Lecture 4

Examples of process management systems

Topics

- How does UNIX manage processes/threads?
- How different a sensor OS can be?
- The lifecycle of an Android application?

1. UNIX
A. Process model

- All processes are created by other ones, except the very first one started after the system is booted.
- By calling *fork()*, a new process is created, copy of the calling process.
- A call to *exec()* replaces the code of the existing program with a new one.
- A process can voluntarily terminate by calling *exit()*.
- Another possibility to terminate a process is to invoke *kill()* from another process (with appropriate privileges).
- When a parent process is ready to pause until a child process has finished and to pick up the child's exit status, it does so by calling *wait()*.

B. Process states

- Beside running, blocked and ready, there are few additional states:
- *SSLEEP* – a blocked state where the process cannot be awakened by a signal;
- *SWAIT* – a blocked state that allows the process to be awakened to handle a signal;
- *SRUN* – the SRUN value identifies running and ready processes; *the u variable* contains the process table info of the currently running process.
- *SIDL* – a process is created but the copy of its parent's memory space can not be done immediately.
- *SZOMB* – a child process that exits before the parent makes the call wait(),…the child process will still exist at some level.
- *SSTOP* – this state is used to identify a process (child) that is being traced (by its parent).

C. Process table

- There are few flags recording the status of the process in respect to memory (the process is in memory or swapped out) and tracing.

- The process table is divided in two.
  - the first part is an array of structures (*proc*). These structures hold admin info, state info, id, scheduling info.
  - other data is not needed when the process is swapped out. They are stored in the *user structure*, part of the data segment. User structures are swapped along with the rest of the process's memory space.

D. Scheduling

- The scheduler uses process priorities. The actual scheduling code is in the context switching function *swtch().* It searches the process table for the highest priority process in memory.
- Processes migrate between memory and disk under the control of the function *sched().*
- Swtch() and sched() represent a two-level scheduler.
- Periodically, the priority of each process is updated:

$$p = \min(127, \frac{c}{16} + 100 + n)$$

- Where c is a cumulative CPU usage since the process was last swapped into memory. *n* is a parameter called *nice*. If nice increases, the priority lowers.

2. TinyOS

- This is an OS for tiny sensors that provides a set of system SW components.
- Only the necessary parts of the OS are compiled with the application → each application is built into the OS.
- An application wires OS components together with application-specific components – *a complete system consists in a scheduler and a graph of components*.
- A component has four interrelated parts: a set of command handlers, a set of event handlers, a fixed-size frame and a bundle of tasks.
- Tasks, events and commands execute in the context of the frame and operate on its state.

A. The program model

- In TinyOS, tasks and events provide two sources of concurrency.
- A hardware event triggers a processing chain that can go upward and can bend downward by commands.
- To avoid cycles, commands cannot signal events.
- Tasks don’t preempt each other.
- The scheduler invokes a new task from the queue only when the current task has completed.
- When there is no task in the queue, the scheduler puts the Core into the *sleep* mode – not the peripherals.

Events

- Events are generated by HW (interrupts).
- The execution of an interrupt handler is called an event context.
- The processing of events also run to completion, but it preempts tasks and can be preempted by other events.
- If the task queue is empty, an event has as result a task being scheduled…
- Event handlers should be small !

B. Schedulers and tasks

- TinyOS 1.x provides a single kind of task, a parameter-free function, and a single scheduling policy, FIFO.
- The task queue in TinyOS 1.x is implemented as a fixed size circular buffer of function pointers. Posting a task puts the task's function pointer in the next free element of the buffer; if there are no free elements, the post returns fail.

- This model has several issues:
  - some components do not have a reasonable response to a failed post;
  - as a given task can be posted multiple times, it can consume more than one element in the buffer;
  - all tasks from all components share a single resource: one misbehaving component can cause other's posts to fail.

