Interoperation of tasks

Tamás Kovácsházy, PhD

4^{th} topic,

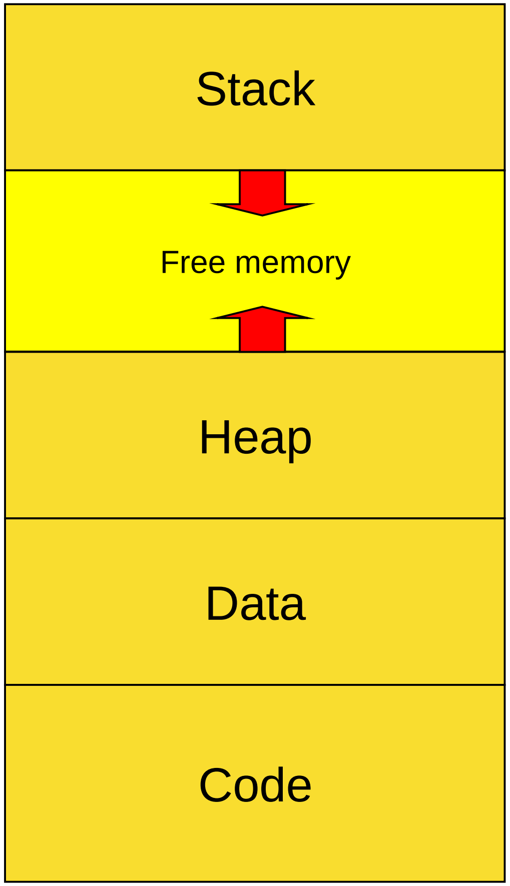
Implementation of tasks, processes and threads

Méréstechnika és
Információs Rendszerek
Tanszék

Budapest University of Technology and Economics
Department of Measurement and Information Systems

Implementation of the concept of task...

- From the point of view of implementation the concept of process is quite close to the concept of task
- The process is a program under execution
  - From the same program multiple processes can be created
  - It has its own code, data, heap, stack, and free memory
  - It is protected from other processes
    - Separation, virtual machine, sandbox



Separation of processes

- They run on their own virtual machine:
  - The virtual machine is created by the OS
  - They cannot have access to other processes and to the operating system (to the CPU states and memory areas)
  - There is a context switch if another process gets to run
- They have their own virtual memory (details will be given later).
  - Processes cannot have access to the virtual memory of other processes and to the physical memory directly
  - The MMU of the CPU provides this functionality
    - It is a possibility of sharing memory areas with “READ” privileges (e.g. shared library code memory).
    - Modern MMUs provide more detailed sharing capabilities (e.g. Write, No Execute, etc.)...

Creation of Processes

- OS specific system call (e.g. CreateProcess() in Windows, fork() in UNIX)
- Parent/child relationship between creator/created processes
  - Process tree
  - The child may have access to the resources of the parent in a configurable way (everything – nothing).
  - The parent may wait for the termination of the child (luckily life is different…)
  - The parent can pass parameters to the child (command line).
- UNIX fork() is going to be introduced later in detail
- Requires lot of administration and resources

Communication of processes

- Processes must interoperate (the actual solutions are going to be detailed later)
  - For that, they need to communicate
- Arbitrary two processes cannot communication through memory
  - The main task of the MMU and the virtual memory is to separate processes from this aspect
  - They can communicate only through system calls, which is resource hungry
- The process is efficient from the point of view of protection/separation
- The process is en inefficient way of solving parallel, strongly interrelated problems
  - E.g. GUI and some CPU intensive computation in the background (WORD „typesetting” after some edit)

