ECE3411 – Fall 2015

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

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 t1=20, m1=1, t2=10, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

- Answer:
  - Since task2_timer is decremented to 0 during the execution of task2(), task2() is called as often as possible.
  - When it is task1()'s turn to be executed, it takes more than one and less than two executions of task2_timer to get task1() decremented to 0.
  - Therefore, the execution pattern converges to a repetition of task2() (takes 15 ms), task2() (takes 15 ms), 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 t1=20, m1=1, t2=25, 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:
  - During the time that task2() is executed (which takes 15 ms), task1_timer (which initial value is 20) is decremented to a value v<=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 task1() is executed taking 1ms 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 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 task2() (takes 15 ms), task1() (takes 1 ms), idle time (takes 9 ms) giving
    - a frequency f_1=f_2=1000/25 Hz.

Example Cont'd

- Suppose t1=4, m1=1, t2=8, and m2=4.
- Assume initially task1_timer = 0 and task2_timer = t2
- What is the average frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

- Answer:
  - Task 1 executes during the intervals [12n,12n+1], [12n+5,12n+6], for integers n>=0.
  - Task 2 executes during intervals [12n+8,12n+12] for integers n>=0.
  - This gives frequencies f_1=1000*2/12 Hz and f_2=1000/12 Hz.

Real Time OS

- What follows is extracted or copied from MIT 16.07 (Perry)

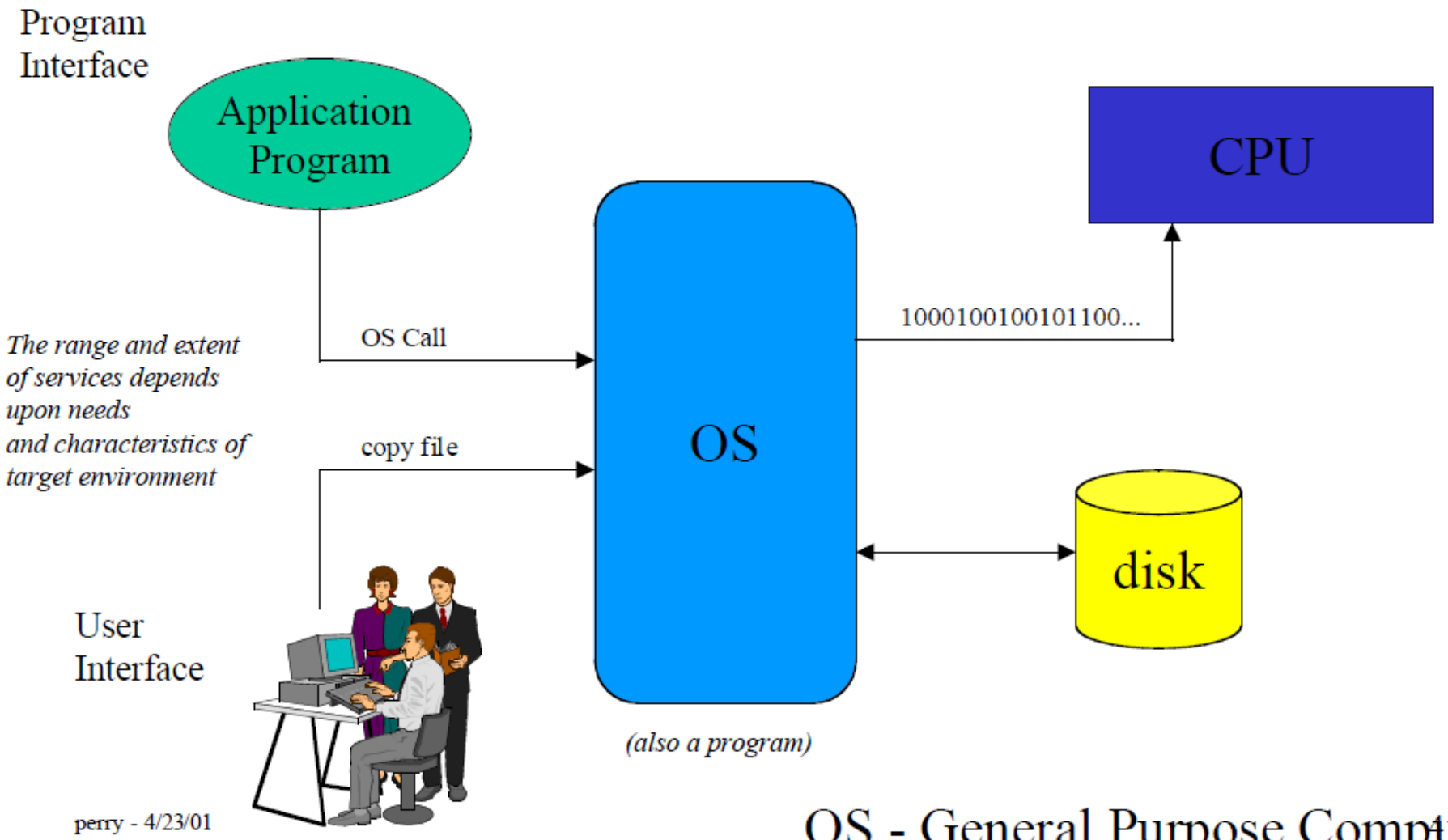
- What is an Operating System (OS)?
- Basic operating system design concepts
- What is a Real Time OS (RTOS)?
- Realtime Kernel Design Strategies

