EMBEDDED SOFTWARE DEVELOPMENT

INTRODUCTION

- As the sophistication of embedded processors (or "microcontrollers" as they are popularly known) has increased over the past decade, more advanced software architectures have become viable for embedded design (Q: what are some of the forces driving embedded processor development?)

- on simple 8- or 16-bit MCUs, the traditional approach to embedded programming has been the "super-loop" or "cyclic executive" implemented in bare metal fashion; no operating system is needed for such an implementation:

EX. a bare-metal super-loop template in C:

```
// pre-processor directives
#include <stdio.h>
   :
#define ____
   :
// main code begins
int main()
{
    // variable declarations
       :

    // super-loop begins
    while (1)
    {
        // loop runs forever, exiting
        // only in the case of an error
           :

    }
    return 1; // non-zero exit indicating an error
              // has occurred
}
```

super-loop

- indeed, the bare-metal super-loop approach works on any type of MCU, and may be preferred in critical applications where performance must be guaranteed by the firmware engineer (Q: why? discuss)

- however, an operating systems approach may be desired for many applications which require one or more of the following:

  1) the coordination of multiple concurrent processes or tasks;
  2) secure, advanced communication protocols;
  3) advanced graphical user interfaces;
  4) interoperability with mobile and/or networked devices;
  5) features or programs available only with an operating system (such as driver support);
  6) the management of multiple processor cores.

- today, the availability of sophisticated 32- and 64-bit MCUs, mean that virtually all software techniques available for server or desktop machines can be adapted to the embedded systems environment

- in this class, we are going to examine the application of multi-tasking and realtime operating systems techniques to embedded software development

→ **What is an operating system?**

- **DEFN**: an operating system is the foundational software which controls and allocates computer resources (i.e., CPU cores, RAM, memory-mapped I/o devices, and peripherals). ■

- most operating systems have a central supervising program, known as a kernel or scheduler.

- tasks performed by a kernel include:

  → process scheduling: essential in a multi-tasking context.

  → memory management: allocating memory dedicated to processes/tasks, possibly the use of virtual memory management.

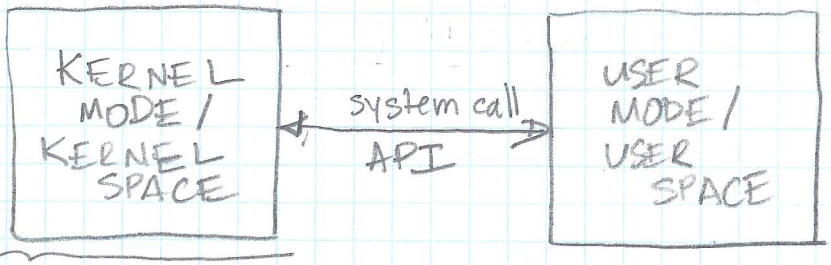
  → provision of a file system: to allow the files to be created, retrieved, updated, etc., on a mass storage medium

  → access to devices: all I/O access may be mediated for security and stability, via the kernel.

  → networking capabilities: all communications can be mediated by the kernel

  → provision of a system call application programming interface (API): to allow processes/tasks to request certain actions from the kernel; known in a Linux environment as "system calls".

- the presence of the kernel often implies a partitioning of memory and CPU resources



the kernel mode is also known as "supervisor mode" which provides privileged access to all system memory.

user mode implies only memory designated as being in "user space" is accessible.

this prevents processes from accessing sensitive areas of memory, outside of their scope

- **TECHNICAL POINT:** we use the terms "process" and "task" interchangeably: processes are often associated with Linux, whereas tasks are often associated with real-time operating systems.

- **DEFN (process vs. program):**

  In Linux, a **process** is an instance of an executing program; a **program** is a file containing the information needed to construct a process at run-time.

MULTI-TASKING

- a multi-tasking OS is one which maintains multiple processes in memory simultaneously and schedules execution in order to provide CPU time for all processes (in accordance with priorities and other criteria)

- on a single-core CPU, only one process or task can be running at any given instant of time, however, the kernel or scheduler may **pre-empt** a process, and start executing a different process by performing a **context switch**; as a result, if context switches occur frequently enough, we have the "illusion" of simultaneous process execution.

- there are many applications in which a computer must handle multiple concurrent phenomena.