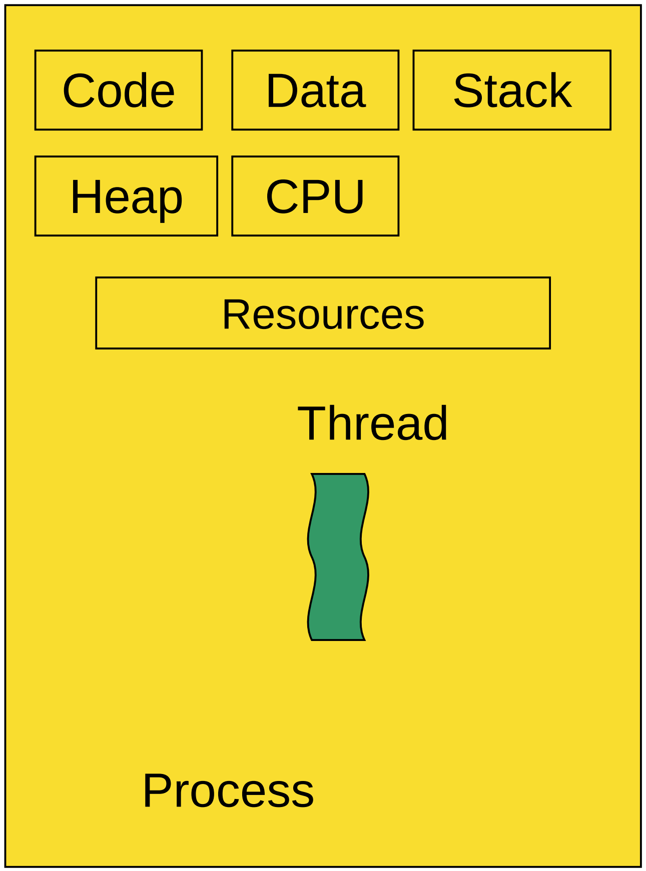
Termination of processes

- OS specific system call (e.g. TerminateProcess() on Windows, exit() on UNIX)
- The opened, previously used resources must be closed
  - E.g. opened files or TCP/IP sockets, etc.
- The parent may get a return value (most cases an integer), it informs it about the status of termination
- What if the parent terminates before the child?
  - OS specific implementation, typical solutions are:
    - The child is assigned a default parent (e.g. UNIX: init process).
    - Automatic termination of all childs (cascading termination).
- Requires lot of administration and resources

