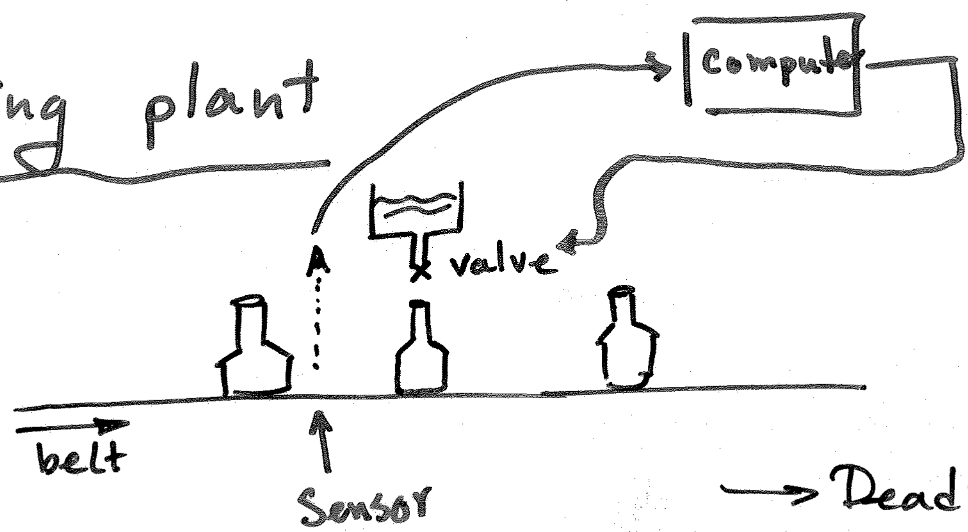
MULTIPLE TASKS AND CONCURRENCY

- Real-time applications: multiple tasks
- Multitasking: single computer sharing CPU time with other tasks
- Implementation:
  — cyclic executive: simple systems
  — real-time multi-tasking kernel: demanding applications
- Multi-tasking kernels allow:
  — assigning priorities to tasks
  — tools for keeping track of tasks' deadlines
  — dealing with events
- Two types of events
  — synchronous: occur at predictable times (e.g., control loop)
  — asynchronous: unpredictable times (e.g., emergency stop button)

Events and deadlines : Examples

* **Bottling plant**



→ Deadline ?

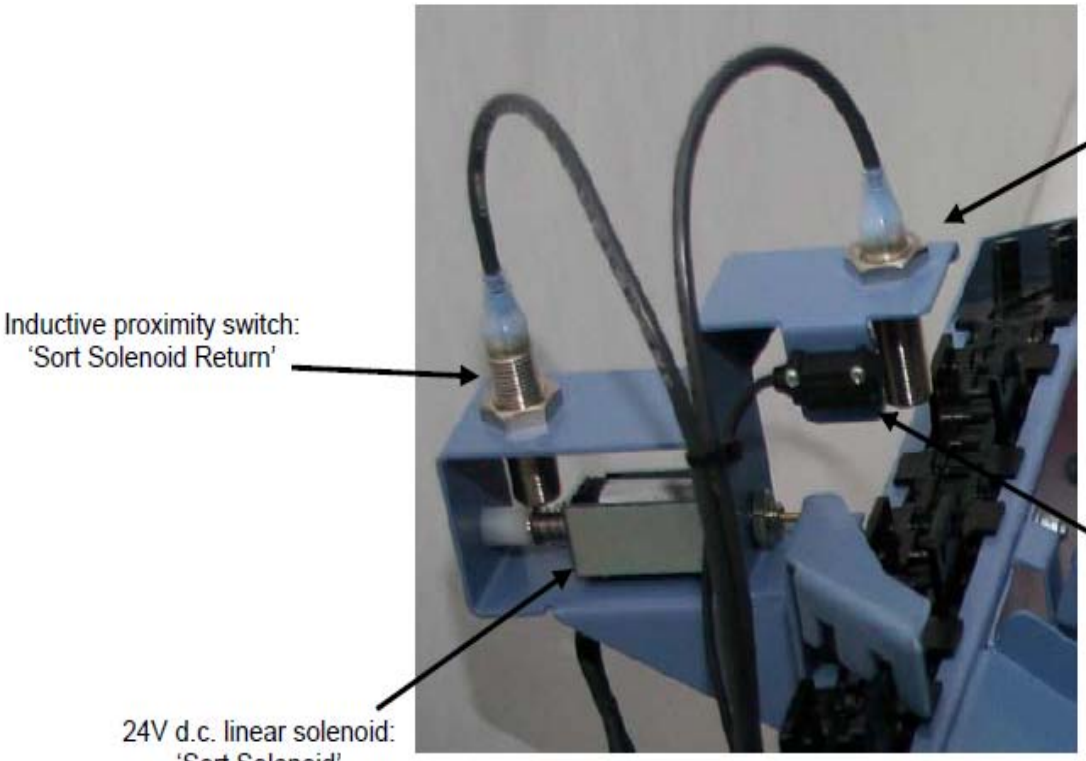
* **Multimedia** → How many video frames should be presented to a user per second ?

