Lecture 3

Examples of process management systems

Few questions…

- How does UNIX manage processes/threads ?
- Is Win NT very different from UNIX ?
- How different can a sensor OS be ?
- What is the lifecycle of an Android application ?

1. UNIX

- For each program commanded to execute, a process is created. 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().*

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).

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.

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:

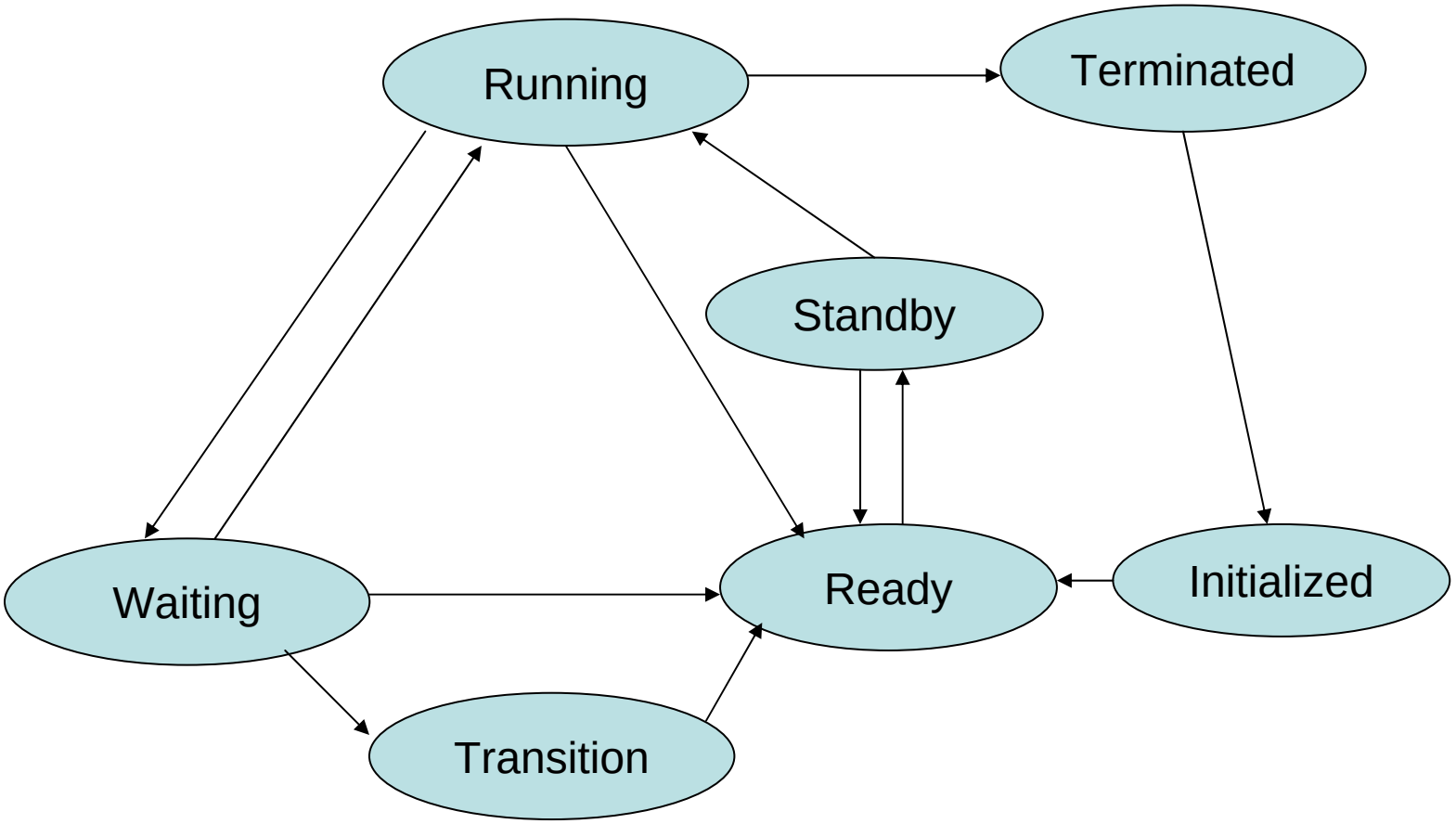
$$p = \min(127, \frac{c}{16} + 100 + n)$$