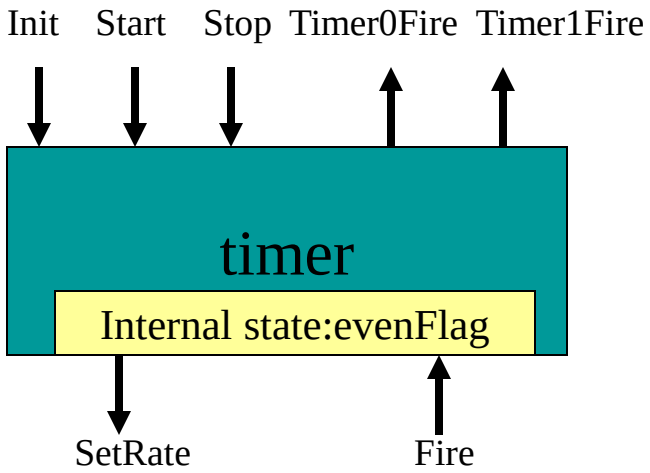
- In TinyOS 2.x, a basic post will only fail if and only if the task has already been posted and has not started execution. A task can always run, but can only have one outstanding post at any time.

- 2.x introduces task interfaces for additional task models. Task interfaces allow users to extend the syntax and semantics of tasks – priority, earliest deadline first, etc.

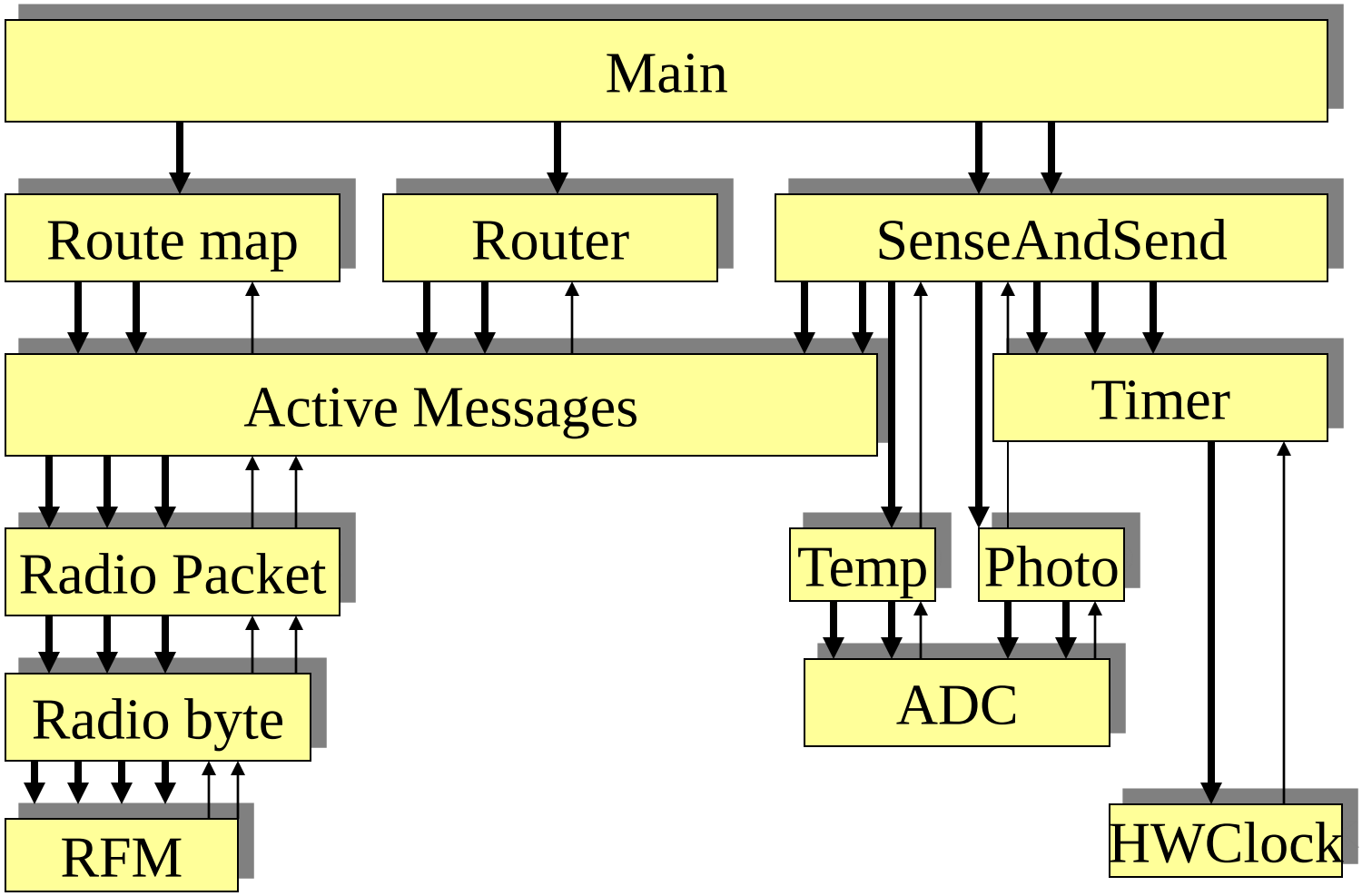
- One byte of state is allocated per task; the assumption is that there will be fewer than 255 tasks in the system.