* **Taking courses** → Is this a RTS ?

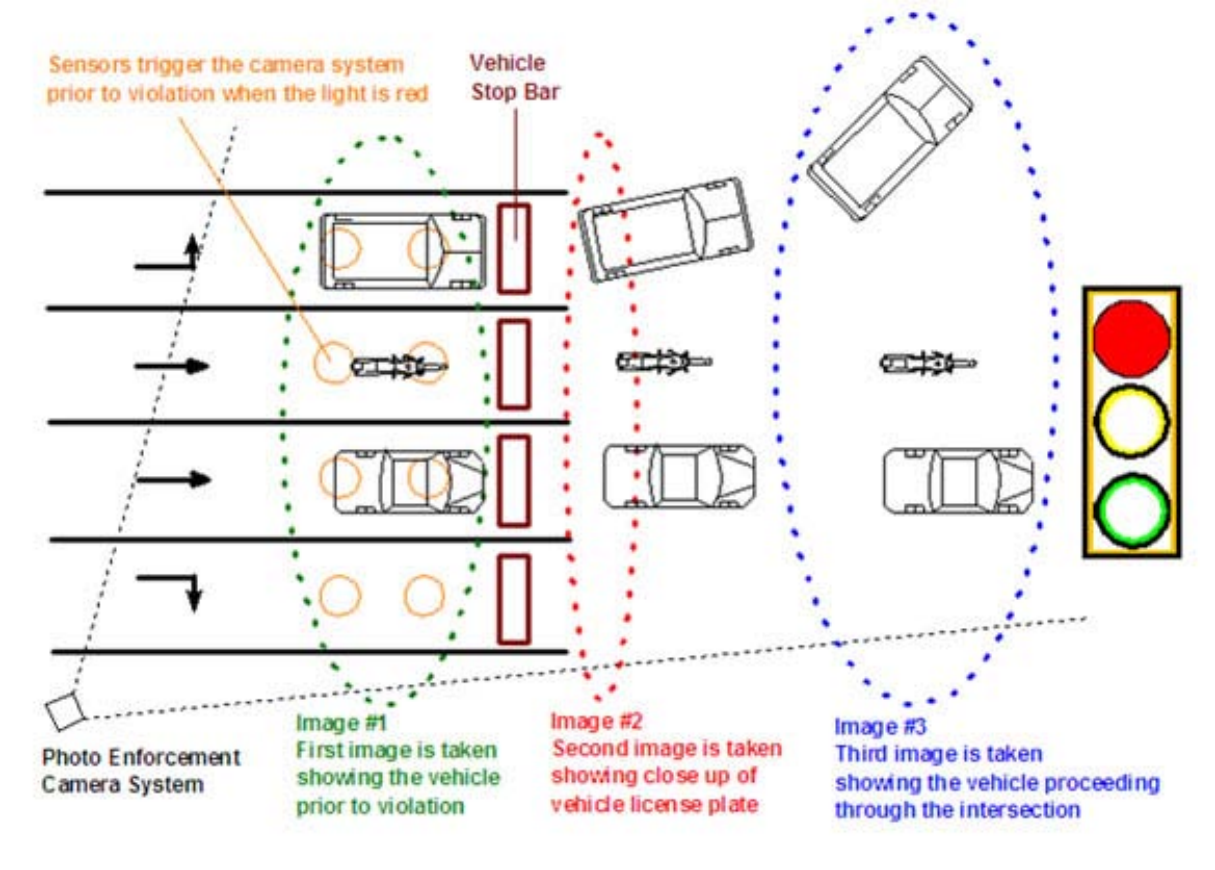
Spectrum of RT applications :

