

ECE3411 – Fall 2016

Lecture 6a.

Task Based Programming Revisited

Real Time Operating Systems

Marten van Dijk, Syed Kamran Haider

Department of Electrical & Computer Engineering

University of Connecticut

Email: {vandijk, syed.haider}@engr.uconn.edu

Copied from Lecture 6a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UConn



Example: How are the tasks scheduled?

```
while (1)
{
    if (task1_timer == 0)           // if task1_timer is not already equal to 0,
                                    // it is being decremented every 1 millisecond
                                    // during a timer ISR
    {
        task1_timer = t1;
        task1();                    // task1 takes m1 milliseconds
    }

    if (task2_timer == 0)           // if task2_timer is not already equal to 0,
                                    // it is being decremented every 1 millisecond
                                    // during a timer ISR
    {
        task2_timer = t2;
        task2();                    // task2 takes m2 milliseconds
    }
}
```

Example Cont'd

- Suppose $t1=5$, $m1=1$, $t2=10$, and $m2=15$
- What is the frequency $f1$ in Hz at which `task1()` is called?
- What is the frequency $f2$ in Hz at which `task2()` is called?

- Answer:
 - Since both `task1_timer` and `task2_timer` are decremented to 0 during the execution of `task2()`, `task1()` and `task2()` alternate.
 - Therefore, $f1=f2 = 1$ every 16 ms which is equal to $1000/16$ Hz.

Example Cont'd

- Suppose $t_1=20$, $m_1=1$, $t_2=10$, and $m_2=15$
- What is the frequency f_1 in Hz at which $\text{task1}()$ is called?
- What is the average frequency f_2 in Hz at which $\text{task2}()$ is called?

- Answer:
 - Since task2_timer is decremented to 0 during the execution of $\text{task2}()$, $\text{task2}()$ is called as often as possible.
 - When it is $\text{task1}()$'s turn to be executed, it takes more than one and less than two executions of task2_timer to get $\text{task1}()$ decremented to 0.
 - Therefore, the execution pattern converges to a repetition of $\text{task2}()$ (takes 15 ms), $\text{task2}()$ (takes 15 ms), $\text{task1}()$ (takes 1 ms) giving
 - a frequency $f_1=1000/31$ Hz and
 - an average frequency $f_2=2 * 1000/31$.

Example Cont'd

- Suppose $t_1=20$, $m_1=1$, $t_2=25$, and $m_2=15$
- What is the frequency f_1 in Hz at which $\text{task1}()$ is called?
- What is the frequency f_2 in Hz at which $\text{task2}()$ is called?
- Answer:
 - During the time that $\text{task2}()$ is executed (which takes 15 ms), task1_timer (which initial value is 20) is decremented to a value $v \leq 5$.
 - The MCU will be idle for v ms after which task2_timer is decremented to $25-15-v$ and task1_timer just turned into 0.
 - So, after v ms $\text{task1}()$ is executed taking 1 ms during which task1_timer reduces to 19 and task2_timer reduces by 1 to $9-v$.
 - The MCU will be idle for another $9-v$ ms after which task1_timer is equal to $10+v$ and task2_timer just turned into 0.
 - Now $\text{task2}()$ is executed (which takes 15 ms) after which task1_timer is equal to 0 and task2_timer is equal to 10.
 - The same argument is now repeated for $v=0$ showing that the execution pattern converges to a repetition of $\text{task2}()$ (takes 15 ms), $\text{task1}()$ (takes 1 ms), idle time (takes 9 ms) giving
 - a frequency $f_1=f_2=1000/25$ Hz.

Example Cont'd

- Suppose $t_1=4$, $m_1=1$, $t_2=8$, and $m_2=4$.
- Assume initially $\text{task1_timer} = 0$ and $\text{task2_timer} = t_2$
- What is the average frequency f_1 in Hz at which $\text{task1}()$ is called?
- What is the average frequency f_2 in Hz at which $\text{task2}()$ is called?
- Answer:
 - Task 1 executes during the intervals $[12n, 12n+1]$, $[12n+5, 12n+6]$, for integers $n \geq 0$.
 - Task 2 executes during intervals $[12n+8, 12n+12]$ for integers $n \geq 0$.
 - This gives frequencies $f_1 = 1000 \cdot 2/12$ Hz and $f_2 = 1000/12$ Hz.