The Timer TinyOS component

- It works with a lower layer HWClock component = SW wrapper around a HW clock that generates periodic interrupts.
- An arrow pointing into the component denotes a call from other components.
- An arrow pointing outside is a call from this component…



C. The FieldMonitor application



3. Android

- Android provides an operating system (Linux-based stack for managing devices, memory and processes), plus middleware and applications – it's a general-purpose system for mobile devices.
- The concept is that *"the handheld is the new PC"*.
- Android has its own JVM, called Dalvik VM.
- *One key architectural goal*: allow applications to interact with one another and reuse components from one another.
- The reuse applies not only to services but also to data and UI.

A. Dalvik VM

- DVM takes the Java class files and combines them into one or more Dalvik executable files (.dex). It reuses duplicate information from multiple class files, effectively reducing the space requirement from the traditional .jar file (by half or even more).

- DVM uses CPU registers as the primary memory instead of the stack; the expected result is 30% less machine instructions.

- Each thread of the application has its own resources (PC register, stack, etc) allocated by the system.

B. Android foundational components

- Android applications don't have a single entry point for everything in the application (no *main() function, for example). They have components that the system can instantiate and run as needed.*
- An *intent* is an amalgamation of ideas such as windowing messages, actions, inter-process communication, publish/subscribe models and application registries.
- *Example:*

```
public static void invokeWebBrowser(Activity activity)
{
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse(http://www.ucc.ie));
        activity.startActivity(intent);
}
```

- Android is asked to start a window to display the content of a web site.

- *Intents* define “intention” to do some work – broadcast a message, start a service, launch an activity, dial a phone number or answer a call. They can also be used by the system to notify the application of specific events (i.e. arrival of a text message).
- *Views* are UI elements that can be used to create a user interface.
- An *activity* is a UI concept. Usually, it represents a single screen in the application; it can contain one or more views.