- 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. Win NT

- At the most coarse-grained level, we have jobs; a job is a set of processes that share certain process management parameters.
- Processes in Win NT are sets of one or more threads that all share a common memory space.
- Threads are the units of execution that are scheduled and managed.
- The CreateProcess() Win 32 call creates a new process and implicitly a new thread. The call takes as argument the name of the program to run in the newly created process.
- createThread() starts a thread from the mentioned function of the process.
- ExitProcess() and ExitThread() terminate processes and threads respectivily.

Thread state



Win NT scheduling

- It uses a multilevel feedback queue with 32 priority levels.
- The lowest, 0, is reserved for a special kernel thread that clears free memory pages.
- Levels 1 – 15 are dynamic levels: are allocated to application threads.
- Levels 16 – 31 are real-time levels. They are not real-time in the sense of guaranteed response time or in terms of scheduling by deadline. They provide a higher priority level than normal applications and a more predictable response because they are not dynamically adjusted by the system.
- Additionally, threads may have a different value of quantum (CPU time slice). It can vary from 20 ms to 120 ms for foreground threads (belong to processes that own the window).

- Generally, threads keep the same priority level during execution. However, in certain circumstances, the priority is increased, after which decays back in a stairstep fashion:
  - when the thread is moved into Ready after an I/O operation;
  - after waiting on an executive event or semaphore, the level in incremented by one;
  - a foreground thread that owns a window after unblocking;
  - threads that own windows when they move to Ready after a windowing event, have the priority incremented by two;
  - a starved thread (ready for 3-4 sec) receives priority 15 and a double quantum.

3. TinyOS

- This is an OS for tiny sensors.
- Only the necessary parts of the OS are compiled with the application → each application is built into the OS.
- It provides a set of system SW components.
- 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.

The program model

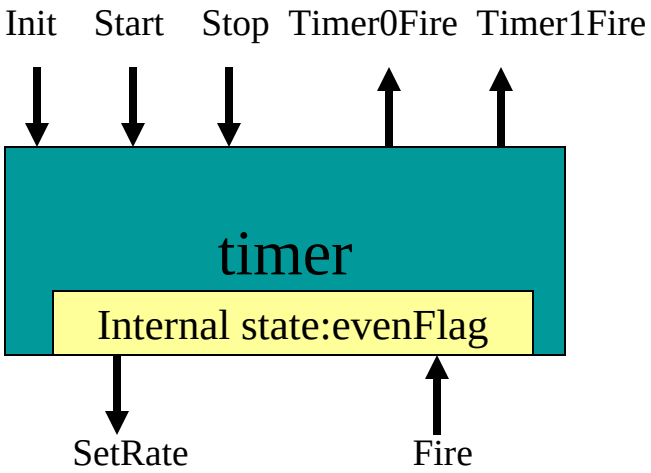
- In TinyOS, tasks and events provide two sources of concurrency.
- A hardware event triggers a processing chain that can go upward and can bend downward by commands.
- To avoid cycles, commands cannot signal events.
- Tasks don’t preempt each other.
- The scheduler invokes a new task from the queue 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