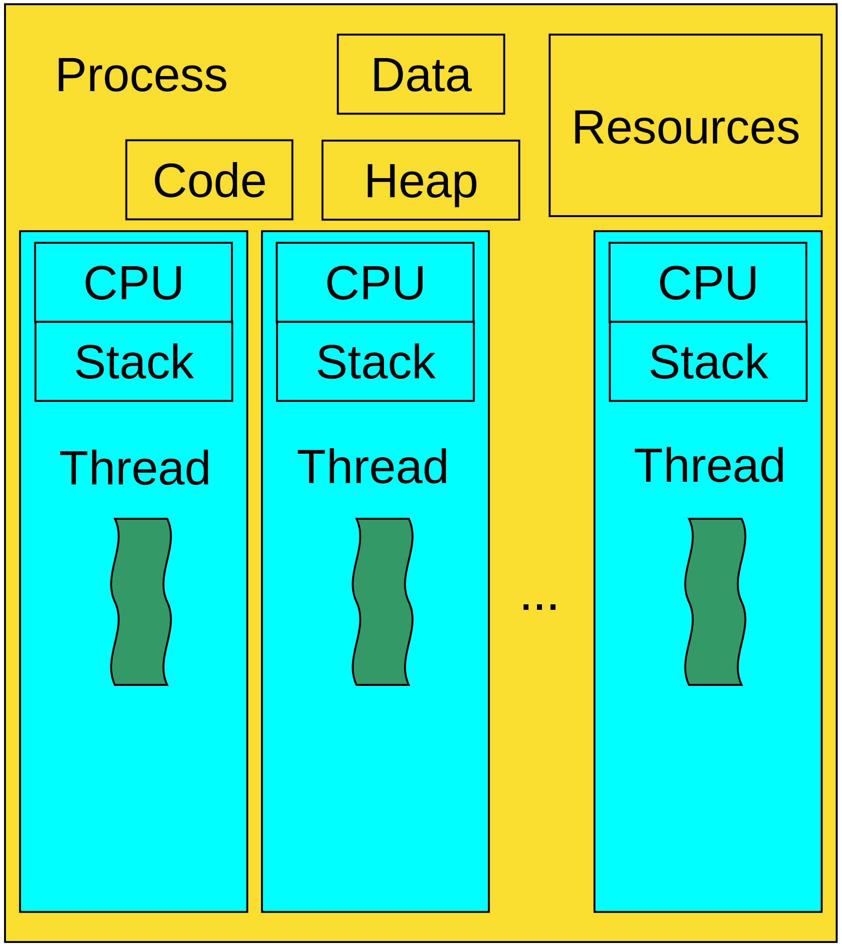
Evaluation of the concept of process

- From the point of view of protection and separation it is good solution, but it requires lot of resources
  - Creation and termination of processes
  - Communication and resource sharing between processes
- Solution: Introduction of the thread
  - The thread the default unit of CPU utilization, it is a sequential code
  - It has its own virtual CPU and stack
  - The code, data, heap and other resources are shared with other threads running in the context of a process
  - The process is a memory container, the thread is a CPU container
- Process = heavyweight process
- Thread = lightweight process

Processes and threads on a figure



Single thread OS supporting only processes, e.g. traditional UNIX



Thread based operating system, e.g. Windows NT and later, modern UNIX

Support of threads

- Modern operating systems support threads in a native way
- Windows:
  - Program or service = process and under the process multiple threads
  - The scheduler schedules threads
- Modern UNIX, Linux:
  - Program or daemon = process and under processes multiple threads
  - The scheduler schedules tasks, and a task can be a process (legacy programs from traditional UNIX) or a thread (new programs)

User space threads

- Under UNIX (even Linux in earlier times).
  - green threads
- The OS knows only processes
- Threads are needed, support is coming, programmers want to use it
  - User space thread libraries...
- The OS can only schedule processes, so if the process runs, its user space thread library can run its own thread level scheduler
  - Multiple threads form a scheduling unit!
    - Only one of those threads or the user space thread scheduler can run
    - Cannot utilize multiple execution units

Thread support (Creation)

- E.g. Win32 API, Pthreads, JAVA thread
- Win32: CreateThread() with complex parameters
- Pthreads: POSIX threads e.g. Linux and other UNIX variants, it supports kernel and user space threads also
- JAVA (VM is the process, inside the VM you can have threads):
  - If the class is inherited from the Thread class
  - If the Runnable interface is implemented
  - JAVA implements threads in a platform specific way
    - Nativ OS specific threads (one-to-one, today it is the typical solution).
    - JAVA specific threads mapping all JAVA threads to one native OS thread (many-to-one, if the OS does not support threads)
    - many-to-many mapping (may require less resources the one-to-one, but allows parallel execution not supported by many-to-one).

