creación de hilos y permite actualizar la UI después de completar la tarea de fondo.

Métodos Principales

- **doInBackground()**: Se ejecuta en un hilo de fondo y realiza todo el trabajo en segundo plano.
- **onPostExecute()**: Se ejecuta en el hilo principal cuando se completa la tarea de fondo y procesa los resultados.

Métodos Auxiliares

- **onPreExecute()**: Se ejecuta en el hilo principal antes de iniciar la tarea de fondo y configura la tarea.
- **onProgressUpdate()**: Se ejecuta en el hilo principal y recibe llamadas desde publishProgress() del hilo de fondo.

Creación de un AsyncTask

Para crear un AsyncTask, sigue estos pasos:

1. **Subclase AsyncTask**:

```
private class MyAsyncTask extends AsyncTask<URL, Integer, Bitmap> {
   // Implementar métodos
}
```

2. **Implementar onPreExecute()**:

```
protected void onPreExecute() {
   // Mostrar una barra de progreso o un mensaje
}
```

3. **Implementar doInBackground()**:

```
protected Bitmap doInBackground(String... query) {
   // Obtener el bitmap
   return bitmap;
}
```

4. **Implementar onProgressUpdate()**:

```
protected void onProgressUpdate(Integer... progress) {
   // Actualizar el progreso
}
```

5. **Implementar onPostExecute()**:

```
protected void onPostExecute(Bitmap result) {
   // Procesar el resultado
}
```

Limitaciones de AsyncTask

- Cuando la configuración del dispositivo cambia, la actividad se destruye y AsyncTask no puede volver a conectarse a la actividad.
- Se crea un nuevo AsyncTask para cada cambio de configuración.
- Los AsyncTasks antiguos permanecen, lo que puede causar fugas de memoria o fallos de la aplicación.

Cuándo Usar AsyncTask

- Tareas cortas o interrumpibles
- Tareas que no necesitan informar de nuevo a la UI o al usuario
- Tareas de baja prioridad que pueden quedar sin terminar

Loaders

¿Qué es un Loader?

Un Loader proporciona carga asíncrona de datos y puede volver a conectarse a la actividad después de un cambio de configuración. También puede monitorear cambios en la fuente de datos y entregar nuevos datos.

Ventajas de Usar Loaders

- Ejecutan tareas fuera del hilo de UI
- El LoaderManager maneja los cambios de configuración
- Son implementados eficientemente por el marco de trabajo
- Los usuarios no tienen que esperar a que se carguen los datos

Anatomía de un Loader

Un LoaderManager gestiona las funciones del Loader a través de callbacks y puede gestionar múltiples loaders.

Obtener un Loader con initLoader()

Para crear e iniciar un loader o reutilizar uno existente, usa initLoader():

```
getLoaderManager().initLoader(0, null, this);
getSupportLoaderManager().initLoader(0, null, this);
```

AsyncTaskLoader

¿Qué es AsyncTaskLoader?

AsyncTaskLoader es una subclase de Loader que utiliza un AsyncTask para realizar la carga en segundo plano.

Pasos para Subclasificar AsyncTaskLoader

1. **Subclase AsyncTaskLoader**:

```
public static class StringListLoader extends AsyncTaskLoader<List<String>> {
   public StringListLoader(Context context, String queryString) {
      super(context);
      mQueryString = queryString;
   }
}
```

2. **Implementar loadInBackground()**:

```
public List<String> loadInBackground() {
   List<String> data = new ArrayList<String>();
   // Cargar los datos de la red o de una base de datos
   return data;
}
```

3. **Implementar onStartLoading()**:

```
protected void onStartLoading() {
   forceLoad();
}
```

Implementación de Callbacks del Loader en la Actividad

1. **onCreateLoader()**:

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
   return new StringListLoader(this, args.getString("queryString"));
```

```
}
```

2. **onLoadFinished()**:

```
public void onLoadFinished(Loader<List<String>> loader, List<String> data) {
   mAdapter.setData(data);
}
```

3. **onLoaderReset()**:

```
@Override
public void onLoaderReset(final Loader<List<String>> loader) {
   // Dejar en blanco la mayor parte del tiempo
}
```

Obtener un Loader con initLoader()

Para iniciar un loader en una actividad, usa la biblioteca de soporte para ser compatible con más dispositivos:

```
getSupportLoaderManager().initLoader(0, null, this);
```

3.2. Broadcasts en Android

Broadcasts

¿Qué es un Broadcast?

Los broadcasts son mensajes enviados por el sistema Android y otras aplicaciones cuando ocurre un evento de interés. Estos mensajes están encapsulados en un objeto Intent que contiene los detalles del evento, como android.intent.action.HEADSET_PLUG enviado cuando se conecta o desconecta un auricular con cable.

Tipos de Broadcasts

Broadcasts del Sistema

Los broadcasts del sistema son mensajes enviados por el sistema Android cuando ocurre un evento del sistema que puede afectar tu aplicación. Ejemplos comunes incluyen:

- ACTION_BOOT_COMPLETED: Enviado cuando el dispositivo se inicia.

- ACTION_POWER_CONNECTED: Enviado cuando el dispositivo se conecta a una fuente de alimentación externa.

Broadcasts Personalizados

Los broadcasts personalizados son mensajes que tu aplicación envía de manera similar a los broadcasts del sistema. Por ejemplo, cuando tu aplicación quiere notificar a otras aplicaciones que se han descargado datos y están disponibles para su uso.

Enviar Broadcasts Personalizados

Métodos para Enviar Broadcasts

Android proporciona tres formas de enviar un broadcast:

1. **Ordered Broadcast**: Entregado a un receptor a la vez usando el método sendOrderedBroadcast().
2. **Normal Broadcast**: Entregado a todos los receptores registrados al mismo tiempo en un orden indefinido usando el método sendBroadcast().
3. **Local Broadcast**: Enviado a receptores dentro de tu aplicación sin problemas de seguridad usando LocalBroadcastManager.

Ejemplo de Envío de Broadcast Personalizado

Para enviar un broadcast personalizado, define una acción única para el intent en la actividad y el receptor de broadcast:

```
private static final String ACTION_CUSTOM_BROADCAST = "com.example.android.powerreceiver.ACTION_CUSTOM_BROADCAST";
```

Broadcast Receivers

¿Qué es un Broadcast Receiver?

Un broadcast receiver es un componente de la aplicación que se registra para varios broadcasts del sistema o personalizados. Son notificados a través de un Intent:

- Por el sistema cuando ocurre un evento del sistema para el que la aplicación está registrada.
- Por otra aplicación, incluida la propia, si la aplicación está registrada para ese evento personalizado.

Implementación de Broadcast Receivers

Registro de Broadcast Receivers

Los broadcast receivers pueden registrarse de dos maneras:

1. **Receptores Estáticos**: Registrados en tu archivo AndroidManifest.xml, también llamados receptores declarados en el manifiesto.
2. **Receptores Dinámicos**: Registrados usando el contexto de la aplicación o de actividades en tus archivos Java, también llamados receptores registrados en el contexto.

Restricciones para Receptores Estáticos

A partir de Android 8.0 (API nivel 26), los receptores estáticos no pueden recibir la mayoría de los broadcasts del sistema. Usa un receptor dinámico para registrarte en estos broadcasts. Si te registras para los broadcasts del sistema en el manifiesto, el sistema Android no los entregará a tu aplicación. Algunas excepciones se aplican a esta restricción.

Crear un Broadcast Receiver

1. **Subclasificar la clase BroadcastReceiver y sobrescribir su método onReceive()**:

```
public class CustomReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    // Este método se llama cuando el BroadcastReceiver recibe un Intent de broadcast.
  }
}
```

2. **Registrar el broadcast receiver y especificar los filtros de intent**:
   - **Estáticamente en el manifiesto**:

```
<receiver android:name=".CustomReceiver" android:enabled="true"
android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

   - **Dinámicamente con registerReceiver()**:

```
IntentFilter filter = new IntentFilter(Intent.ACTION_POWER_CONNECTED);
CustomReceiver mReceiver = new CustomReceiver();
registerReceiver(mReceiver, filter);
```

Implementación del Método onReceive()

Ejemplo de implementación del método onReceive() para manejar eventos de conexión y desconexión de energía:

```
@Override
public void onReceive(Context context, Intent intent) {
   String intentAction = intent.getAction();
   switch (intentAction) {
      case Intent.ACTION_POWER_CONNECTED:
         // Manejar evento de conexión de energía
         break;
      case Intent.ACTION_POWER_DISCONNECTED:
         // Manejar evento de desconexión de energía
         break;
   }
}
```

Restricción de Broadcasts

Importancia de Restringir los Broadcasts

Restringir tus broadcasts es altamente recomendable. Un broadcast no restringido puede representar una amenaza de seguridad. Por ejemplo, si el broadcast de tu aplicación no está restringido e incluye información sensible, una aplicación que contenga malware podría registrarse y recibir tus datos.