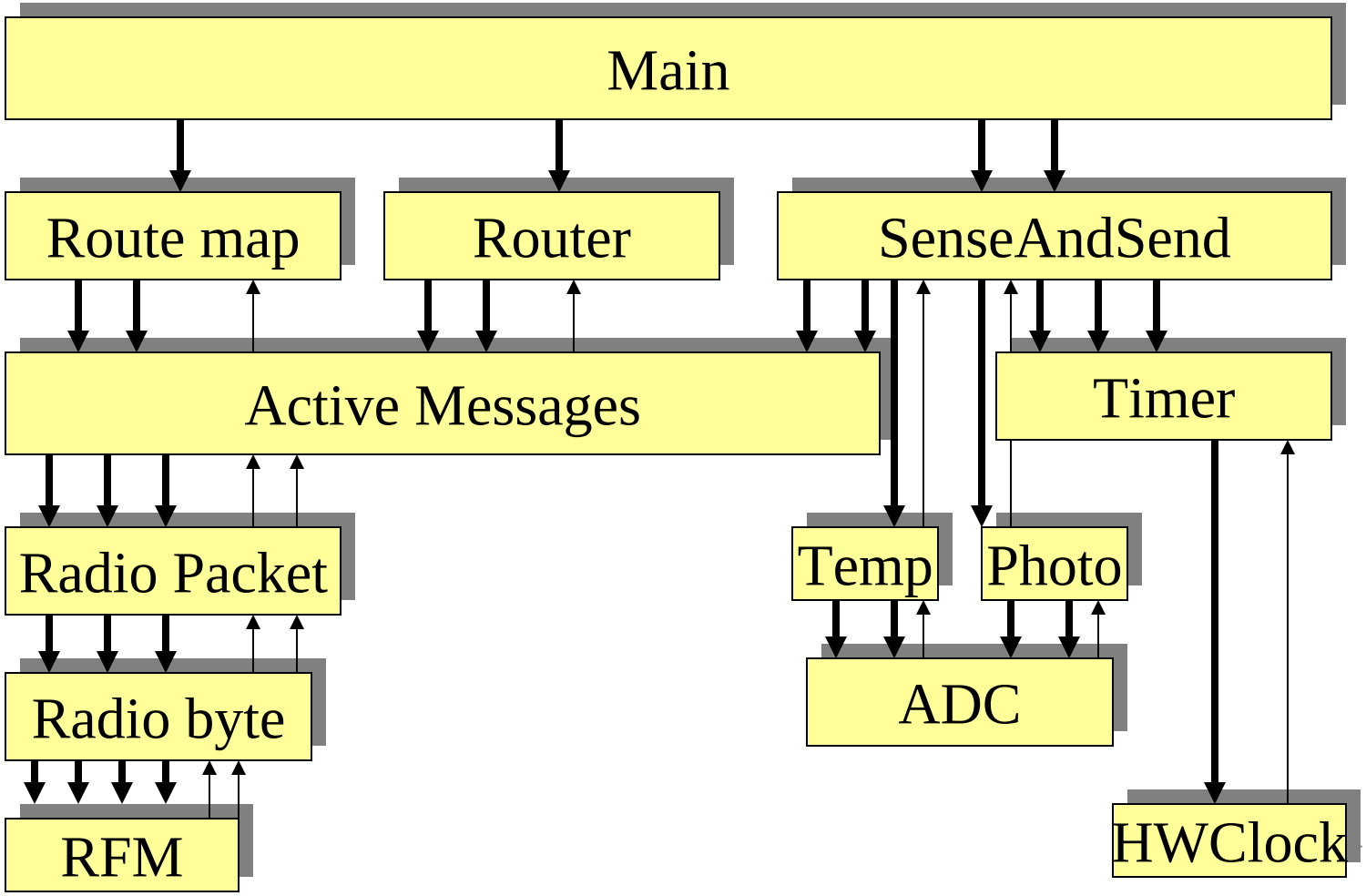
- Events are generated by HW (interrupts).
- The execution of an interrupt handler is called an event context.
- The processing of events also run to completion, but it preempts tasks and can be preempted by other events.
- If the task queue is empty, an event has as result a task being scheduled…
-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

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…



The FieldMonitor application



4. 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.
- The Android SDK supports most of Java SE, except for the Abstract Window Toolkit (AWT) and Swing. However, it offers an extensive UI framework.
- 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.

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).
- There is no JIT compiler because most of Android's libraries are implemented in C and C++. Java graphics APIs are actually thin wrapper classes around the native code using the Java Native Interface (JNI).
- DVM uses CPU registers as the primary memory instead of the stack; the expected result is 30% less machine instructions.

Android software stack

- Android core is the *Linux kernel v 2.6*; device drivers include Display, Camera, Keypad, WiFi, FlashMemory, Audio, IPC.
- A set of C/C++ libraries sit on top of the kernel: OpenGL, WebKit (browser support), FreeType (font support), SSL, the C runtime library (libc), SQLite (relational database available on the device) and Media.
- Most of the application framework accesses these core libraries through DVM. Each application will get its instance of DVM.
- The Java API's main libraries include resources, telephony, locations, UI, content providers (data) and package managers.
- On the very top are user applications such as Home, Contacts, Phone, Browser, etc.