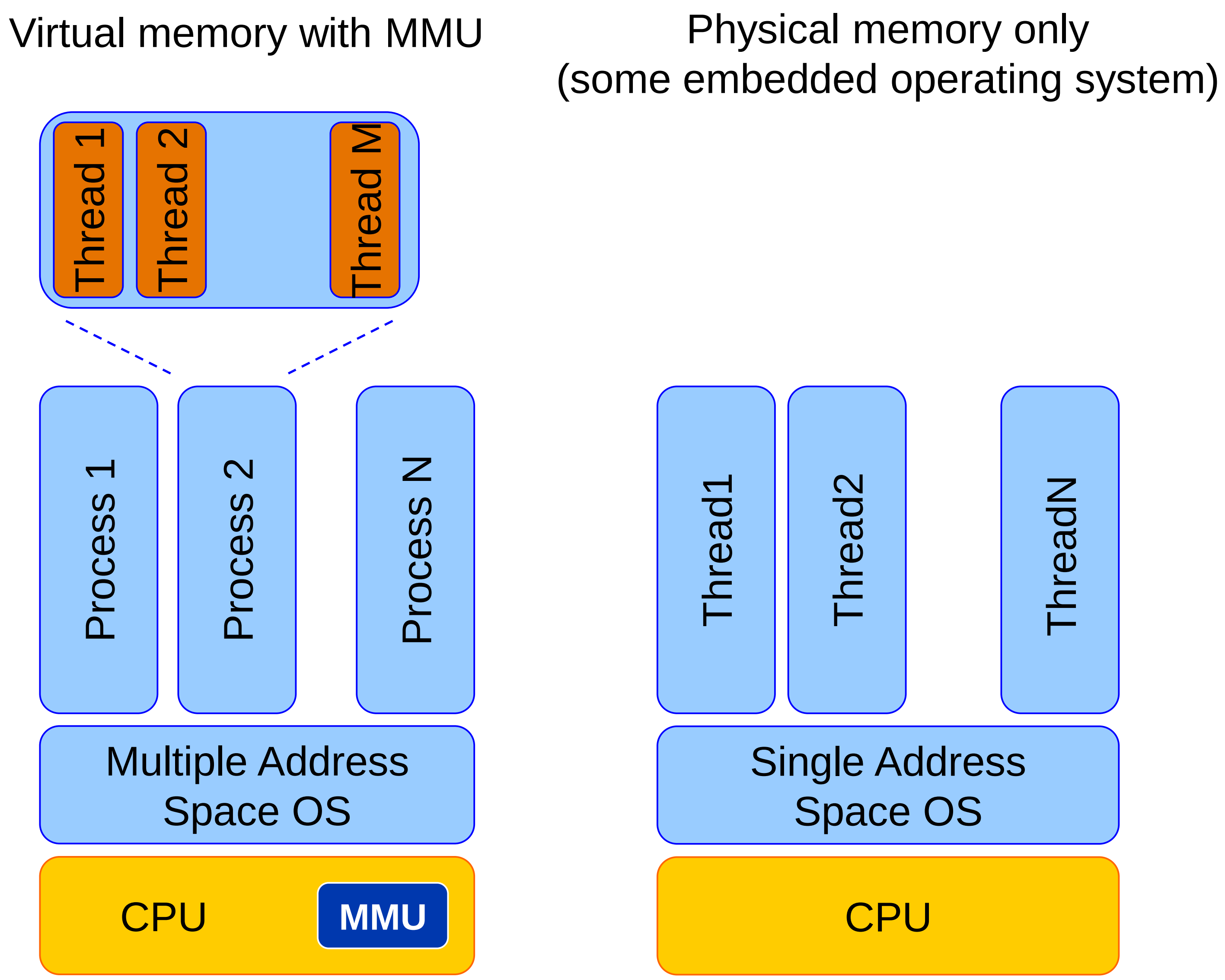
Advantages of using threads

- Low amount of resources are needed to create and terminate them
  - Some estimates it is 1/10 of the processes.
- Multiple running thread in an application
  - The GUI responsive of the application does some computation in the background
- Fast communication in-between threads running in the context of a process
  - They run in the same virtual memory
  - Stack is thread specific, but also shared as memory
    - Lot of problems may be caused by this
- Scalability
  - Multiple execution unit can be utilizes in one application

Consequences of using threads

- Communication using shared memory is dangerous
  - The consistency of data structures used for communication may be violated by multiple threads accessing them
  - We are going to deal with this problem later in at least two lectures
  - Threads running in the context of different processes must use system calls for communication
    - It is needed less frequently, because closely interrelated functionalities can be implemented using thread in a process