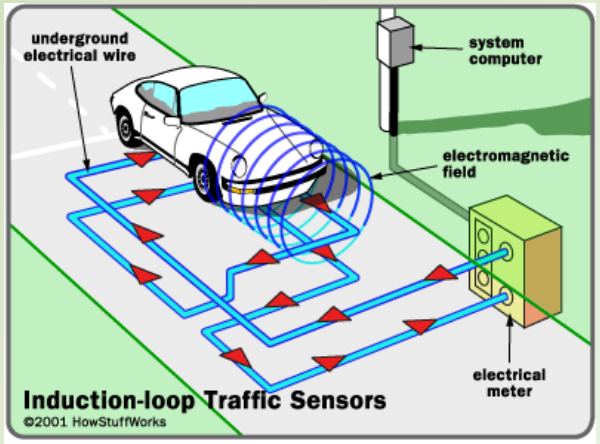
Non real-time   Soft RT   Hard RT

Events and deadlines in an Industrial Control System: Sorting Station



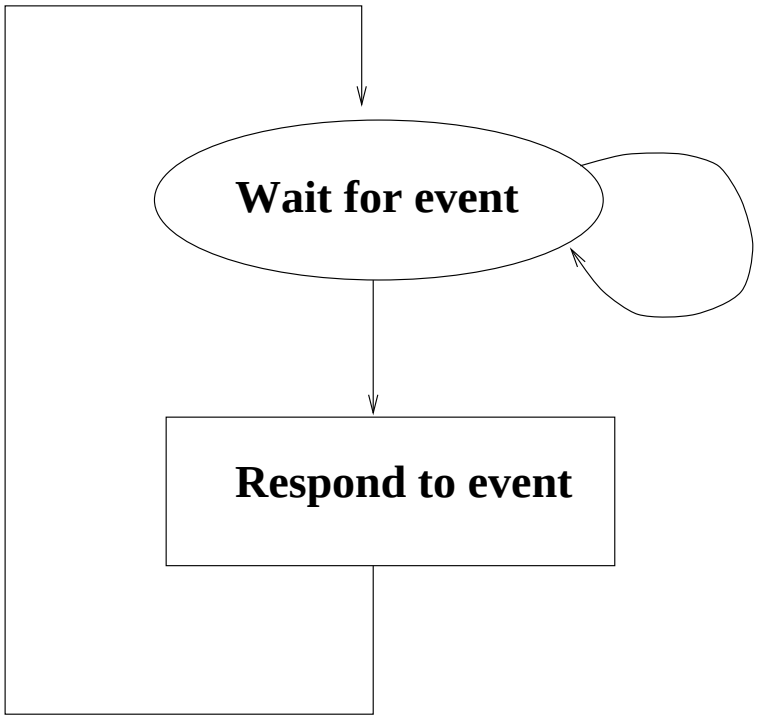
Events/Deadlines in Traffic Light Camera





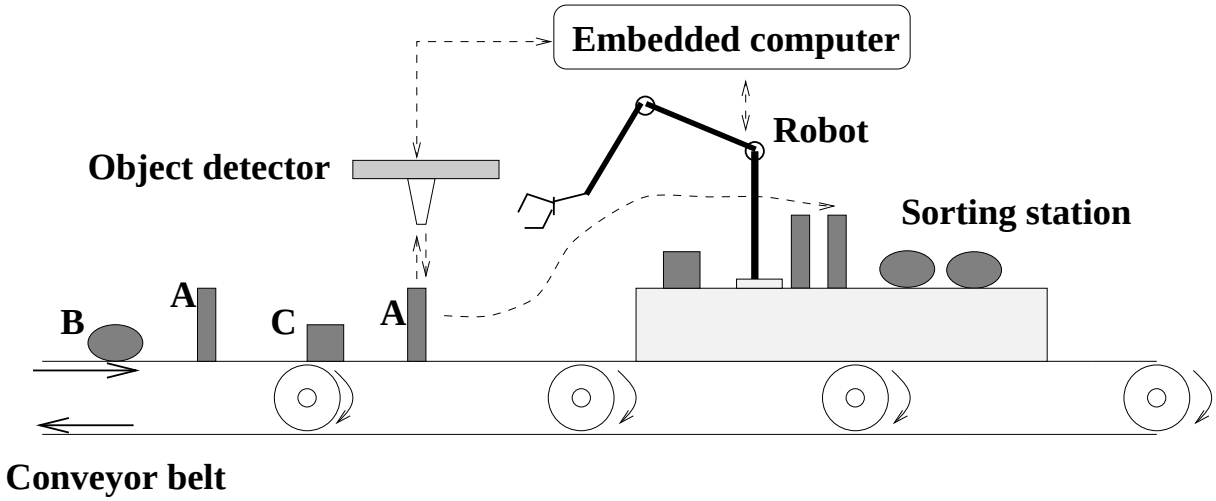
The Need for Real-Time Executives

- Program to check a status register bit indicating the occurrence of an external event.