Real Time OS

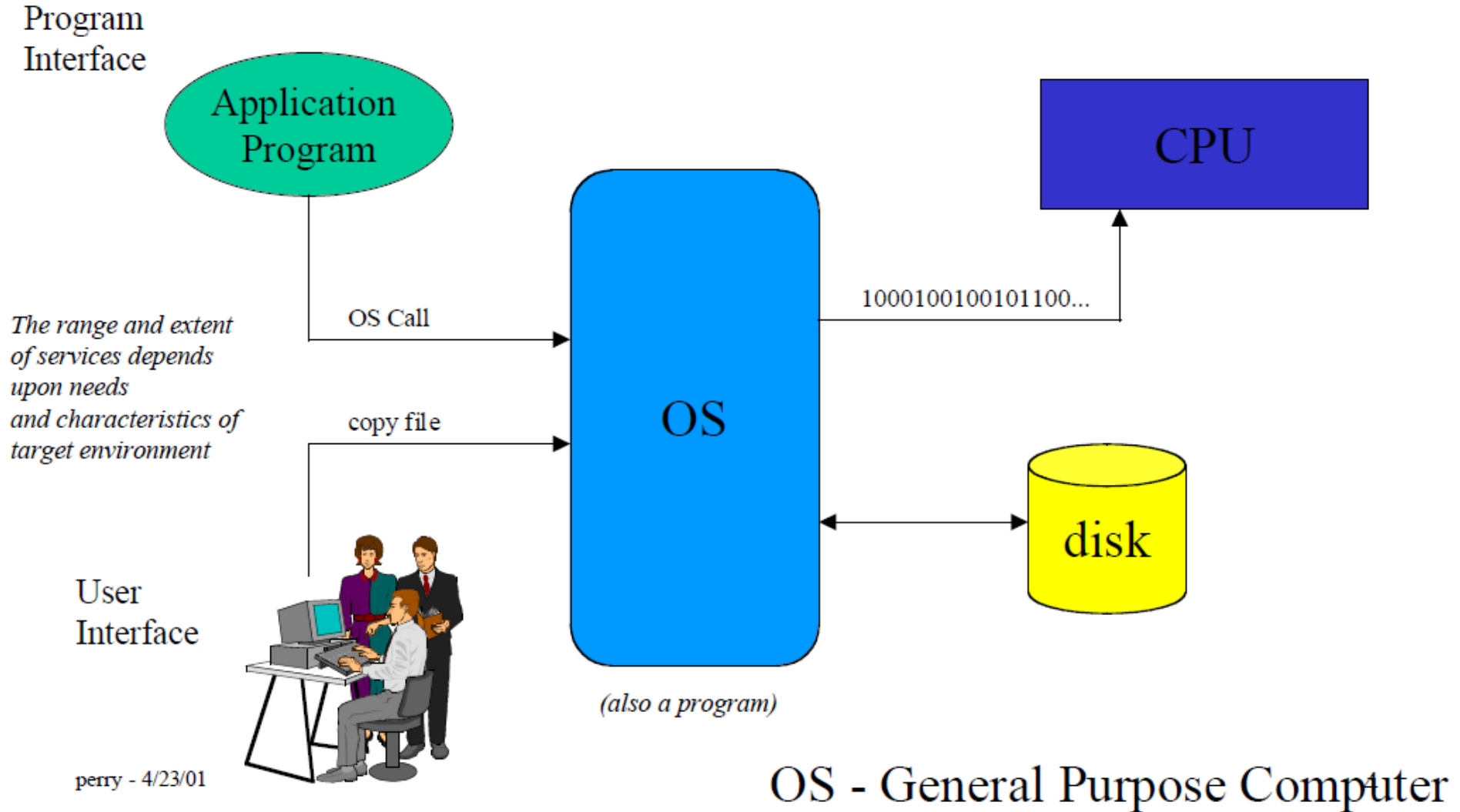
- What follows is extracted or copied from MIT 16.07 (Perry)
- What is an Operating System (OS)?
- Basic operating system design concepts
- What is a Real Time OS (RTOS)?
- Realtime Kernel Design Strategies

What is an operating system?

An organized collection of software extensions of hardware that serve as...

- control routines for operating a computer (for example, to gain access to computer resources (like file I/O))
- an environment for execution of programs

OS Services



What does an OS do?

