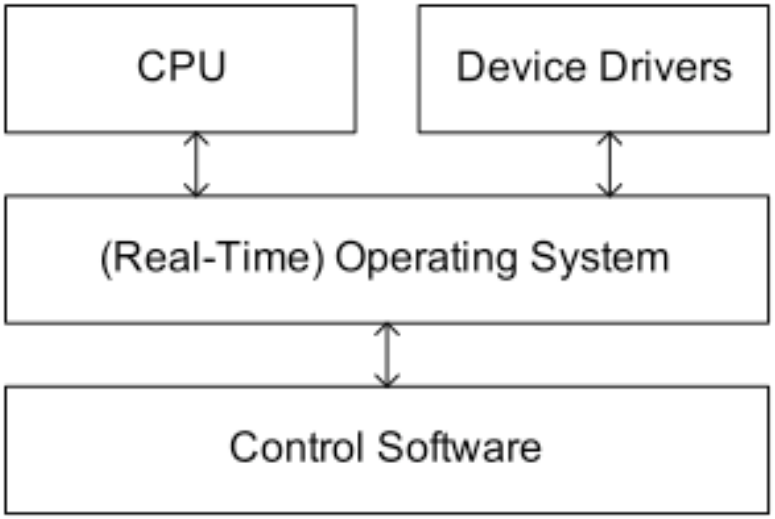
POSIX and Operating Systems

Introduction to real-time operating systems
POSIX standards

POSIX and operating systems



Operating System:

- **Controls access to the processor (schedules user tasks on the CPU)**
- **Controls access to the device drivers**

There are special real-time operating systems available

Do we need an operating system?

- Well, not really
  - We can write our code as one big task, and think about the scheduling ourselves
  - Need to write low level device driver routines
  - For really memory-constrained systems, perhaps this is possible

- But, a real-time operating system will make your life (as a systems architect or developer) much easier

Can we use a regular operating system?

- Regular operating systems were not designed with real-time in mind.
- Regular Operating Systems: on average, make things fast
- Real-Time Operating Systems: do everything in a bounded time
- Note: Fast does not necessarily equal Real-Time
- However, for some applications, a regular O/S might be fine

Regular OS vs. RTOS

- Regular operating systems
  - Process/task management
  - Memory management
  - Interrupt handling
  - Exception handling
  - Process/task synchronization
  - CPU scheduling
  - Disk management

Regular OS vs. RTOS

- Real-time operating system
  - Process/task management
    - Tasks might have deadlines (hard or soft)
  - Memory management
    - Not likely to use virtual memory (why?)
  - Interrupt handling (very important)
  - Exception handling
  - Process/task synchronization
    - Avoid priority inversion (we'll talk about this later)
  - CPU scheduling (of course!)
- We will talk about all of these in this course

Interrupt handling and its importance

- Interrupt latency
  - Time interval that begins when an interrupt signal reaches the processor to the start of the interrupt service routine (ISR)
- When an interrupt occurs
  - Processor must finish executing the current instruction
  - Recognize the interrupt type (done in hardware)
  - If interrupts are enabled, save the current system state (context) and start the ISR Context switch overhead
- Conventional operating systems disable interrupts during a system call
  - In a real-time operating system, this may not be desirable
  - It should be possible to preempt the kernel

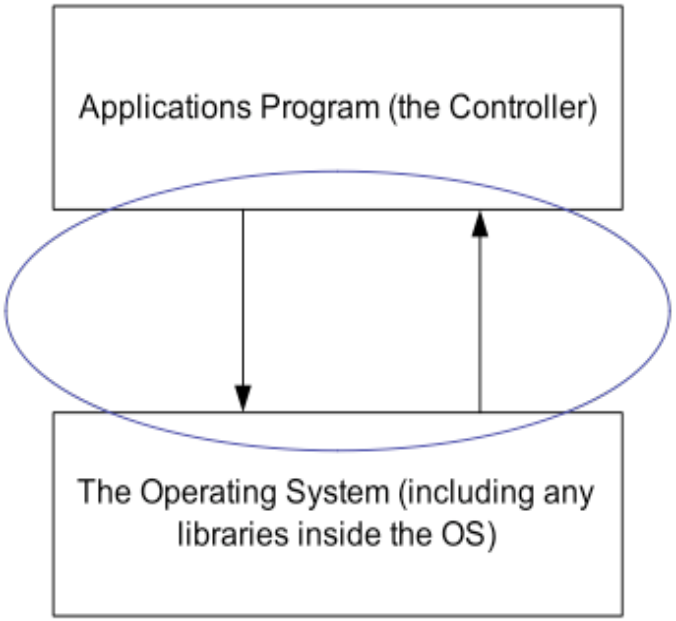
Some Popular RTOSs

- Wind River VxWorks
- Integrated System's pSOS
- Microtec VRTX
- QNX
- Real-time Linux
- and some more…
- The choice of platform may dictate a RTOS and vice-versa
    - Need to think about both when you are choosing the platform

What is POSIX?