- Real-time systems are rarely as simple as this case.

A robotic assembly system



- The robot has to pick up the object and place it at a sorting station
- System components: object detector, robot, conveyor belt, computer

Consider the following cases:
Events occur in a pre-specified cyclic pattern:
e.g., A, B, C, A, B, C, ...

```
while (TRUE){
        while (TRUE){
                if (isEventA( )){
                    respond2A( );
                    break;
                    }
            }
        while (TRUE){
                if (isEventB( )){
                respond2B( );
                break;
                }
            }
        while (TRUE){
                if (isEventC( )){
                respond2C( );
                break;
                }
            }
   }
```

— would not work in general, e.g., what would hapen if `ACAB ...` arrives?

A simple solution:

```
while (TRUE){
      if ( isEventA() == TRUE ) respond2A( );
      if ( isEventB() == TRUE ) respond2B( );
      if ( isEventC() == TRUE ) respond2C( );
}
```

Timing considerations

Assume:

— if (isEventX() == TRUE), (X $\in$ {A,B,C }) takes 0.1 sec

— respond2A( ) takes 4 sec

— respond2B( ) takes 3.5 sec

— respond2C( ) takes 3 sec

— while(TRUE) has negligible overhead

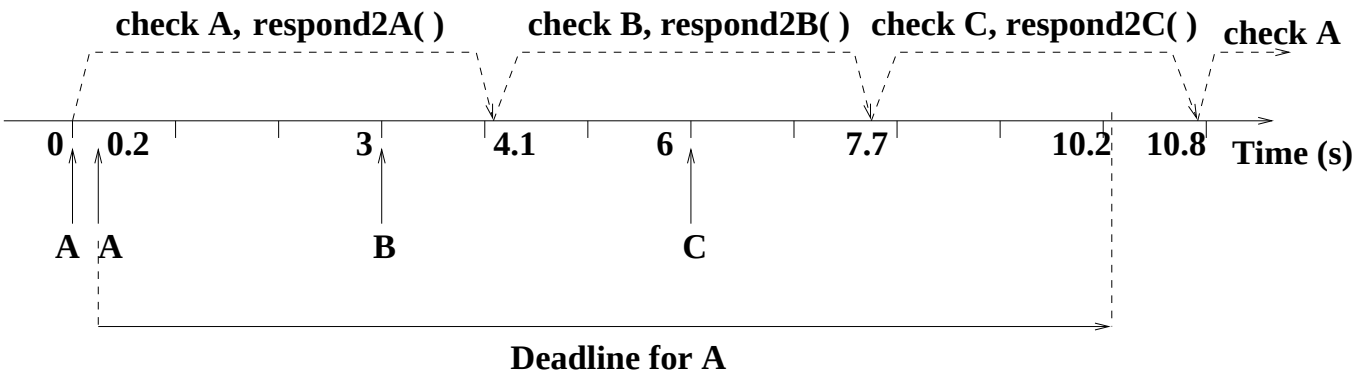
Assume: Minimum inter-arrival time of A is 0.2 seconds.

The total loop time:

$$0.3 = 0.1+0.1+0.1 \leq T_l \leq 0.1+4+0.1+3.5+0.1+3 = 10.8. \quad (1)$$

— Deadlines: A (10 seconds), B and C (15 seconds each)

— Does the system meet its deadlines?



One way to solve this problem: Check event A more frequently

```
while (TRUE){
      if ( isEventA() == TRUE ) respond2A( );
      if ( isEventB() == TRUE ) respond2B( );
      if ( isEventA() == TRUE ) respond2A( );
      if ( isEventC() == TRUE ) respond2C( );
  }
```

— time that A has to wait to be served: between 7.2 and 7.7 s (A meets its deadline)
— loop time