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



OS - General Purpose Computer

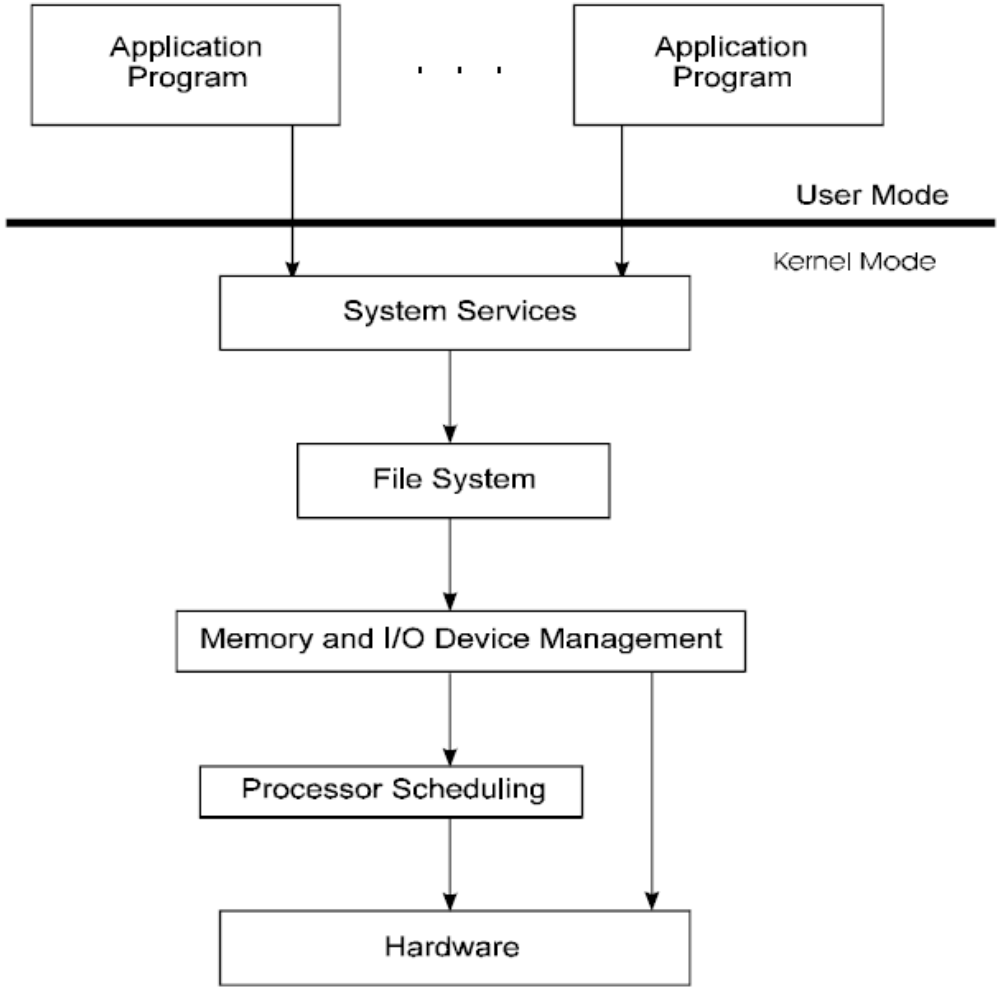
What does an OS do?