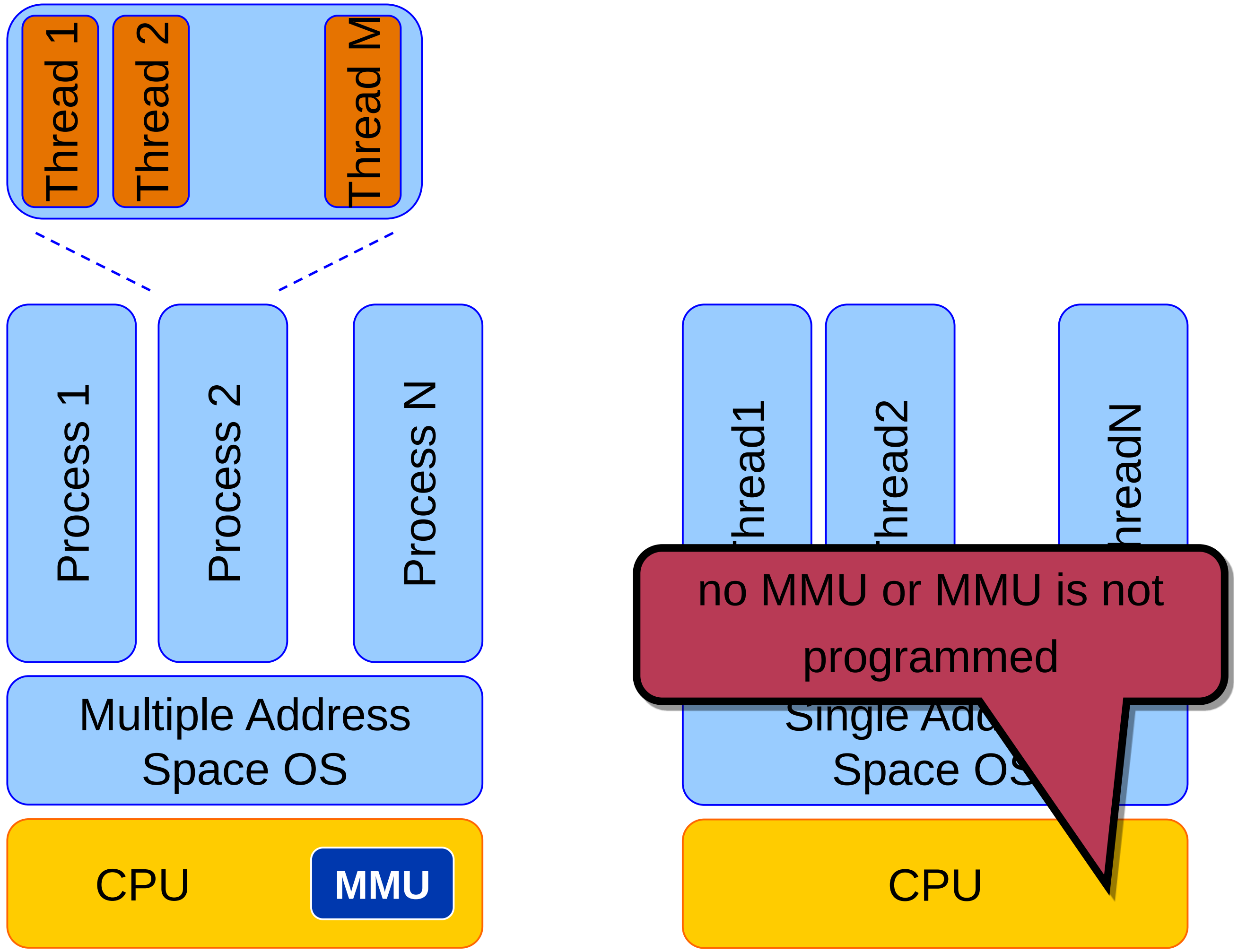
HW support


Virtual memory with MMU
Physical memory only
(some embedded operating system)
Thread 1
Thread 2
Thread M
Process 1
Process 2
Process N
Thread1
Thread2
ThreadN
Multiple Address Space OS
Single Address Space OS
CPU
MMU
CPU

HW support

Virtual memory with MMU

Physical memory only
(some embedded operating system)



Coroutine or fiber

- Cooperative multitasking
  - Inside a process or a thread
    - OS support or programming language level implementation
    - On the OS level the process or thread is scheduled
    - The scheduling of coroutines or fibers are in the hand of the programmer (cooperative scheduling).
  - Coroutine: programming language level construct
    - Haskell, JavaScript, Modula-2, Perl, Python, Ruby, etc.
  - Fiber: system (OS) level solution
    - Win32 API (ConvertThreadToFiber and CreateFiber).
    - Symbian