- Manages computer system resources (processor, memory, I/O, etc.)
 - Keeps track of status and “owner” of each resource
 - Decides who gets resource
 - Decides how long the resource can be in use
- In systems that support *concurrent execution* of programs, it
 - Resolves conflicts for resources
 - Optimizes performance given multiple users

Types of operating systems

- Simplest = small kernel on embedded processor
- Most complex = full featured commercial OS
 - Multi-user security
 - Graphics support
 - Networking support
 - Peripherals communication
 - Concurrent execution of programs

OS Hierarchy

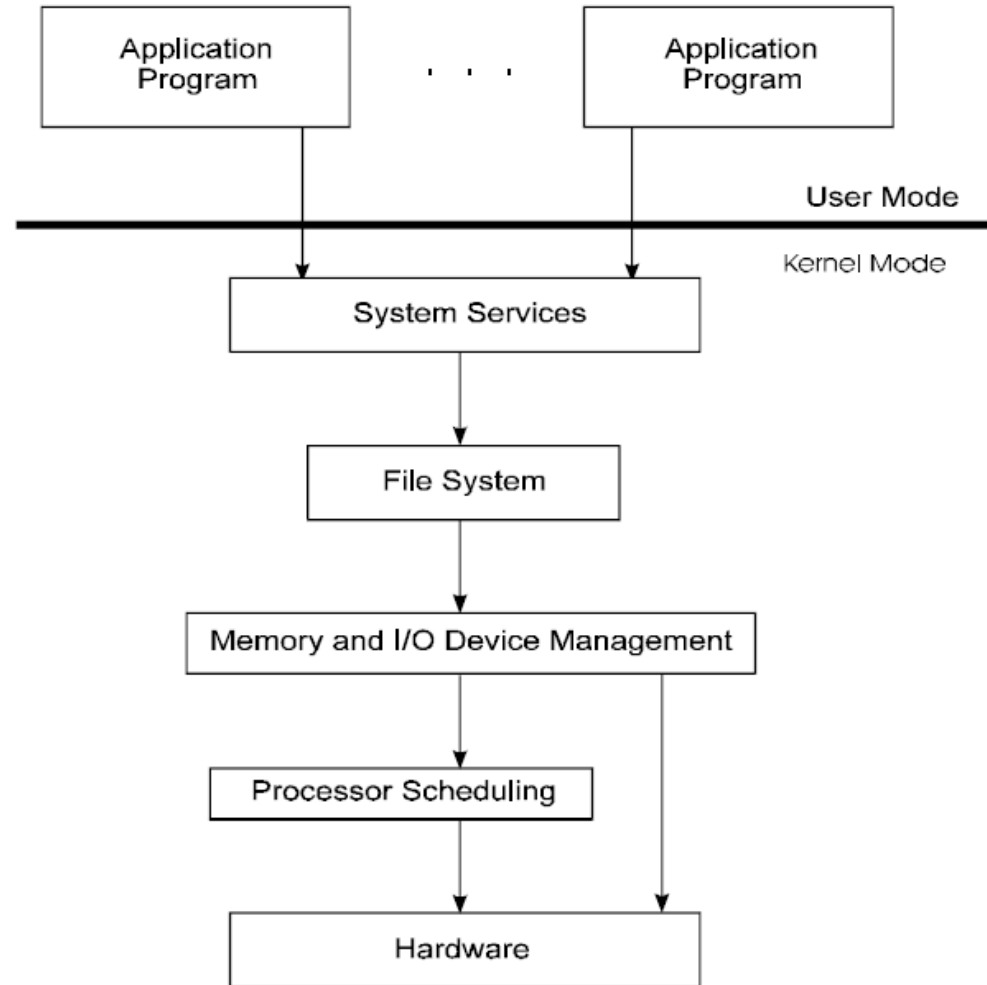
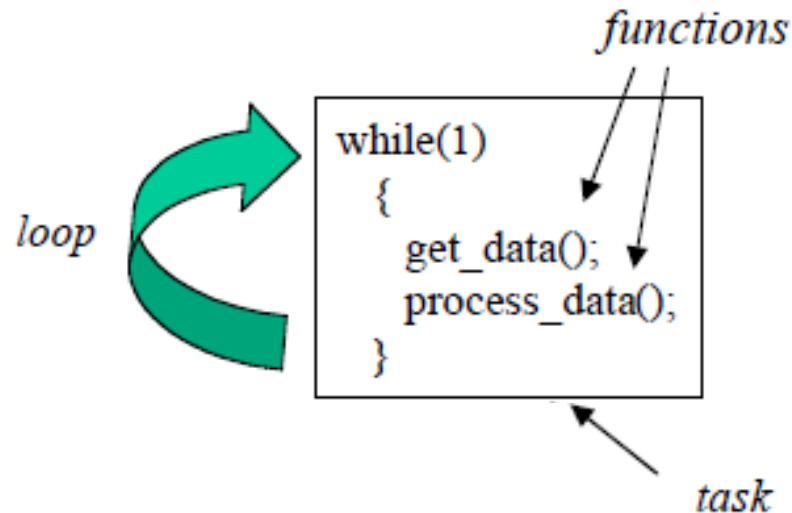