Android foundational components

- 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);
}
```

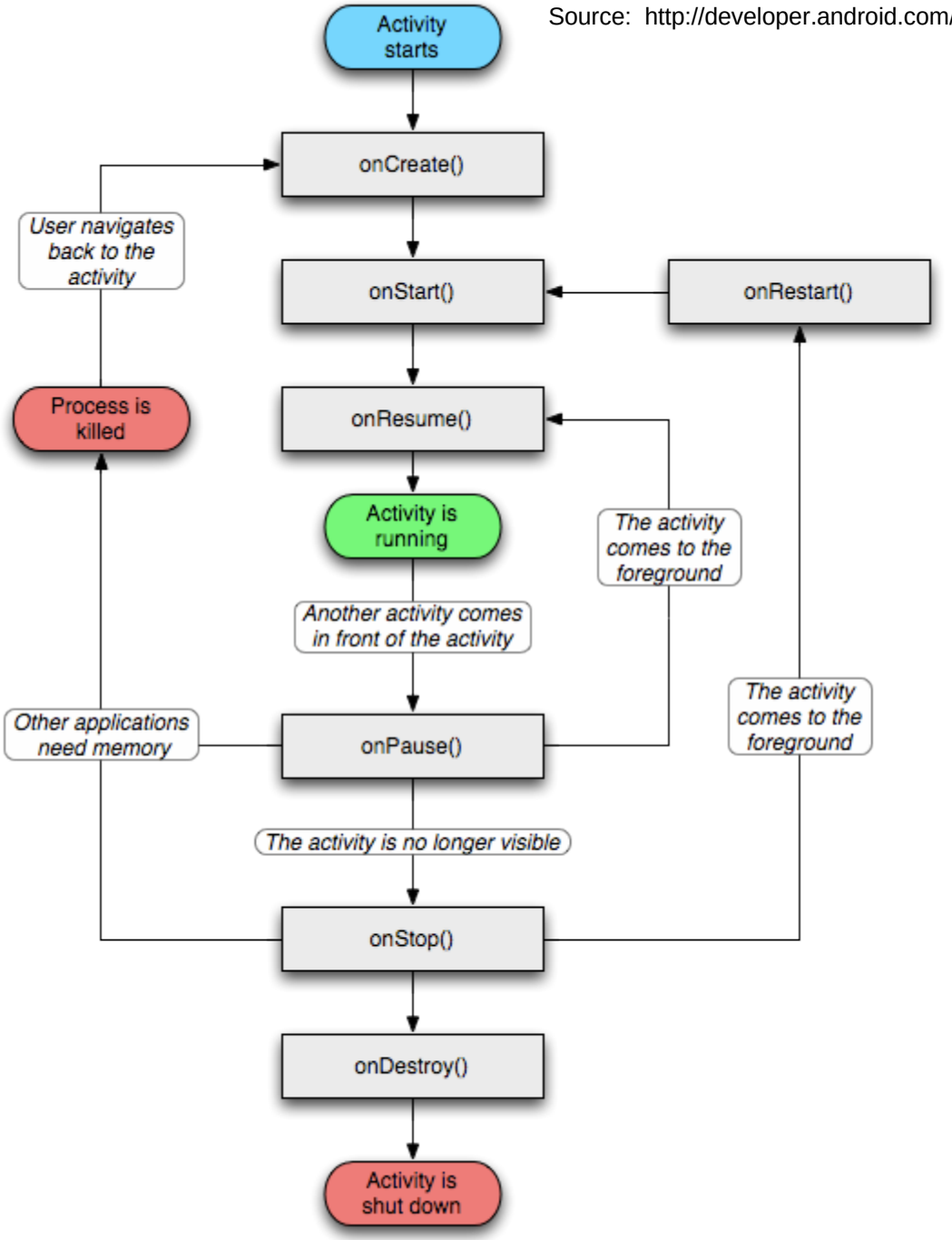
- Android is asked to start a window to display the content of a web site.

- *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).

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



Conclusions

- Compare the four different OS in terms of how processes/threads are defined and managed by the system.