Formas de Restringir un Broadcast

- **Usar LocalBroadcastManager**: Mantiene los datos dentro de tu aplicación evitando fugas de seguridad.
- **Usar el método setPackage()**: Pasa el nombre del paquete. Tu broadcast se restringe a las aplicaciones que coincidan con el nombre del paquete especificado.
- **Enforzar permisos por el emisor o receptor**.

Enforzar Permisos por el Emisor

Para enforzar un permiso al enviar un broadcast, suministra un argumento de permiso no nulo a sendBroadcast(). Solo los receptores que soliciten este permiso usando la etiqueta <uses-permission> en su archivo AndroidManifest.xml pueden recibir el broadcast.

Enforzar Permisos por el Receptor

Para enforzar un permiso al recibir un broadcast:

- **Registro Dinámico**: Suministra un permiso no nulo a registerReceiver().
- **Registro Estático**: Usa el atributo android:permission dentro de la etiqueta <receiver> en tu archivo AndroidManifest.xml.

4. Alarmas y programadores

4.1. Notificaciones en Android

¿Qué son las Notificaciones?

Definición

Una notificación es un mensaje que se muestra al usuario fuera de la interfaz de usuario normal de la aplicación. Una notificación típica incluye:

- **Icono pequeño**
- **Título**
- **Texto de detalle**

Uso de las Notificaciones

Android emite una notificación que aparece como un icono en la barra de estado. Para ver los detalles, el usuario abre el cajón de notificaciones. Las notificaciones nuevas se muestran como una "insignia" (también conocida como "punto de notificación") en el icono de la aplicación.

Insignia del Icono de la Aplicación

Disponible solo en dispositivos con Android 8.0 (API nivel 26) y superior. Los usuarios pueden hacer una pulsación larga en el icono de la aplicación para ver las notificaciones de esa aplicación, similar al cajón de notificaciones.

Canales de Notificación

¿Qué son los Canales de Notificación?

Los canales de notificación permiten crear un canal personalizable por el usuario para cada tipo de notificación que se mostrará. Más de una notificación puede agruparse en un canal.

Configuración de Comportamiento

Puedes establecer comportamientos como sonido, luz, vibración, y más para todas las notificaciones en ese canal.

Importancia de los Canales de Notificación

Introducidos en Android 8.0 (API nivel 26). Todas las notificaciones deben asignarse a un canal en dispositivos con API nivel 26 o superior, de lo contrario, no se mostrarán.

Apariencia en Configuraciones

Los canales de notificación aparecen como categorías bajo las notificaciones de la aplicación en la configuración del dispositivo.

Creación de un Canal de Notificación

Pasos para Crear un Canal de Notificación

1. **Instancia del Canal de Notificación**:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
   NotificationChannel notificationChannel =
      new NotificationChannel(CHANNEL_ID, "Mascot Notification",
      NotificationManager.IMPORTANCE_DEFAULT);
}
```

2. **Especificar Detalles del Canal**:
   - ID único dentro del paquete.
   - Nombre visible para el usuario.
   - Nivel de importancia.

Nivel de Importancia

Disponible en Android 8.0 (API nivel 26) y superior. Establece el nivel de intrusión como el sonido y la visibilidad para todas las notificaciones publicadas en el canal. Rango de IMPORTANCE_NONE (0) a IMPORTANCE_HIGH (4).

Prioridad de Notificación

Determina cómo el sistema muestra la notificación con respecto a otras notificaciones en versiones anteriores a API nivel 26. Establecido usando el método setPriority() para cada notificación. Rango de PRIORITY_MIN a PRIORITY_MAX.

Creación de Notificaciones

Uso de NotificationCompat.Builder

Para crear una notificación, usa la clase NotificationCompat.Builder. Pasa el contexto de la aplicación y el ID del canal de notificación al constructor.

```
NotificationCompat.Builder mBuilder =
   new NotificationCompat.Builder(this, CHANNEL_ID);
```

Configuración del Contenido de la Notificación

1. **Icono pequeño**: Establecido por setSmallIcon(). Este es el único contenido requerido.
2. **Título**: Establecido por setContentTitle().
3. **Texto del cuerpo**: Establecido por setContentText().