- Manages computer system resources (processor, memory, I/O, etc.)
  - Keeps track of status and "owner" of each resource
  - Decided 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

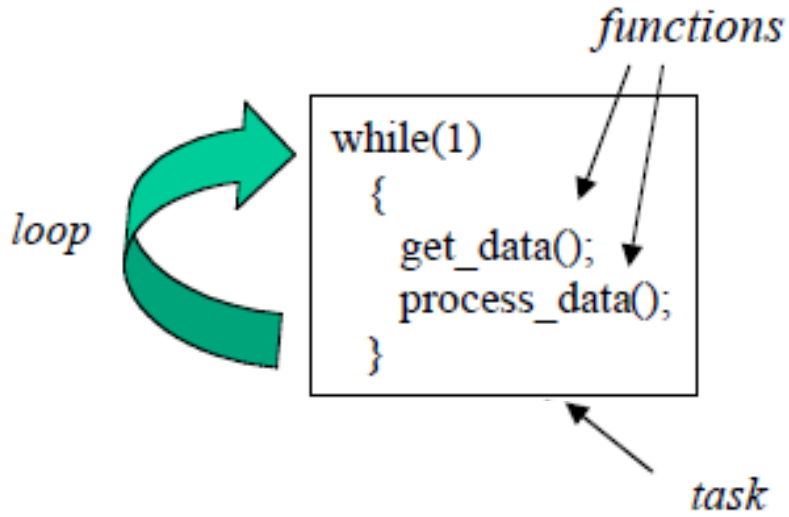


Taken from http://www.cloudbus.org/~raj/

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 and 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)
  - Disabed (processor doesn't check or ignores all of them)
  - Masked (processor only sees some interrupts)