14.9s

$$0.4 = 0.1+0.1+0.1+0.1 \leq T_l \leq 0.1+4+0.1+3.5+0.1+4+0.1+3 = \quad (2)$$

— 14.9 is close to B and C deadlines

If B had a deadline of 13 sec

— have to modify loop using more conditionals and calls to routines

— e.g.,

```
while (TRUE){
   if ( isEventC() == TRUE ) respond2C( );
   if ( isEventA() == TRUE ){
          respond2A( );
          if ( isEventB() == TRUE ){
                    respond2B( );
                    if (isEventA()==TRUE)
                              respond2A();
                }
          continue; /* go to start of while */
     }
 if ( isEventB() == TRUE ) respond2B( );
}
```

As the system becomes more complex:

— more control code

— code becomes more difficult to understand, debug, and restructure

Conclusion: Sequential programming is not adequate for real-time systems

Solution: Multi-tasking

- Dedicate one processor to each event, implement three simple programs using while loops
- Use a *multiplexer* which allows different programs to *share* a single processor
- A *multi-tasking executive* creates a number of *virtual processors* running independent tasks
  — allows modular construction of applications
  — maximizes CPU utilization

Concurrent Programming

- Programming techniques for expressing parallelism
  — must resolve: communication and synchronization problems
- Notion of processes or tasks

Concurrent programming may be implemented in one of the following ways:

- *Multiprogramming or Multitasking*
  — multiple execution on a single processor
  — processor is shared among tasks

- *Multiprocessing*

  — processes multiplex their execution on a multiprocessor system

  — processors share memory

- *Distributed Processing*

  —- execution on several processors not sharing memory (e.g., network)

Concurrent programming should provide facilities for

1. *Creation of separate tasks*

   — may be independent, cooperating, or competing

2. *Synchronization of tasks*

   — e.g., data acquisition task should synchronize its activity with the control algorithm task

3. *Interprocess communication*

4. *Interfacing with special purpose hardware*

5. *Guaranteed response times*

   — means more than *fast* response time although being fast can help real-time performance.

6. *Extreme reliability*
   — redundant processors (Boeing 777)
7. *Efficient implementation*
   — program speed
   — data and program size
   — power consumption.

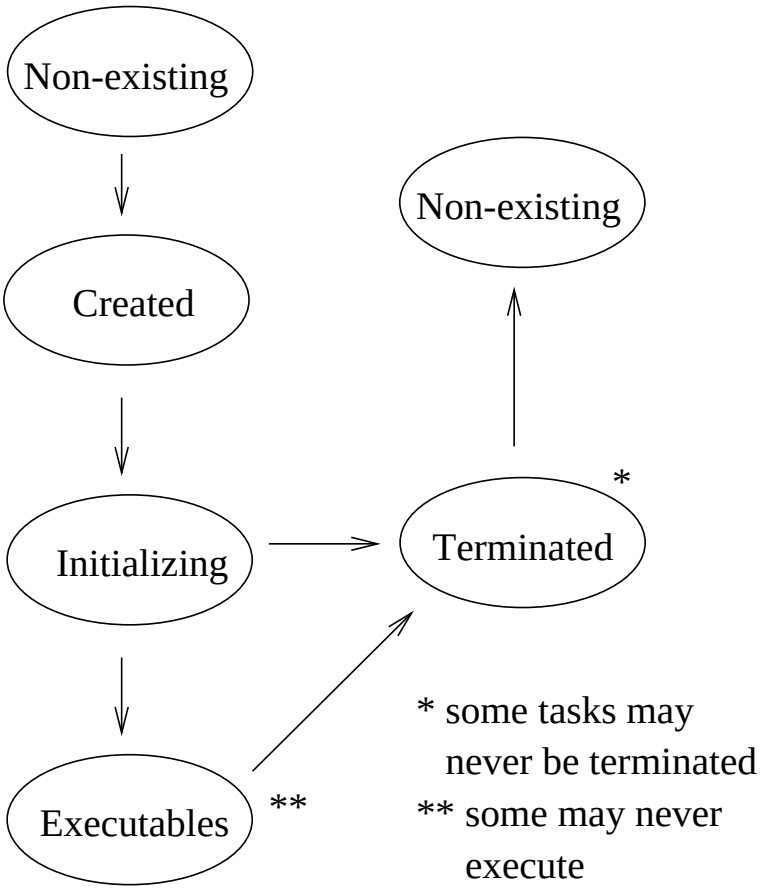
Processes, Threads, and Tasks

- A process is a passive object containing threads
  — similar to a complete application
  — e.g., display driver, data acquisition, controller,
  — a word processor, a database, and a spreadsheet operate as three different processes in three different windows.
  — allow for breaking systems into modular and decoupled units
- Threads are active objects within a process occupying memory and using the CPU
  — within a process, if one of the threads allocates memory, all other threads have access to it