```
NotificationCompat.Builder mBuilder =
   new NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.android_icon)
        .setContentTitle("You've been notified!")
        .setContentText("This is your notification text.");
```

Acciones y Botones de Acción

Acción de Tocar la Notificación

Cada notificación debe responder cuando se toca, usualmente lanzando una Actividad en tu aplicación. Configura una intención de contenido usando el método setContentIntent() y pasa la intención envuelta en un objeto PendingIntent.

Botones de Acción de la Notificación

Los botones de acción pueden realizar una variedad de acciones en nombre de tu aplicación, como iniciar una tarea en segundo plano o hacer una llamada telefónica. Desde Android 7.0 (API nivel 24), se puede responder a mensajes directamente desde las notificaciones.

Para añadir un botón de acción, pasa un PendingIntent al método addAction().

Notificaciones con Vista Expandida

Notificaciones Expandibles

Las notificaciones en el cajón de notificaciones aparecen en dos diseños principales: vista normal (predeterminada) y vista expandida. Las notificaciones con vista expandida se introdujeron en Android 4.1. Úsalas con moderación, ya que ocupan más espacio y atención.

Notificaciones de Texto Grande

Para notificaciones de formato grande que incluyen mucho texto. Usa la clase auxiliar NotificationCompat.BigTextStyle.

Notificaciones de Imagen Grande

Para notificaciones de formato grande que incluyen una imagen grande adjunta. Usa la clase auxiliar NotificationCompat.BigPictureStyle.

Notificaciones de Medios

Para notificaciones de reproducción de medios, acciones para controlar medios como música y una imagen para la portada del álbum. Usa la clase auxiliar NotificationCompat.MediaStyle.

Entrega de Notificaciones

Construcción de una Notificación

Usa la clase NotificationManager para entregar notificaciones. Crea una instancia de NotificationManager y llama a notify() para entregar la notificación.

```
mNotifyManager = (NotificationManager)
   getSystemService(NOTIFICATION_SERVICE);