Examples interrupt-driven system

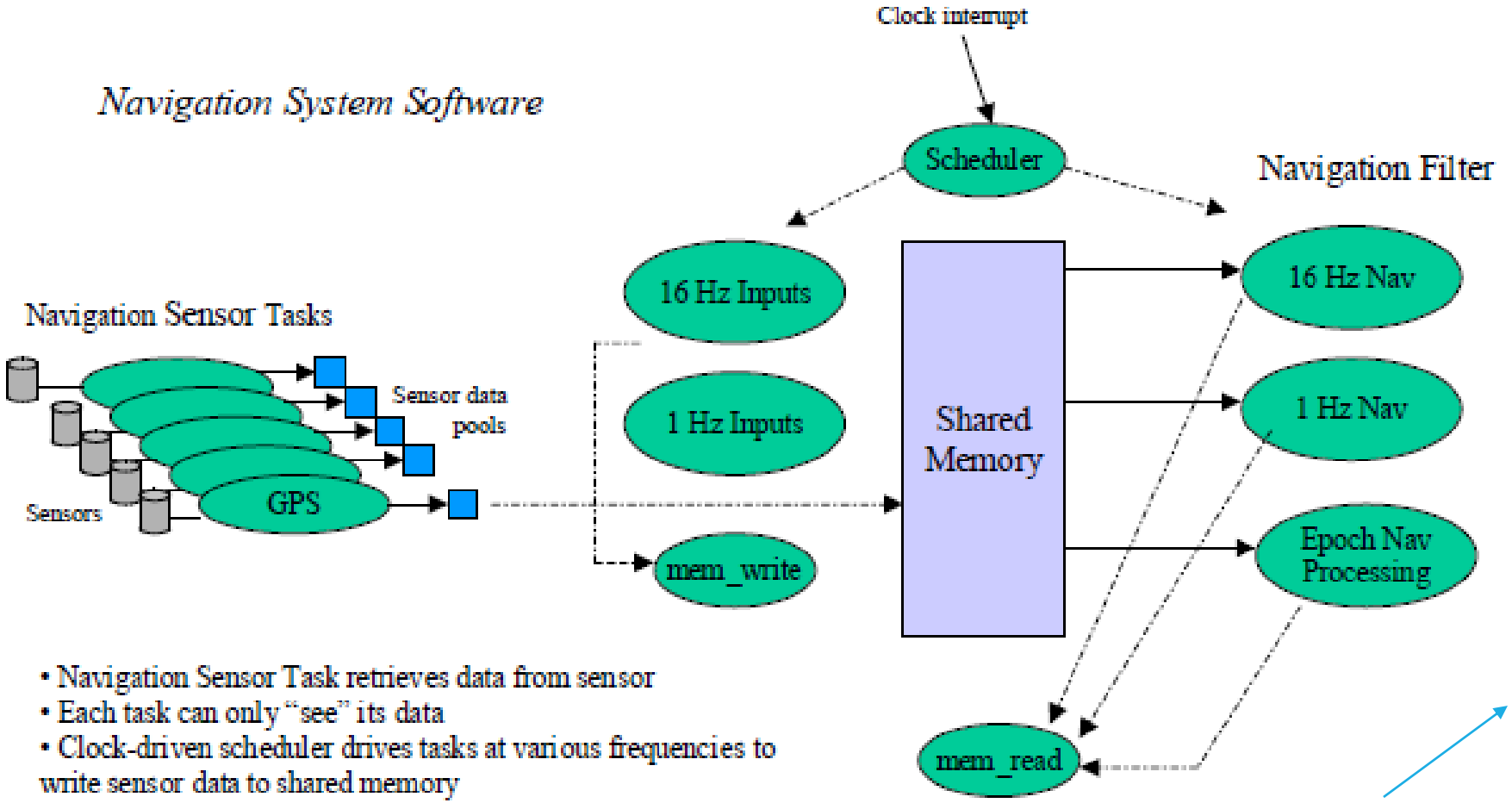
Interrupt Driven Software Examples

- 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

Multitasking

- Separate tasks that share one processor (or processors)
- Each task executes within its own context
  - Owns processor
  - Sees its own variables
  - May be interrupted
- Tasks may interact to execute as a whole program

Example



Mailbox is a circular buffer with a read and a write pointer

Context Switching

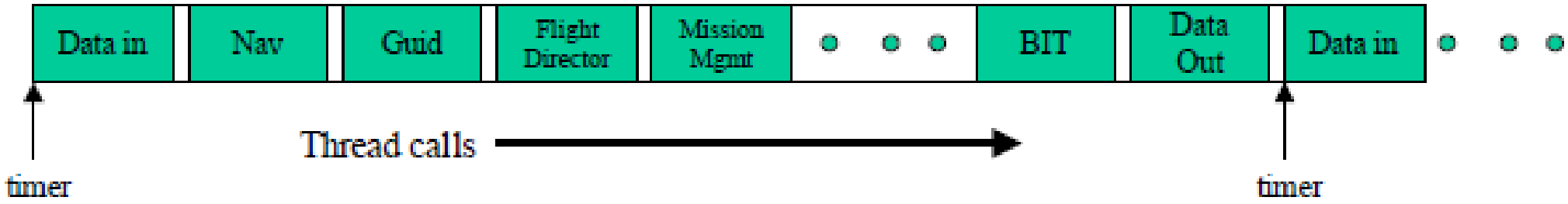
- When the CPU switches from one task to running another, its is said to have *switched contexts*
- Save the minimum needed to restore the interrupted process
  - Contents of registers
  - Contents of the program counter
  - Contents of coprocessor registers (if applicable)
  - Memory page registers
  - Memory-mapped I/O
  - Special variables
- During context switching, interrupts are often disabled
- Real time systems require minimal times for context switches

Multitasking

- How do many tasks share the same CPU?
  - Cyclic executive systems
  - Round robin systems
  - Pre-emptive priority systems