Coroutine

- Generalization of the Subroutine
  - Subroutine:
    - LIFO (Last In/called, First Out/returns).
    - Single entry point, multiple exit points (return/exit)
    - The stack is used to pass parameters and return value
  - Coroutine:
    - First entry point is the same as in case of the subroutine
    - After that its entry point is after the last exit point!
    - Transfer is with the „yield to Coroutine_id" call.
    - ***Cannot use stack***, it never returns!

Coroutine example, 1st call

```
var q := new queue

coroutine produce
    loop while q is not full
        create some new items
        add the items to q
    yield to consume


coroutine consume
    loop while q is not empty
        remove some items from q
        use the items
    yield to produce
```

Coroutine example, all other calls

```
var q := new queue
coroutine produce
    loop while q is not full
        create some new items
        add the items to q
    yield to consume

coroutine consume
    loop while q is not empty
        remove some items from q
        use the items
    yield to produce
```

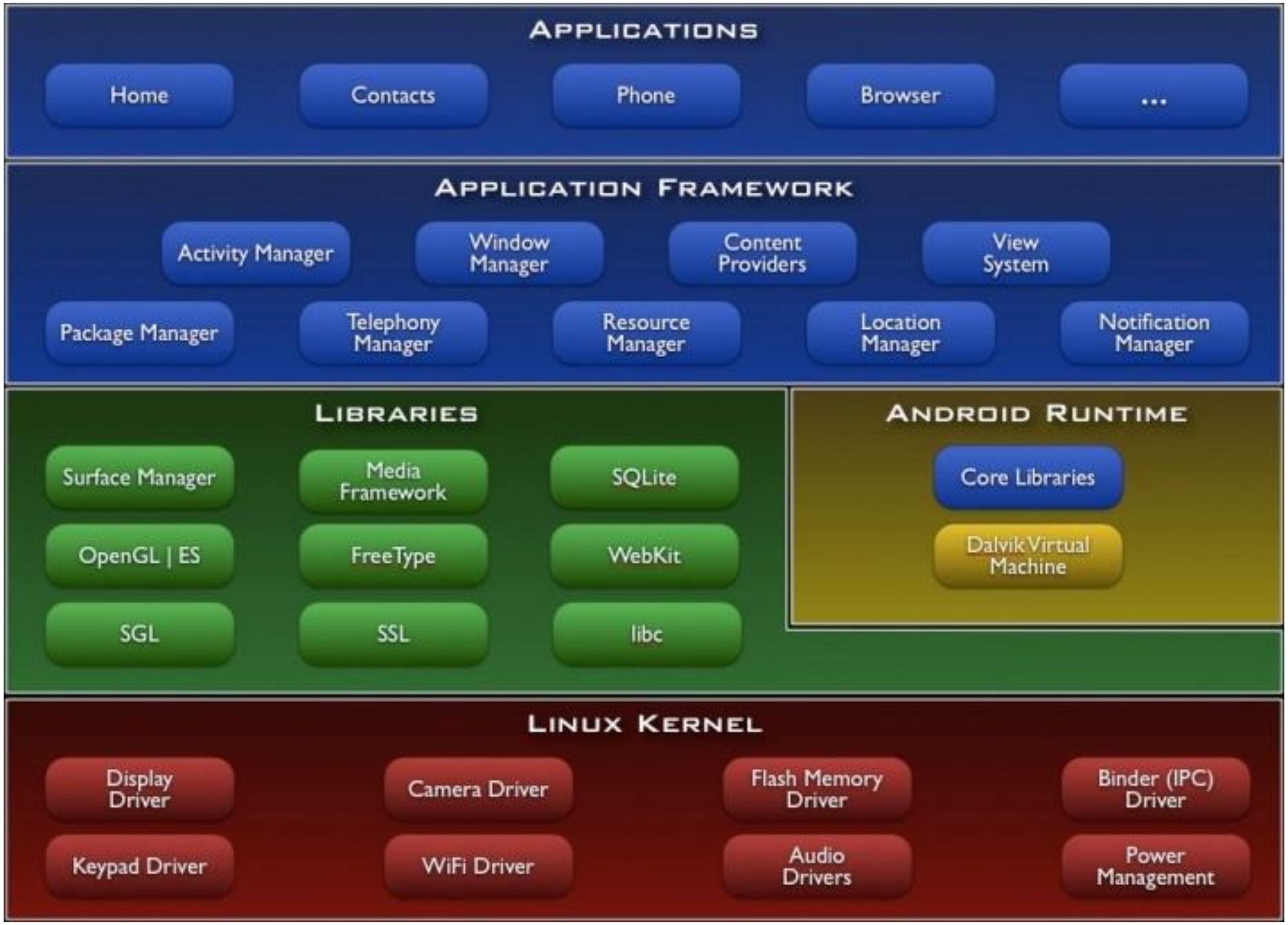
Evaluation of coroutine and fiber

- For problems solvable with cooperative multitasking
- For stack based environments (e.g. C/C++) implementation is hard if not done on the system level (fiber)
- No resource sharing is required
  - No specific OS calls for resource sharing
  - Less overhead
- The OS schedules them in a thread, they cannot utilize multiple execution units

Some other approaches…

- Android is based on Linux
  - Based on the process and thread support it builds an interesting framework for supporting mobile applications
- Let’s see it in details…

Android


Applications
Home
Contacts
Phone
Browser
...
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Package Manager
Telephony Manager
Resource Manager
Location Manager
Notification Manager
Libraries
Surface Manager
Media Framework
SQLite
OpenGL | ES
FreeType
WebKit
SGL
SSL
libc
Android Runtime
Core Libraries
Dalvik Virtual Machine
Linux Kernel
Display Driver
Camera Driver
Flash Memory Driver
Binder (IPC) Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management

Android and Linux

- Application Security Sandbox
- Android applications run in an application specific instance of Dalvik Virtual Machine (VM)
  - The Dalvik VM runs in a UNIX process