mNotifyManager.notify(NOTIFICATION_ID, myNotification);
```

Gestión de Notificaciones

Actualización de Notificaciones

Actualiza una notificación cambiando y/o añadiendo parte de su contenido. Emite la notificación con parámetros actualizados usando el constructor. Llama a notify() pasando el mismo ID de notificación. Si la notificación anterior sigue visible, el sistema la actualiza. Si la notificación anterior ha sido descartada, se entrega una nueva notificación.

Cancelación de Notificaciones

Las notificaciones permanecen visibles hasta que:

- El usuario las descarta deslizando o usando "Borrar todo".
- Llamar a setAutoCancel() al crear la notificación la elimina de la barra de estado cuando el usuario hace clic en ella.

- La aplicación llama a cancel() o cancelAll() en NotificationManager.

mNotifyManager.cancel(NOTIFICATION_ID);
Directrices de Diseño

- Si tu aplicación envía demasiadas notificaciones, los usuarios desactivarán las notificaciones o desinstalarán la aplicación.
- **Relevante**: Si esta información es esencial para el usuario.
- **Oportuna**: Las notificaciones deben aparecer cuando sean útiles.
- **Corta**: Usa la menor cantidad de palabras posible.
- Da a los usuarios el poder de elegir: Usa canales de notificación apropiados para categorizar tus notificaciones.

4.2. Alarmas y Programadores en Android

¿Qué son las Alarmas?

Definición

En Android, una alarma no es un reloj despertador real. Las alarmas se utilizan para programar algo que debe suceder a una hora establecida. Permiten disparar intents en momentos o intervalos específicos y pueden ser de una sola vez o recurrentes. Pueden basarse en el reloj en tiempo real o en el tiempo transcurrido desde el arranque del sistema. La aplicación no necesita estar ejecutándose para que la alarma esté activa.

Cómo Funcionan las Alarmas con los Componentes

- **BroadcastReceiver**: Se activa y entrega una notificación.
- **Activity**: Crea una notificación y establece una alarma.
- **Alarm triggers**: La alarma se dispara y envía un intent. La aplicación puede estar destruida, por lo que el BroadcastReceiver despierta la aplicación y entrega la notificación.

Beneficios de las Alarmas

- La aplicación no necesita estar en ejecución para que la alarma esté activa.
- El dispositivo no necesita estar despierto.
- No usa recursos hasta que se dispara.

- Se usa junto con BroadcastReceiver para iniciar servicios y otras operaciones.

Medición del Tiempo

- **Tiempo Transcurrido (Elapsed Real Time)**: Tiempo desde el arranque del sistema. Es independiente de la zona horaria y la configuración regional. Utilizado para intervalos y tiempo relativo. Incluye el tiempo en que el dispositivo estuvo en reposo.
- **Reloj en Tiempo Real (RTC)**: Tiempo UTC. Utilizado cuando importa la hora del día en la configuración regional.

Comportamiento de Despertar

- **Despierta el CPU del dispositivo si la pantalla está apagada**: Usado solo para operaciones críticas en el tiempo. Puede agotar la batería.
- **No despierta el dispositivo**: Se dispara la próxima vez que el dispositivo esté despierto. Es más educado con la batería.

Tipos de Alarmas

- **Tiempo Transcurrido (ERT)**:
  - No despierta el dispositivo: ELAPSED_REALTIME
  - Despierta el dispositivo: ELAPSED_REALTIME_WAKEUP
- **Reloj en Tiempo Real (RTC)**:
  - No despierta el dispositivo: RTC
  - Despierta el dispositivo: RTC_WAKEUP

Buenas Prácticas para Alarmas

- Añadir aleatoriedad a las solicitudes de red en alarmas.
- Minimizar la frecuencia de las alarmas.
- Usar ELAPSED_REALTIME en lugar del tiempo de reloj si es posible.

Batería

- Minimizar el despertar del dispositivo.
- Usar alarmas inexactas. Android sincroniza múltiples alarmas repetitivas inexactas y las dispara al mismo tiempo, reduciendo el consumo de batería.
- Usar setInexactRepeating() en lugar de setRepeating().

Cuándo No Usar una Alarma

- **Ticks** y **timeouts** mientras la aplicación está en ejecución: Usar Handler.
- **Sincronización de servidor**: Usar SyncAdapter con el servicio de mensajería en la nube.
- **Tiempo inexacto y eficiencia de recursos**: Usar JobScheduler.

Alarm Manager

¿Qué es AlarmManager?

AlarmManager proporciona acceso a los servicios de alarma del sistema. Programa operaciones futuras y cuando la alarma se dispara, el Intent registrado se emite. Las alarmas se retienen mientras el dispositivo está en reposo y pueden despertar el dispositivo cuando se disparan.

Obtener un AlarmManager

AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);

Programación de Alarmas

Elementos Necesarios para Programar una Alarma

- Tipo de alarma.
- Hora para activarse.
- Intervalo para alarmas repetitivas.
- PendingIntent para entregar en el momento especificado.

Programar una Alarma Única

- **set()**: Alarma inexacta única.
- **setWindow()**: Alarma inexacta única en una ventana de tiempo.
- **setExact()**: Alarma exacta única.
- Opciones de ahorro de energía en API 23+.

Programar una Alarma Repetitiva

- **setInexactRepeating()**: Alarma repetitiva inexacta.
- **setRepeating()**: Antes de API 19, crea una alarma repetitiva exacta. Después de API 19, es lo mismo que setInexactRepeating().

Ejemplo de setInexactRepeating()