Cyclic Executive Systems

- Calls to statically ordered threads

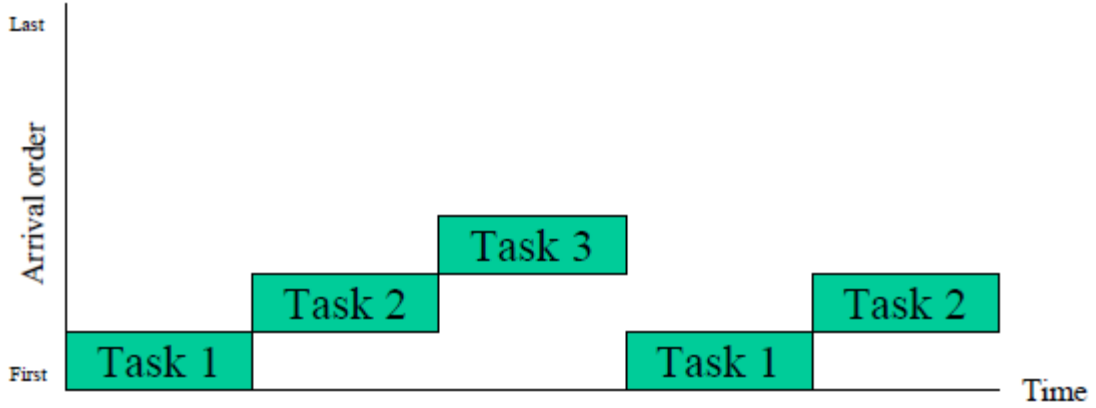


- Pros
  - Easy to implement (used extensively in complex safety critical systems)
- Cons
  - Not efficient in overall usage of CPU processing
  - Does not provide optimal response time

Round Robin Systems

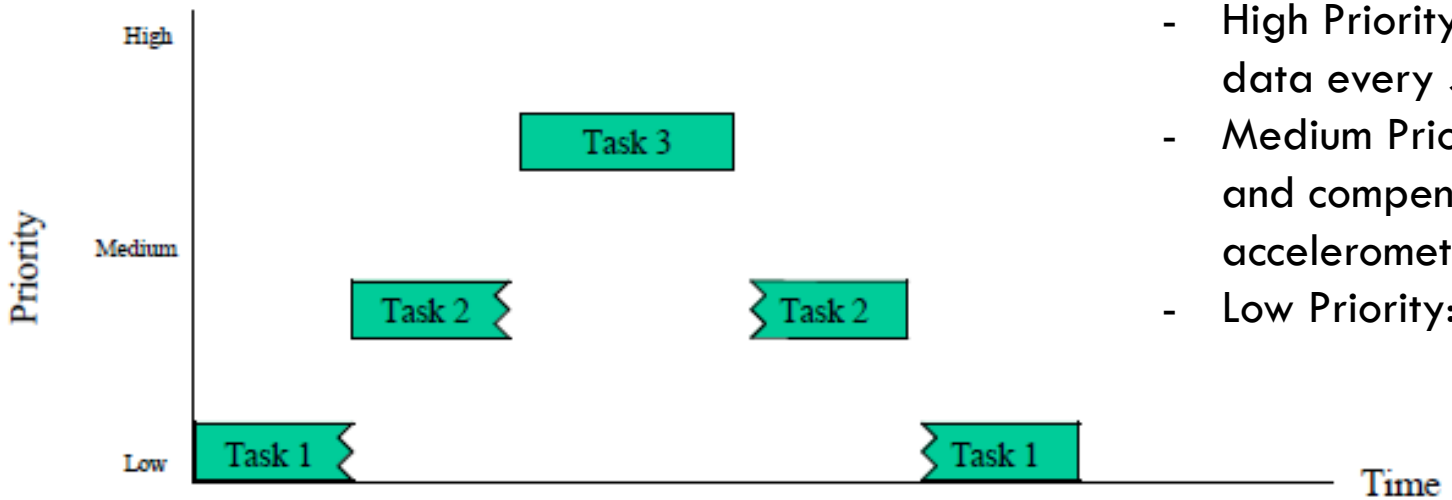
- Several processes execute sequentially to completion
- Often in conjunction with a cyclic executive
- Each task is assigned a fixed time slice
- Fixed rate clock initiates an interrupt at a rate corresponding to the time slice
  - Task executes until it completes or its execution time expires
  - Context saved if task does not complete