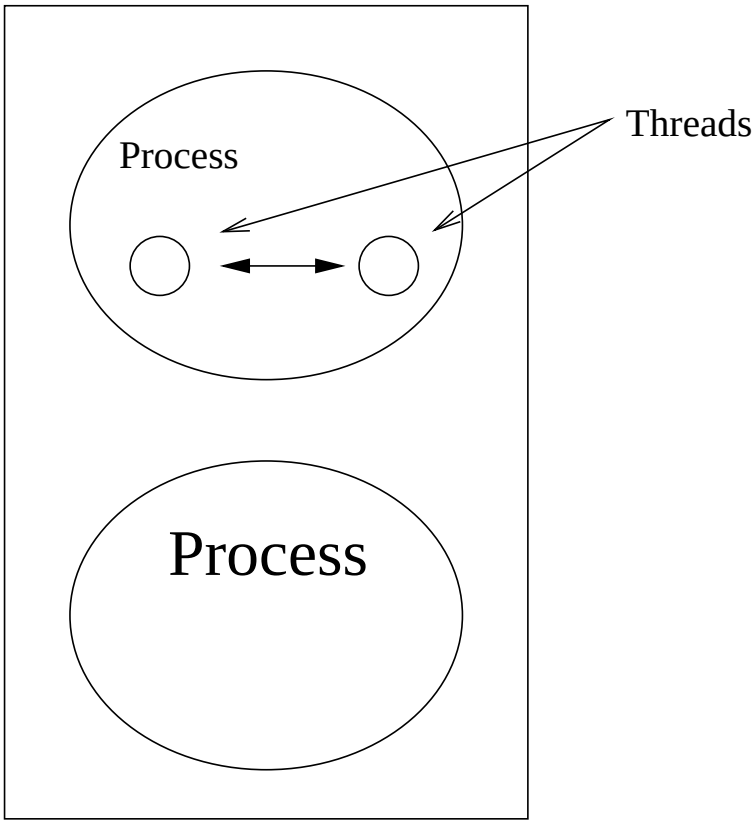
- *Task*: A sequence of instructions composing an entity that occupies as well as allocates memory and uses the CPU.

  — In a single process system the threads may be called tasks

Task states in a real time system:



Relationships among processes and threads

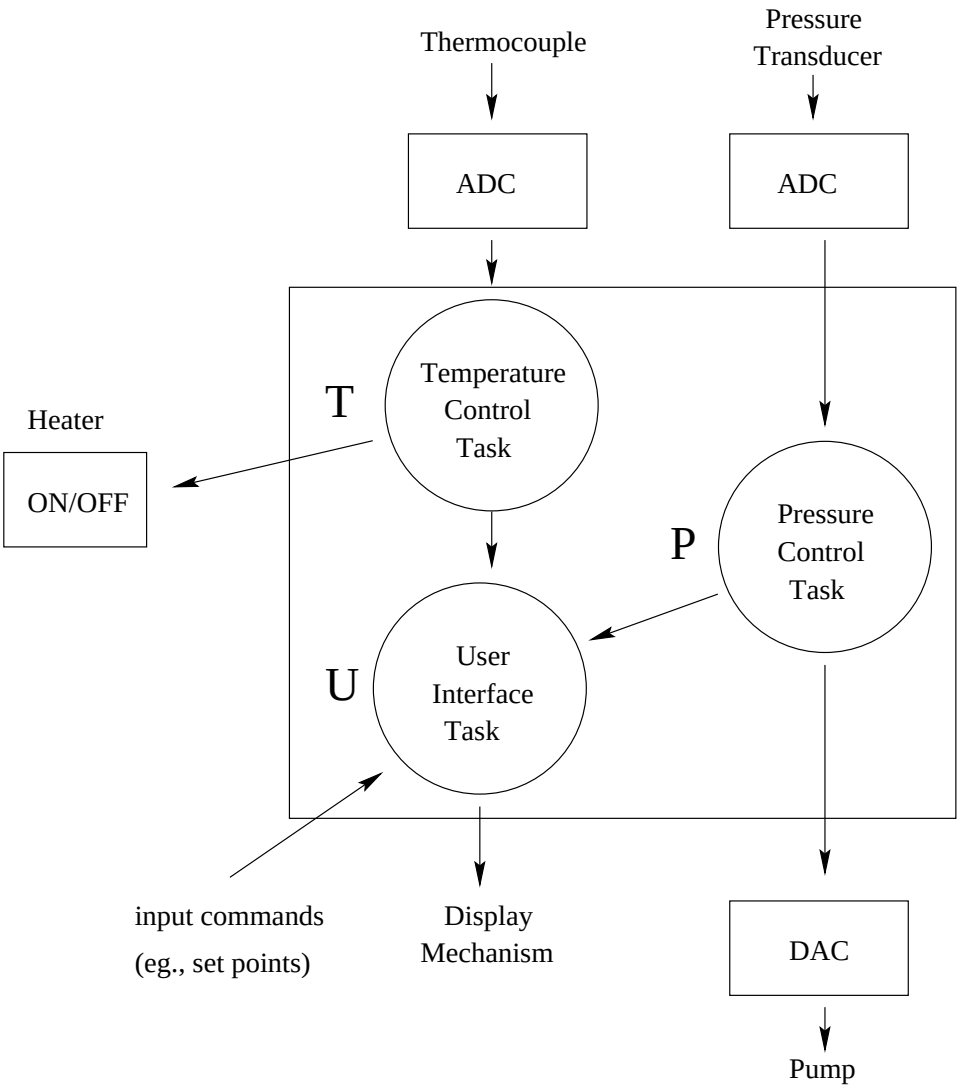


For real-time systems, threads are generally the best choice

- Threads are generally faster
- Threads are more memory efficient
- A thread–based kernel is smaller

CONCURRENT PROGRAMMING

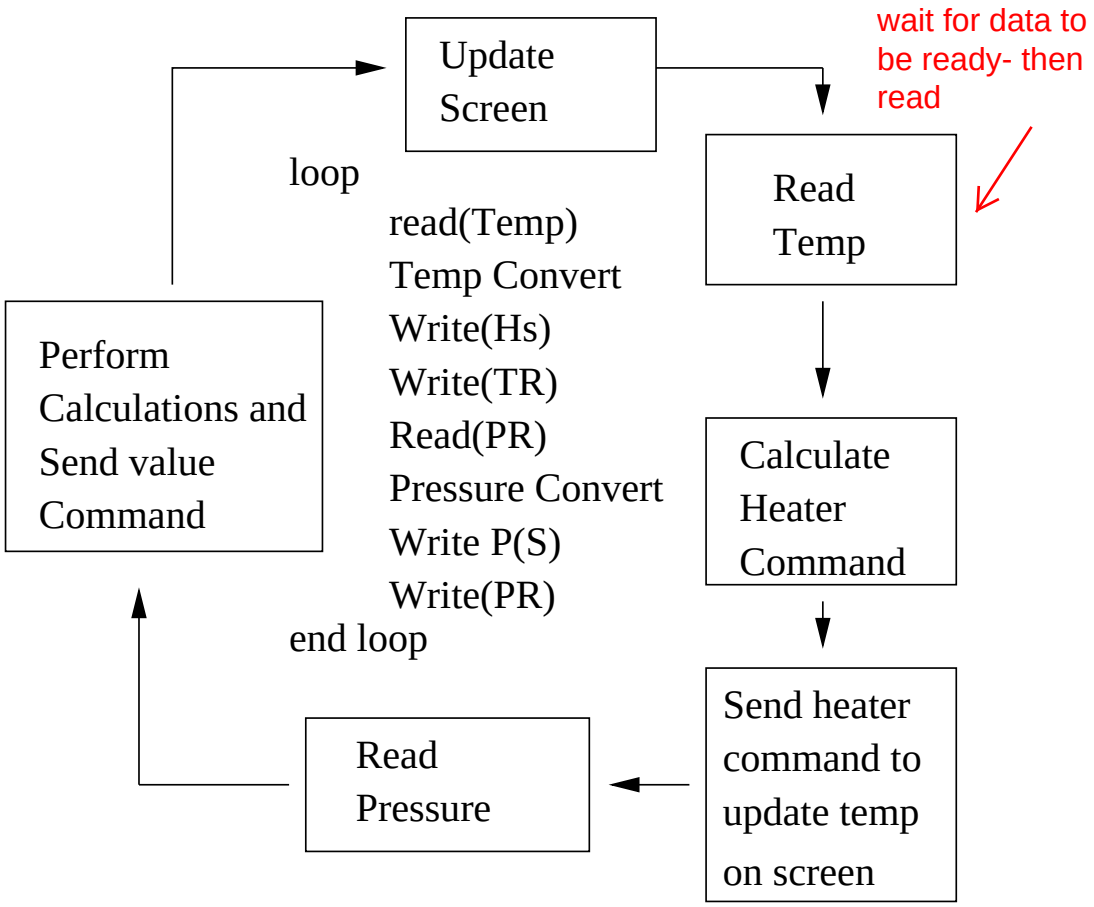
Concurrent Programming: Regulate T & P, update display