```
alarmManager.setInexactRepeating(
   AlarmManager.ELAPSED_REALTIME_WAKEUP,
   SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_FIFTEEN_MINUTES,
   AlarmManager.INTERVAL_FIFTEEN_MINUTES,
   notifyPendingIntent);
```

Consideraciones Adicionales sobre Alarmas

Comprobación de una Alarma Existente

boolean alarmExists = (PendingIntent.getBroadcast(this, 0, notifyIntent, PendingIntent.FLAG_NO_CREATE) != null);

Doze y Standby

- **Doze**: Dispositivo completamente inmóvil, desenchufado y en reposo.
- **Standby**: Dispositivo desenchufado en aplicaciones en reposo. Las alarmas no se dispararán en estos modos. Aplicable en API 23+.

Alarmas Visibles para el Usuario

- **setAlarmClock()**: La UI del sistema puede mostrar la hora/icono. Preciso y funciona cuando el dispositivo está en reposo. La aplicación puede recuperar la próxima alarma con getNextAlarmClock(). Disponible en API 21+.

Cancelar una Alarma

Llama a cancel() en el AlarmManager y pasa el PendingIntent.

alarmManager.cancel(alarmPendingIntent);

Alarmas y Reinicios

Las alarmas se borran cuando el dispositivo está apagado o se reinicia. Usa un BroadcastReceiver registrado para el evento BOOT_COMPLETED y establece la alarma en el método onReceive().

4.3. Transferencia Eficiente de Datos y JobScheduler en Android

Transferencia de Datos Eficiente

Uso de Recursos al Transferir Datos

- **Radio inalámbrico**: Utiliza la batería del dispositivo.

- **Planes de datos**: Consumir datos puede costar dinero a los usuarios, especialmente en aplicaciones gratuitas.

Estados de Energía del Radio Inalámbrico

- **Potencia completa**: Conexión activa con la tasa de transferencia de datos más alta.
- **Baja potencia**: Estado intermedio que usa un 50% menos de energía.
- **En espera**: Energía mínima sin conexión activa.

Transiciones del Estado del Radio Inalámbrico en 3G

El radio inalámbrico en un dispositivo 3G consume energía durante casi 20 segundos en cada sesión de transferencia de datos.

Agrupar Transferencias de Datos

- **Transferencias no agrupadas**: Enviar datos durante 1 segundo cada 18 segundos mantiene el radio principalmente en potencia completa.
- **Transferencias agrupadas**: Enviar datos en paquetes de 3 segundos mantiene el radio principalmente en estado inactivo.

Descargar Datos de Antemano

Descargar todos los datos que probablemente necesites durante un período determinado en una sola ráfaga sobre una sola conexión a plena capacidad puede reducir el costo de la batería y la latencia.

Monitorear el Estado de Conectividad

- **Wi-Fi**: Usa menos batería y tiene más ancho de banda que el radio inalámbrico.
- Usa ConnectivityManager para determinar qué radio está activo y adaptar tu estrategia de transferencia de datos.

Monitorear el Estado de la Batería

- Espera condiciones específicas para iniciar operaciones intensivas en batería.
- BatteryManager transmite todos los detalles de la batería y la carga en un Broadcast Intent.
- Usa un BroadcastReceiver registrado para las acciones del estado de la batería.

JobScheduler

¿Qué es JobScheduler?

JobScheduler se utiliza para la programación inteligente de tareas en segundo plano basadas en condiciones en lugar de un horario fijo. Es más eficiente que AlarmManager, ya que agrupa tareas para minimizar el consumo de batería. Está disponible a partir de la API 21+ y no está en la biblioteca de soporte.

Componentes de JobScheduler

- **JobService**: Clase de servicio donde se inicia la tarea.
- **JobInfo**: Patrón de construcción para establecer las condiciones de la tarea.
- **JobScheduler**: Programa y cancela tareas, lanza el servicio.

JobService

Implementación de JobService

- **Subclase de JobService**: Implementa tu tarea aquí.
- **Sobrescribir onStartJob() y onStopJob()**: Ejecuta el trabajo y maneja la parada de la tarea.

onStartJob()

- **Implementar el trabajo**: Es el método donde se realiza el trabajo.
- **Condiciones cumplidas**: El sistema llama a este método cuando se cumplen las condiciones.
- **Desplazar el trabajo pesado a otro hilo**: Corre en el hilo principal, por lo que las tareas pesadas deben ser manejadas en otro hilo.

Finalización de JobService

- **onStartJob() retorna un booleano**:
  - **FALSE**: El trabajo ha terminado.
  - **TRUE**: El trabajo ha sido desplazado. Llama a jobFinished() desde el hilo de trabajo y pasa el objeto JobParams de onStartJob().

onStopJob()

- **Llamado por el sistema**: Si determina que la ejecución del trabajo debe detenerse.
- **Ejemplo**: Las condiciones especificadas ya no se cumplen (como la pérdida de conectividad Wi-Fi).
- **Return TRUE**: Para reprogramar el trabajo.

Código Básico de JobService

```
public class MyJobService extends JobService {
   private UpdateAppsAsyncTask updateTask = new UpdateAppsAsyncTask();

   @Override
   public boolean onStartJob(JobParameters params) {
      updateTask.execute(params);
      return true; // el trabajo ha sido desplazado
   }

   @Override
   public boolean onStopJob(JobParameters jobParameters) {
      return true;
   }
}
```

Registrar JobService

```
<service
  android:name=".NotificationJobService"
  android:permission="android.permission.BIND_JOB_SERVICE"/>
```

JobInfo

Construcción de JobInfo

Establece las condiciones de ejecución usando JobInfo.Builder.

Objeto Builder de JobInfo

- **Arg 1**: ID del trabajo.
- **Arg 2**: Componente del servicio.
- **Arg 3**: JobService a lanzar.