- Just like our task-based programming without fixed times slices per task



Pre-emptive Priority Systems

- Higher priority task can preempt a lower priority task if it interrupts the lower-priority task
- Priorities assigned to each interrupt are based upon the urgency of the task associated with the interrupt
- Priorities can be fixed or dynamic
  - Round Robin Systems - Preemptive Scheduling of 3 Tasks



Example: Aircraft Navigation System
- High Priority: Task that agthers accelerometer data every 5ms
- Medium Priority: Task that collects gyro data and compensates this data and the accelerometer data every 40ms
- Low Priority: Display update, Built-in-Test (BIT)

Problems Multitasking

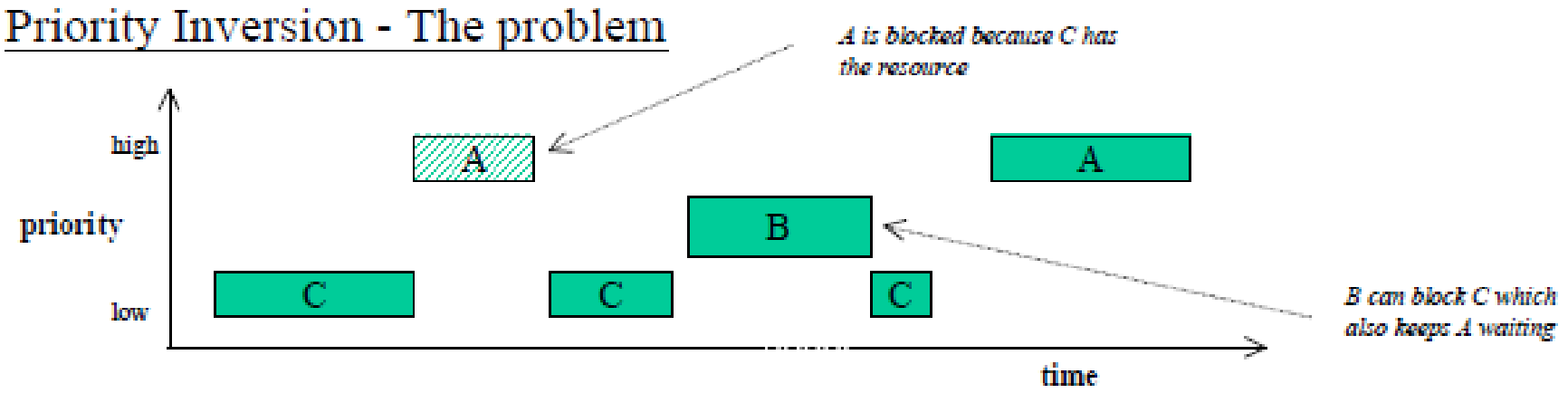
- High priority tasks hog resources and starve low priority tasks
- Low priority tasks share a resource with high priority tasks and block high priority tasks