EX. a flight-control computer may have to simultaneously handle:

1) input from a human pilot;
2) sensor data from many devices measuring pressure, temperature, speed, orientation and acceleration;

3) control of multiple actuators for adjusting aircraft velocity and orientation, such as the servos for flaps; ailerons and rudders;

4) the display of information to a human pilot or navigator;

5) the relaying of telemetry data to a satellite or radio towler.

- **TECHNICAL POINT (PRE-EMPTION):**

Depending on who you talk to, the term pre-emption can have slightly different meanings.

For Linux developers, pre-emption simply means that the kernel decides (rather than allowing processes to decide for themselves) when a context switch from one process to another occurs. The goal of the Linux kernel is to ensure fair "time sharing" amongst all processes.

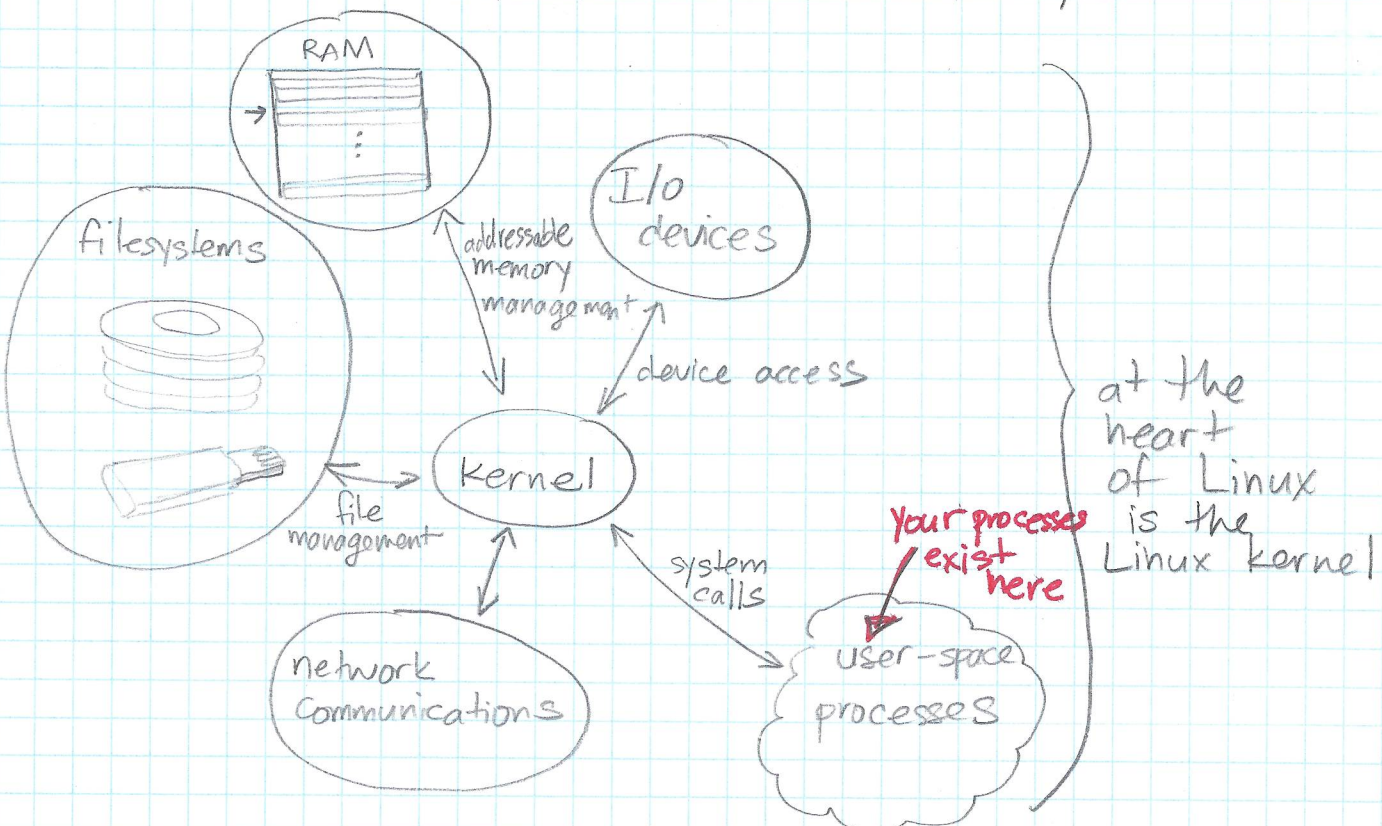
For developers of real-time operating systems, preemption of a process by the scheduler may occur at any time (modulo the smallest system time interval, known as the "tick") in order to ensure that processes meet their deadlines. Thus the terms "real-time" and "preemptive" are used interchangeably.