```
JobInfo.Builder builder = new JobInfo.Builder(
   JOB_ID,
   new ComponentName(getPackageName(), NotificationJobService.class.getName()));
```

Configuración de Condiciones

- **setRequiredNetworkType(int networkType)**
- **setBackoffCriteria(long initialBackoffMillis, int backoffPolicy)**
- **setMinimumLatency(long minLatencyMillis)**
- **setOverrideDeadline(long maxExecutionDelayMillis)**
- **setPeriodic(long intervalMillis)**
- **setPersisted(boolean isPersisted)**
- **setRequiresCharging(boolean requiresCharging)**

- **setRequiresDeviceIdle(boolean requiresDeviceIdle)**

Ejemplo de Configuración

```
JobInfo.Builder builder = new JobInfo.Builder(
      JOB_ID,
      new ComponentName(getPackageName(), NotificationJobService.class.getName()))
   .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
   .setRequiresDeviceIdle(true)
   .setRequiresCharging(true);

JobInfo myJobInfo = builder.build();
```

JobScheduler

Programación del Trabajo

- **Obtener un objeto JobScheduler del sistema**.
- **Llamar a schedule() en JobScheduler con el objeto JobInfo**.

```
mScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
mScheduler.schedule(myJobInfo);
```

5. Conclusiones

En esta segunda unidad didáctica, hemos abordado el manejo de tareas en segundo plano dentro del contexto de la programación dirigida por eventos en aplicaciones Android. El principal reto para los desarrolladores de Android es asegurar que las aplicaciones sean responsivas y no se vean afectadas por tareas largas que puedan bloquear el hilo principal. Es por ello que en esta unidad se ha puesto especial énfasis en la utilización de herramientas como **AsyncTask**, **Loaders**, y el uso eficiente de **hilos (threads)** para delegar las tareas pesadas a un contexto de segundo plano.

Se ha destacado la importancia del **hilo principal**, responsable de la interfaz de usuario y la necesidad de mantenerlo libre de procesos intensivos para garantizar una experiencia fluida para el usuario. Además, el manejo de **broadcasts**, tanto del sistema como personalizados, permite una comunicación eficiente entre las aplicaciones y el sistema operativo, mejorando la interacción entre distintos componentes.

Otra herramienta esencial es el **AlarmManager**, que permite la programación de tareas en momentos específicos sin necesidad de que la aplicación esté activa, lo que maximiza el uso de los recursos del sistema sin afectar la experiencia del usuario. También hemos explorado **JobScheduler**, introducido en versiones más recientes de Android, que facilita la programación inteligente de tareas en segundo plano, ajustándose a las condiciones del sistema como el estado de la red y el nivel de batería.

En resumen, esta unidad proporciona las bases necesarias para gestionar tareas en segundo plano de manera eficiente, asegurando la estabilidad y el rendimiento óptimo de las aplicaciones Android. Estos conceptos son clave para el desarrollo de aplicaciones robustas y escalables en un entorno de programación móvil.

6. Bibliografía

Android Developers. (2023). Background Processing in Android. Google. Disponible en: https://developer.android.com/guide/components/processes-and-threads

Meier, R. (2015). Professional Android: Building Applications with Android Studio. John Wiley & Sons.

Phillips, B., Stewart, C., Hardy, K., & Marsicano, B. (2019). Android Programming: The Big Nerd Ranch Guide. Big Nerd Ranch.

Steele, J. (2014). The Android Developer's Cookbook: Building Applications with the Android SDK. Addison-Wesley.

Nagpal, S. (2018). Android Threading: Asynchronous Processing Patterns for Android Applications. Apress.

Estas fuentes ofrecen un soporte esencial en la comprensión de la gestión de hilos, tareas en segundo plano y el uso adecuado de herramientas como AsyncTask, AlarmManager, y JobScheduler para optimizar el rendimiento de aplicaciones Android.

UAX Universidad Alfonso X el Sabio

G R A C I A S

UAX.COM