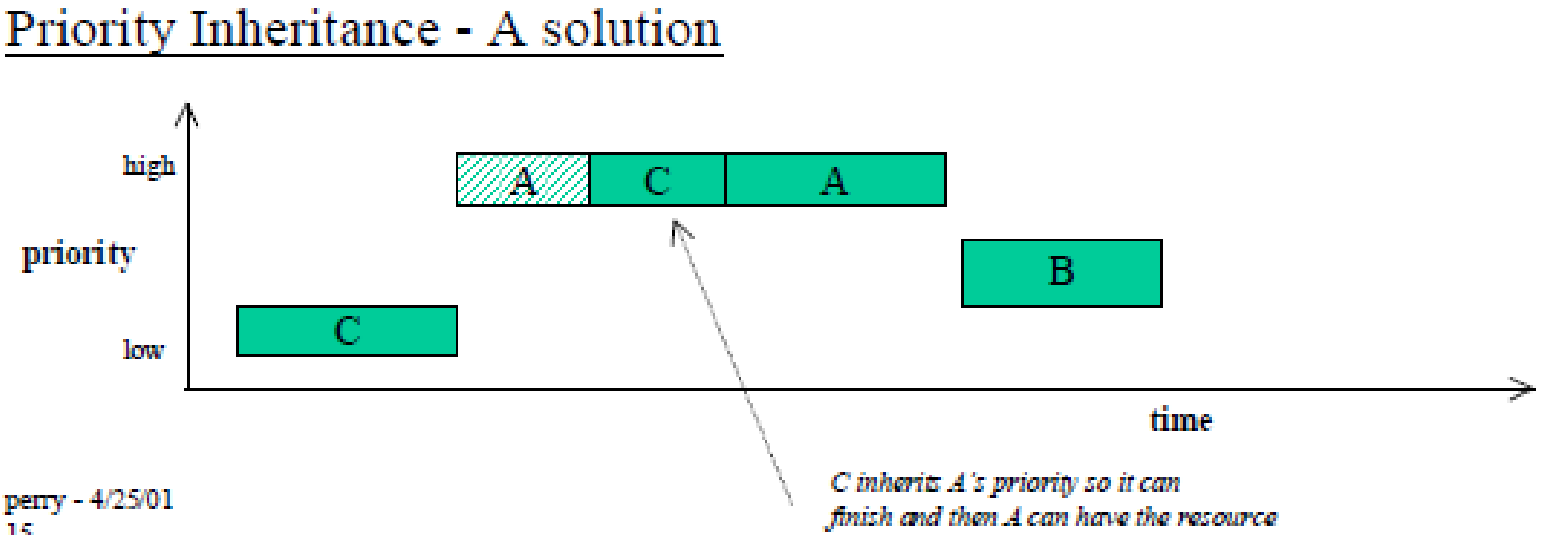
- How does a RTOS deal with some of these issues?
  - Rate Monotonic Systems (higher execution frequency = higher priority)
  - Priority Inheritance

Priority Inversion / Priority Inheritance

- Task A and Task C share a resource
- Task A is high priority
- Task C is low priority
- Task A is blocked when Task C runs (effectively assigning A to C's priority, hence priority inversion)
- Task A will be blocked for longer, if Task B of medium priority comes along to keep Task C from finishing
- A good RTOS would sense this condition and temporarily promote Task C to the high priority of Task A (Priority Inheritance)