    - Thread and virtual memory support of Linux are used
  - Dalvik is developed for mobile/embedded use
    - Low memory usage
    - Register based, not stack based VM
- Every running Android application gets its own Linux User ID (UID), to deny access to other applications
  - By default it can access only files created by the application
  - *Principle of least privilege*
- Properties of the application is described in the Manifest File
- An application can be terminated by the OS any time in case of low resource availability
  - It is done by the OS automatically
  - There are “Task managers” on the Android Market to do this
  - No “Exit/Quit” button in most of the Android Apps (not needed)

Android application components 1.

- Relevant from the point of view of task implementation
- Activity
  - A screen with user interface
  - Multiple one in an application
  - In an application activities are independent entities, but they interoperate while the user accesses the application
  - It can be 3 states:
    - Resumed (active screen), the user can “tap” on it
    - Paused (inactive, visible, part of covered by the active screen),
    - Stopped (inactive and not visible screen)
    - Only activities visible on the screen are executed, all the others are stored with their state (resource optimization)
    - Lifecycle callbacks to inform the application on activity state changes
  - Other applications may start an activity in the application If that is allowed by the application

Android application components 2.

- Service
  - It runs in the background without any user interface
    - For background tasks running continuously, such as the MP3 playing component of an MP3 player
    - It does not create a thread for itself, if it is CPU intensive, a thread must be created for it for better user experience
  - Started service
    - It runs as long as it finishes its task, the application has minimal control over it
    - It can run longer than the application starting it
    - Example: Downloading a big file
  - Bound service
    - Its lifecycle bound to the application
    - It provides a well-defined interface to the application
    - Example: Background MP3 player service controlled from an application (play, stop, forward, backward, volume, speed, etc.)
  - Lifecycle callbacks are present here also