Figure 2.2: Layered Operating System

Tasks & Functions

- A task is a process that repeats itself
 - Loop forever
 - Essential building block of real time software systems
- A function is a procedure that is called. Once called, it runs and then exits possibly returning a value.



RTOS

- Often RTOS = OS Kernel
- An embedded system is designed for a single purpose so the user shell and file/disk access features are unnecessary
- RTOS gives you control over your resources
 - No background processes that “just happen”
 - Bounded number of tasks
- RTOS gives you control over timing by allowing:
 - Manipulation of task priorities
 - Choice of scheduling options

Components OS Kernel

- Task Scheduler: To determine which task will run next in a multitasking system
- Task Dispatcher: To perform necessary bookkeeping to start a task
- Intertask Communication: To support communication between one process (i.e. task) and another

Realtime Kernel Design Strategies

- Polled Loop Systems
- Interrupt Driven Systems
- Multi-Tasking
- Foreground/Background Systems

Polled Loops

- Simplest RT kernel
- A single and repetitive instruction tests a flag that indicates whether or not an event has occurred
 - Examples: Non-blocking LCD instructions, Non-blocking “get string” over the UART channel
- No intertask communication or scheduling needed. Only single tasks exist
- Excellent for handling high-speed data channels, especially when
 - Events occur at widely spaced intervals and
 - Processor is dedicated to handling the data channel

Polled Loops

- Pros:
 - Simple to write and debug
 - Response time easy to determine (as compared to our task-based programming example with two rather than a single task)
- Cons:
 - Can fail due to burst of events
 - Generally not sufficient to handle complex systems
 - Waste of CPU time, especially when event being polled occurs infrequently

Using Polled Loops

- Often used inside other real time schemes to, e.g.,
 - Poll a suite of sensors for data
 - Check for user inputs (keyboard, keypad, UART data)

- Opposite of interrupt driven systems

What is an Interrupt (recap)?

- A HW signal that initiates an event
- Upon receipt of an interrupt, the processor
 - Completes the instruction being executed
 - Saves the program counter (so as to return to the same execution point)
 - Loads the program counter with the location of the interrupt handler code (ISR)
 - Executes the interrupt handler (ISR)
- In practice, real time systems can handle several interrupts in priority fashion
 - Interrupts can be enabled/disabled (By setting appropriate registers.)
 - Highest priority interrupts serviced first (Which ones have the highest priority in Atmega328P?)
- Processor must check for interrupts very frequently: If any have arrived, it stops immediately and runs the associated ISR
 - Processor repeats: do one operation; check interrupts; if interrupts then suspend task and run ISR

ISR

- ISR is a program run in response to an interrupt
 - Disables all interrupts
 - Clears the interrupt flag that got it called
 - Runs code to service the event
 - Re-enables interrupts
 - Exits so the processor can go back to its running task
- Should be as fast as possible, because nothing else can happen when an interrupt is being serviced (when interrupts happen very frequently, tasks are being stalled and progress very slowly, in the worst case one instruction per ISR)
- Interrupts can be
 - Prioritized (service some interrupts before others)
 - Disabled (processor doesn't check or ignores all of them)
 - Masked (processor only sees some interrupts)

Examples interrupt-driven system

Interrupt Driven Software Examples

- IFF receiver sees a threat and interrupts an aircraft mission computer to sound a cockpit alarm
- Inertial Navigation Unit data (Δ velocities in north/east/up coordinates) is available at 32 Hz and interrupts the navigation software with new data when it is ready
- Sonar contact data interrupts signal processing software when new data is available
- Low altitude indicator triggers a fly-up command for a pilot