But because the Linux kernel is not deadline-driven, it is not real-time, and, therefore, not considered pre-emptive by real-time OS developers.

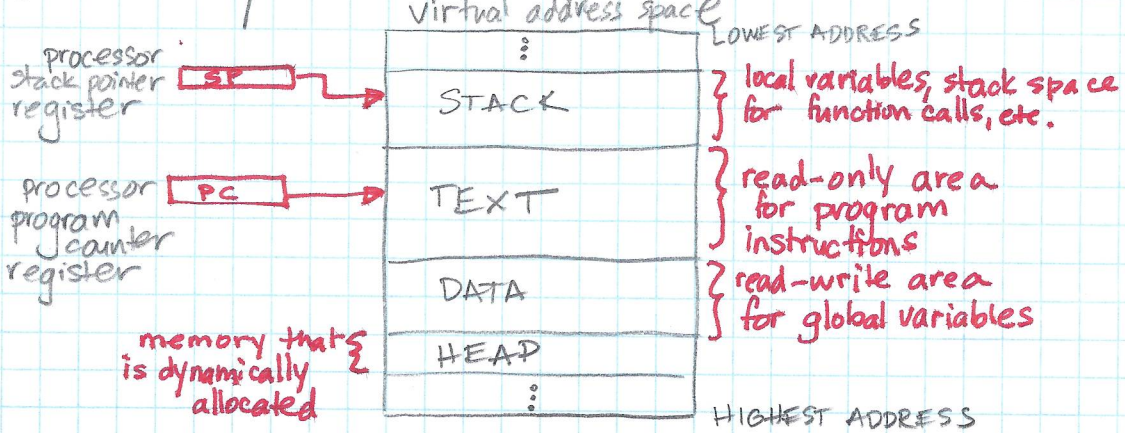
- **DEFN (context switch):** a context switch occurs when the kernel or scheduler saves the state of a process, thread or task so that the state can be restored to allow execution to resume later; context switching allows multiple processes or threads to share a single CPU.

MULTI-TASKING IN LINUX (PTHREADS)

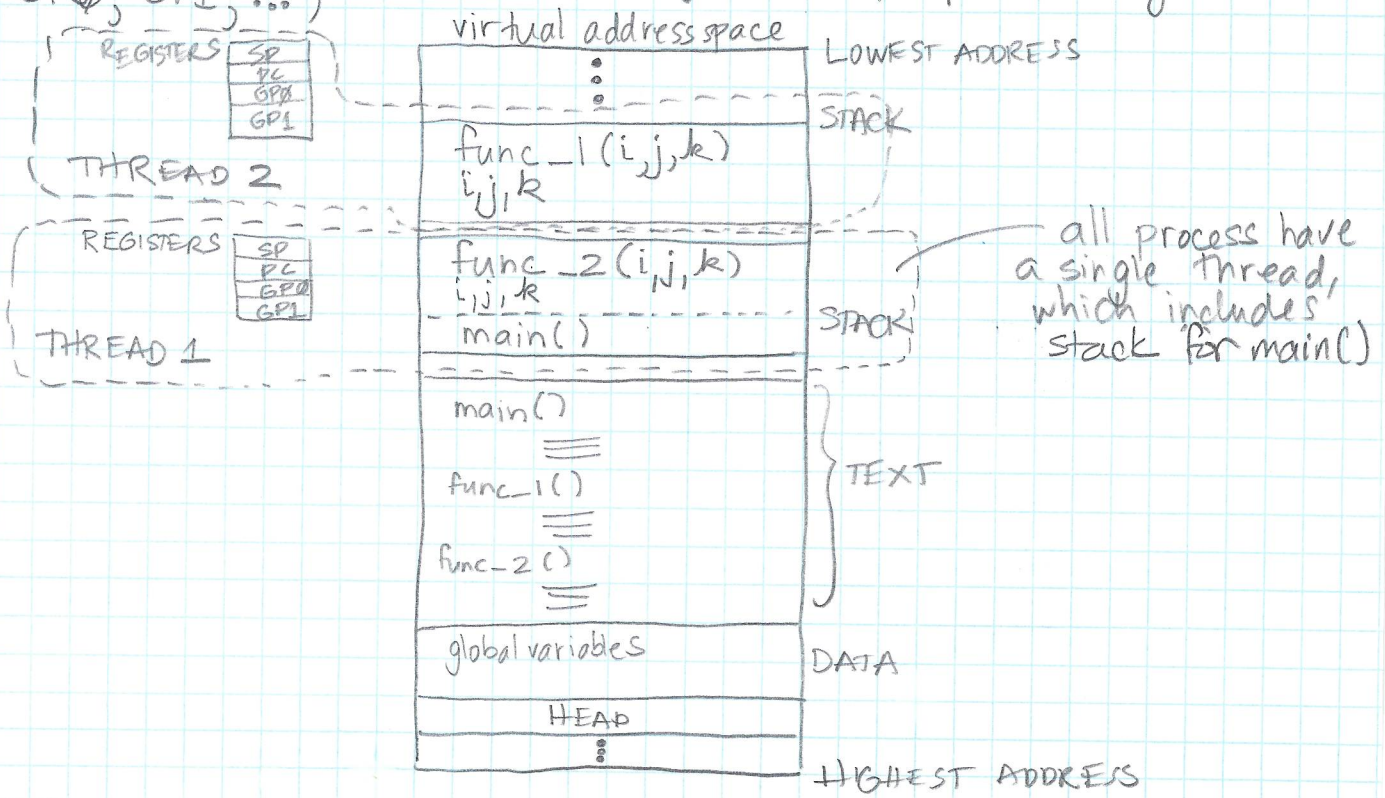
- Linux (or GNU-Linux) is a well-known multi-tasking OS based on the UNIX architecture which originated in the late 1960s. The original UNIX OS was written in assembly language for the PDP-7 mini-computer but was re-written in C and ported to many platforms within a few years. Linux itself is a port of UNIX to PCs (the first release was intended for 80386 Intel machines)