Not an operating system! [Portable Operating System Interface for Unix]

Defines the interface between portable applications programs and the operating system



POSIX Defines this interface

Linux and Unix typically support POSIX

Windows NT kernel is POSIX-compliant

So are VxWorks, Solaris, HP-UX

This isn't a course on POSIX, but it is always useful to be able to illustrate concepts with something concrete

What is POSIX?

- Written as a set of IEEE Standards
- The first was POSIX.1:
- - Specified some of the basic calls that UNIX programmers use
- Then came POSIX.4 (now called P1003.1b)
  - Calls needed to implement real-time systems
  - (A)Synch I/O, shared memory, memory queues, timers, semaphores, signals, memory locking, etc.
  - Note that just because the calls are there, doesn't mean that the operating system implements them in a real-time-way!
- Then came P1003.1c
  - Calls needed to support threading

There are lots of other parts to POSIX:

- P1003.1a: Miscellaneous
- P1003.1b: Real-Time Extensions
- P1003.1c: Threads
- P1003.1d: More R/T Extensions
- P1003.1e: Security Enhancements
- P1003.1f: Transparent File Access
- P1003.1g: Network Services
- P1003.1h: Distributed Systems

- We will not care about most of these in this course.

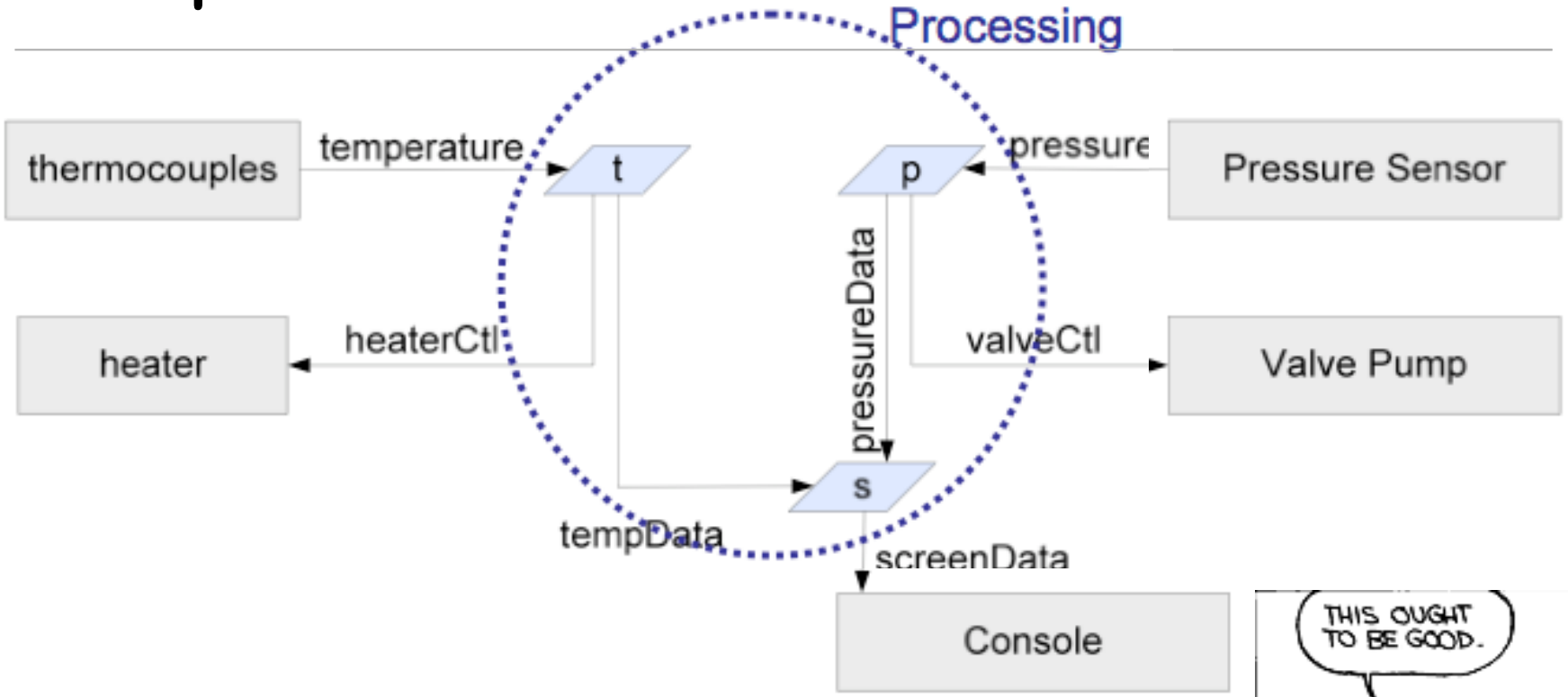
Regular OSs vs. RTOSs; POSIX

Regular Operating Systems: on average, make things fast

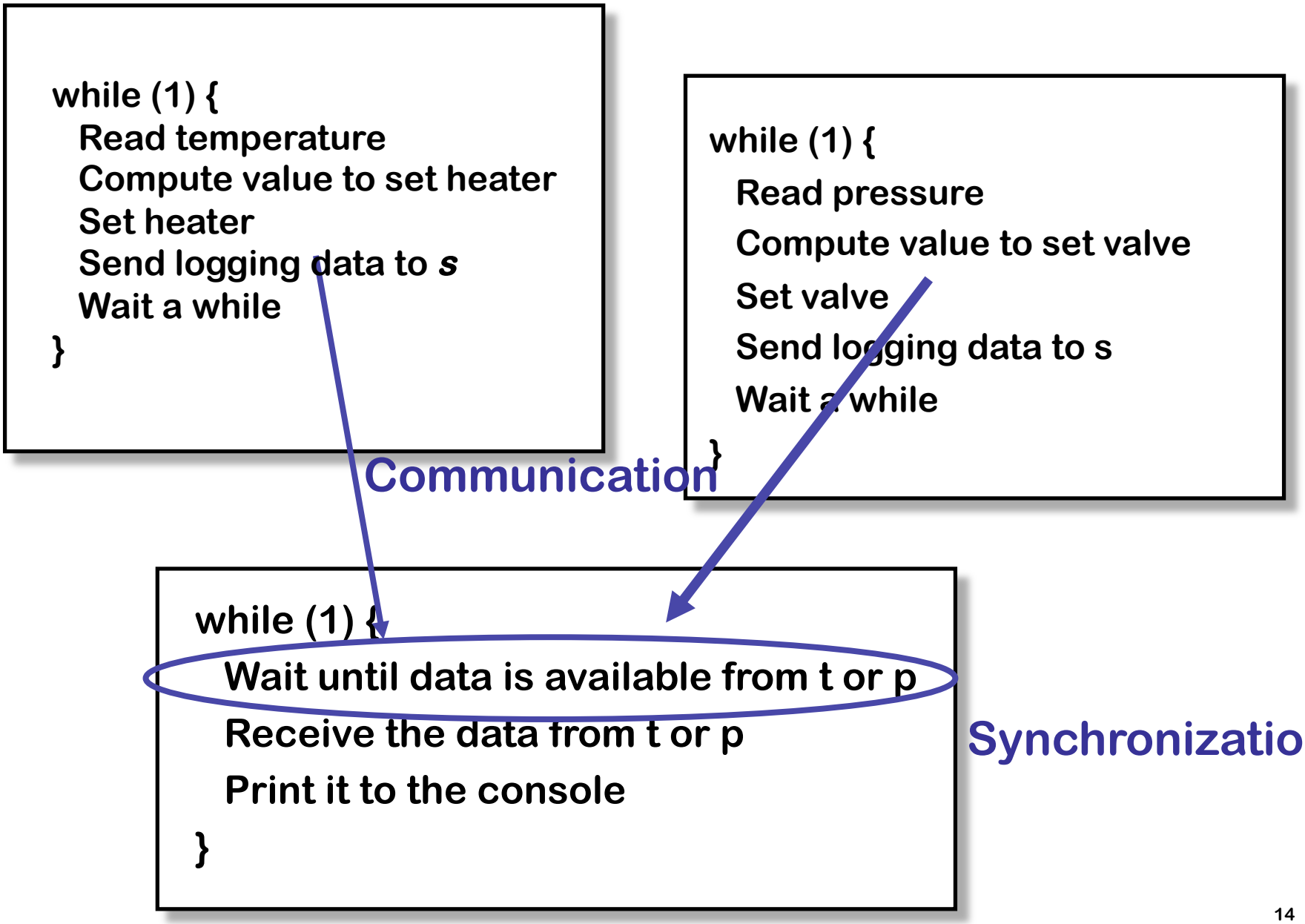
Real-Time Operating Systems: do everything in a bounded time

POSIX defines standard interfaces for functionality that is required to implement real-time systems (and more)

Example



We could write separate threads/processes for *t*, *p*, and *s*.


while (1) {
Read temperature
Compute value to set heater
Set heater
Send logging data to s
Wait a while
}
while (1) {
Read pressure
Compute value to set valve
Set valve
Send logging data to s
Wait a while
}
Communication
while (1) {
Wait until data is available from t or p
Receive the data from t or p
Print it to the console
}
Synchronization

Support for multiple processes/threads

- A real-time OS needs to provide:
- An API that allows the user to:
  - Create and kill processes and threads
  - Communicate between processes and threads
- A method to time-slice processes and threads on one or more CPUs such that:
  - Hard Deadlines are obeyed (first priority)
  - Soft Deadlines are obeyed (second priority)
  - Deadlocks do not occur (another first priority)
    - Scheduling policies that we have studied so far (at least in part)

Highlights of the lecture

- Real-time operating systems need to provide extra (different?) functionality when compared to a regular OS
- Interfaces are defined in the POSIX standard
  - But the standards, by themselves, do not guarantee real-time behavior
- POSIX primitives - when implemented correctly - will allow developers to build a real-time application