Priority Inversion / Priority Inheritance





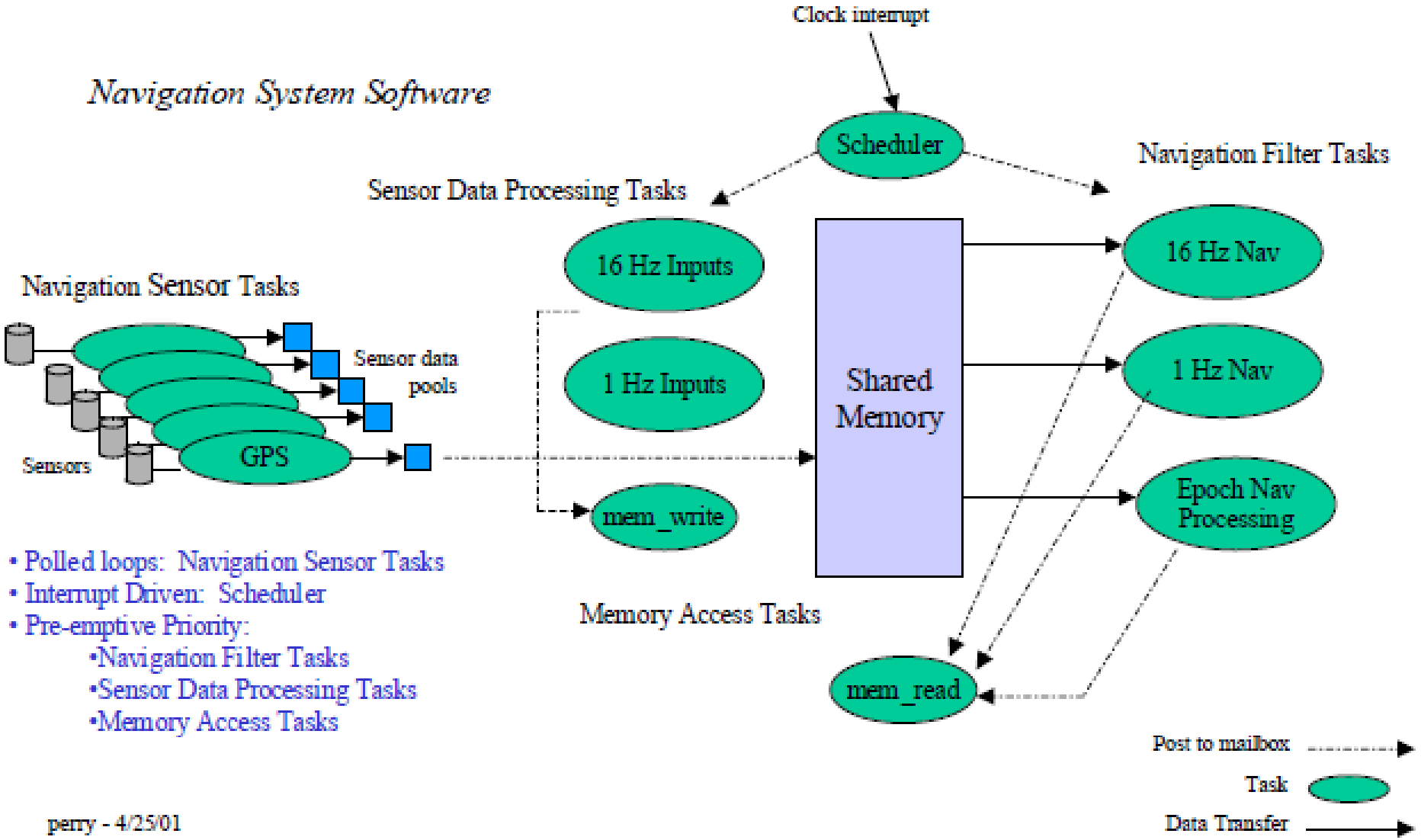
Foreground/Background Systems

- Most common hybrid solution for embedded applications
- Involve interrupt driven (foreground) AND noninterruptive driven (background) processes
- All realtime solutions are just a special case of foreground/background systems
  - Polled loops = background only system
  - Interrupt-only systems = foreground only system
- Anything not time-critical should be in background
  - Background is process with lowest priority

Foreground/Background Systems

- Gives hybrid systems = combining what we have seen so far
  - Polled loops
  - Interrupt-driven systems
  - Multi-tasking
    - Pre-emptive priority or
    - Round robin or
    - Cyclic executive

Back to the multitasking example



Multitasking Pros & Cons

- Pros
  - Segments the problem into small, manageable piece (modular computer system design principle)
  - Makes more modular software (can reuse portions more easily)
  - Allows software designer to prioritize certain tasks over others
- Cons
  - Depending upon implementation, timing may not be deterministic (jitter caused by variations in timing of incoming data)
  - Context switching adds overhead

Full Featured RTOS

- Expand foreground/background solution
  - Add network interfaces
  - Add device drivers
  - Add complex debugging tools
- Most common choice for complex systems
- Many commercial operating systems available

Choosing a RTOS approach

- How do you know which one is right for your application?
- Look at what is driving your system (arrival pattern of data)
  - Irregular (known but varying sequence of intervals between events)
  - Bursty (arbitrary sequence with bound on number of events)
  - Bounded (minimum interarrival interval)
  - Bounded with average rate (unpredictable event times, but cluster around mean)
  - Unbounded (statistical prediction only)
- What is the critical I/O?
- Are there absolute hard deadlines?

Choosing a RTOS approach

How do you know which one is right for your application? Let's look at some real life choices.

- Reusable Launch Vehicle for satellites. Thrust Vector Control SW requires new attitude data every 40 msec or rocket becomes unstable.
  - *We chose cyclic executive.*

- Navigation and Control System for submarine. Interface to multiple sensors at multiple data rates. Information from the Inertial Reference Unit is most critical, but exact timing of input data is not essential.
  - *We chose preemptive priority scheme running on a commercial RTOS. Important tasks given highest priority.*

Choosing a RTOS approach

How do you know which one is right for your application? Let's look at some real life choices.

- Avionics System requires new data from flight control surfaces, navigation equipment, and radar system every 50 msec.
  - *Cyclic executive. Each task runs to completion. Tasks run in series. Last tasks may not finish before 50msec interrupt occurs.*

- Microcontroller running to switch radar antennae and check for incoming signal. If the signal is there, power up the signal processing chip.
  - *We chose polled loop.*