- the Linux process in detail: recall that a process is an instance of an executing program (the program itself is just a file).
  - → a process occupies a structured area of memory:



- multiple such processes can exist simultaneously under Linux. processes can "spawn" new (child) processes using the fork() system call, and while multiple processes constitute a multi-tasking scenario, we will take another approach to multi-tasking, known as threading
- specifically we will make use of the pthread library in Linux; the pthread library bears some resemblence to the free RTOS framework, making pthreads an excellent stepping stone to RTOS development
- a <u>thread</u> is essentially a "sub-process" that exists within a process; multiple threads can exist in the same process. Because they share, in effect, the same stack, text, data and heap, inter-thread communication is much more straightforword and efficient than inter-process communications (not discussed here).
- although threads have separate Stack space within a process, they share the Text, Data and Heap memory areas with the process. in addition, each thread has its own copy of the CPU registers, including stack pointer (SP), program counter (PC) and general purpose registers (GP0, GP1, ...)



- study the provided pthread example programs, pthreads_0.c and pthreads_1.c
- in Eclipse, you must add the pthread library to your Build Settings — ask your instructor how to do this — in order to build these projects.
- Note the following pthread-specific features:

  → pthread_t : a pthread type for "handles" to manage threads

  → pthread_create() : a function to create a thread (usually based on a function)

  → pthread_join() : a function that allows a thread to complete, storing the return value

  → pthread_cancel() : a function that terminates a thread, using the thread's handle.

- **LAB QUESTIONS:**

1) How many threads are in pthreads_0.c? Where is the stack space for threadFunction()? Where is the stack space for main()?

2) Consider the program pthreads_1.c. Build the program and run it on your host machine
   a) Build the program on your Beaglebone Black or Pocket Beagle; use the commands:

```
beaglebone$ gcc -pthread pthreads_1.c -o pthreads_1
beaglebone$ ./pthreads1
```

   in the appropriate directory. How do the executions on the two platforms compare?

   b) Given that the two thread functions are identical, do you expect them to run an equal number of times? Do they run an equal number of times? Why or why not?

HOMEWORK (TO BE TURNED IN LATER AS AN ASSIGNMENT)

- COMPLETE THE main() FUNCTION FOR pthreads_2.c IN THE COURSE GIT REPOSITORY
- SHOW THAT YOUR PROGRAM WORKS
- PROVIDE A CODE LISTING (PROFESSIONAL-STYLE FORMATTING, PLEASE!)