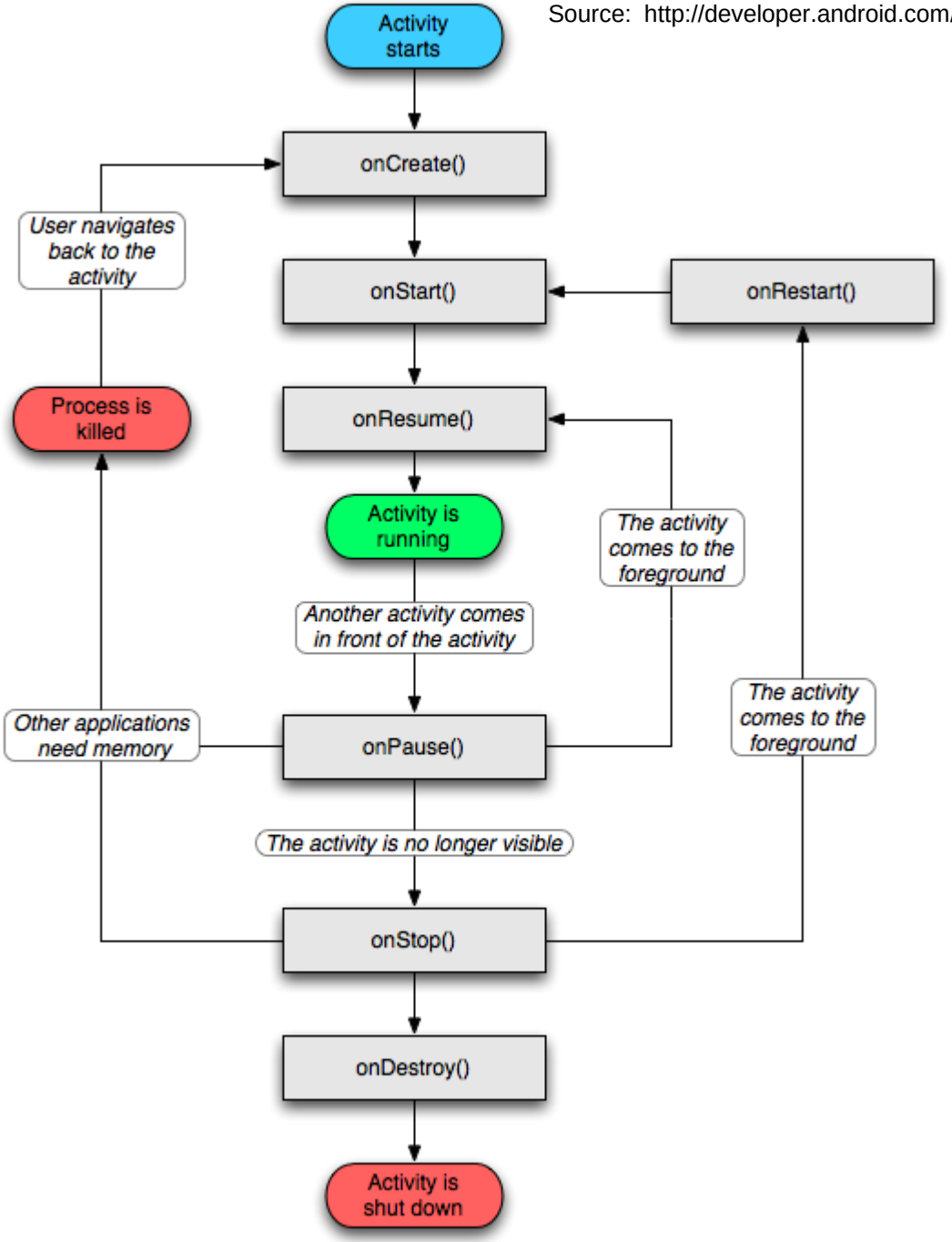
- *Content providers* allow to expose data to sharing by multiple applications.

- A *service* is a background process, *local* (accessed only by the application hosting it), or *remote* (accessed by other applications running on the device).

C. Android application lifecycle

- The Android application lifecycle is managed by the system based on the user needs, available resources, etc.
- The system decides if an application can be loaded or it is paused or stopped.
- The activity currently used gets higher priority while an activity not visible can be killed to free resources.
- Each Android application runs in a separate process which hosts its own virtual machine. This is a protected-memory environment. Then, its priority can be controlled by the system.

Source: http://developer.android.com/reference/android/app/Activity.html



D. Activities and tasks

- A *task* is *a stack of activities*. There is no way to set values for a task independently of its activities. Values for the task as a whole are set in the *root activity (launcher)*.
- One activity can start another one, including one defined in a different application. For example, the user wants to display a street map of some location. There's already an activity that can do that, so all your activity needs to do is put together an Intent object with the required information and pass it to startActivity(). The map viewer will display the map. When the user hits the BACK key, your activity will reappear on screen. To the user, it will seem as if the map viewer is part of the same application, even though it's defined in another application and runs in that application's process.

- Task activities are arranged in a stack: the root activity in the stack is the one that began the task — typically, it's an activity the user selected in the application launcher. The activity at the top of the stack is one that's currently running — the one that is the focus for user actions. When one activity starts another one, the new activity is pushed on the stack; it becomes the running activity. The previous activity remains in the stack. When the user presses the BACK key, the current activity is popped from the stack, and the previous one resumes as the running activity.
- The stack contains objects, so if a task has more than one instance of the same Activity subclass open — multiple map viewers, for example — the stack has a separate entry for each instance. Activities in the stack are never rearranged, only pushed and popped.

Conclusions

- Each OS is a set of services that abstracts the specific machine architecture.
- There are many different OS, depending on their range of use and the characteristics of the host devices.

- Suggested further study:
  - compare the three different OS in terms of how processes are defined and managed by the system;
  - check how Android java app threads are scheduled on a multi-core device (more advanced topic);
  - a foreground activity has the highest priority. How is the system managing priorities?