Possible software architectures:

1. Sequential solution– ignore concurrency
2. Sequential programming language & a multitasking OS.
3. Use of a real-time language (such as ADA)– concurrency recognized in the code itself

Sequential Solution



1. Temperature and pressure readings taken at the same frequency

   — use of a counter and if statements could improve the situation

   — complicates the code readability/maintainability

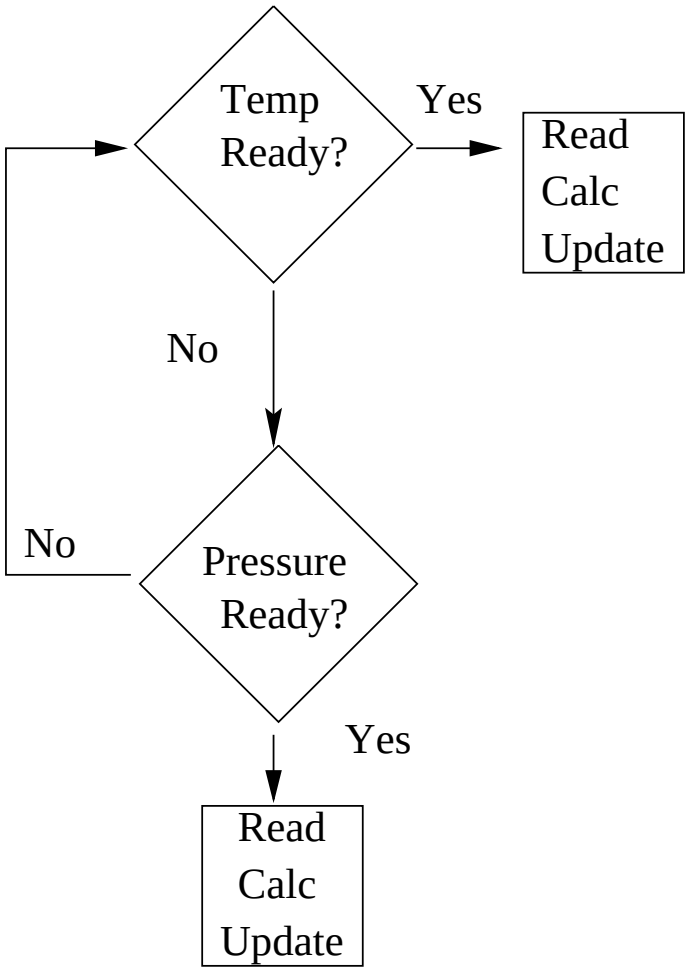
2. While handling temperature, no attention given to the pressure task.

   — failure in temperature control (e.g., no return) can fail pressure control

Improved Buffer Solution

The system *polls* two digital ports to see if they are ready
→ inefficient *busy waiting*

```
If Ready_temp then
        Read(TR);
        Convert(TR_HS)
        Write(HS);
        Write(TR);
end if
if Ready_Pressure then
        Read (PR);
        ~~~~~~~
end if
```



Using Operating System Primitives

Three separate entities:

- Temperature control task
- Pressure control task
- User interface task

Can be implemented using 3 infinite while loops sharing one processor

— using a concurrent language (ADA), or

— using sequential programming and an operating system interface (OSI) package (C/VxWorks)

```
void temp(){
     /* tr: temperature reading */
     tr=readTemp( );
     /* hs: heater setting */
     getHeaterCommand(tr, hs);
     writeHeater(hs);
     write(tr);
    }

void pressure(){
     /* pr: pressure reading */
     pr=readPressure( );
     /* ps: pressure setting */
     getPumpCommand(pr, ps);
     writePump(ps);
     write(pr);
    }

void userInterface( ){
     /*
     process input commands and
     display temperature and pressure.
     */
     ....
```

```
    }

....

  runTask(tempTask, temp, frequency1 ...);


  runTask(pressureTask, pressure,
                        frequency2, ...);


  runTask(userInterfaceTask, userInterface,
                        frequency3, ...);
....
```