ECE 1175
Embedded Systems Design

Final Review

Wei Gao

Final Exam

- **When**: 4/26 Mon 8:20AM – 9:50AM
  - Students with disabilities will have 50% extra time
- 25% of your final grade
- **What about**: Everything covered in this course
  - Closed-book, closed-notes, no laptop, no discussion
  - All included in class slides (as well as textbook, assigned papers)
    - Distribution: 40% on the first half of semester, 60% on the second half of semester
  - Will include some programming questions
  - More questions than the midterm exam
  - Make your answers short to include only key points
    - Do answer every question, DON'T leave blank

Logistics of Online Exam

- Online proctoring via the lecture zoom link
  - Everyone with your video on throughout the exam
  - Make sure that we can see not only your face, but also your hands
- Exam procedure
  - The exam sheet will be distributed online via zoom right before the exam starts
  - You will need to write your answers on paper and scan your answers back
    - Recommended software: Adobe scan via smartphone
    - Make sure that you settle a scanning method and get it ready before the exam!
  - After the exam
    - Stay online with video on
    - You will need to scan your answers and send them out right after the exam
    - You can only log off after your answers are received
    - You have to make sure your answers are received within 30 minutes after the exam

First half of the semester

- Introduction to embedded systems
  - Real-time, low power, small memory footprint, low cost
  - Soft/hard real-time system
- Design methodology
  - Microprocessors, FPGA and ASIC
  - Design procedure and example: GPS
- Microprocessors
  - von Neumann vs. Harvard, RISC vs. CISC, SHARC, ARM7
- CPUs, I/O, interrupt
  - Busy-wait I/O, interrupt-based I/O, interrupt mechanism
- Caches and memory
  - Memory system, average memory access time in multi-level cache
  - Cache organization: direct-mapped, N-way set-associative
- Real-time scheduling

Definition

- Embedded system: any device that includes a computer but is not itself a general-purpose computer.
- System characteristics
  - Non-functional requirements: Real-time, Low power, Small memory footprint, Low cost
  - Hard vs. soft real-time

Alternative Technology

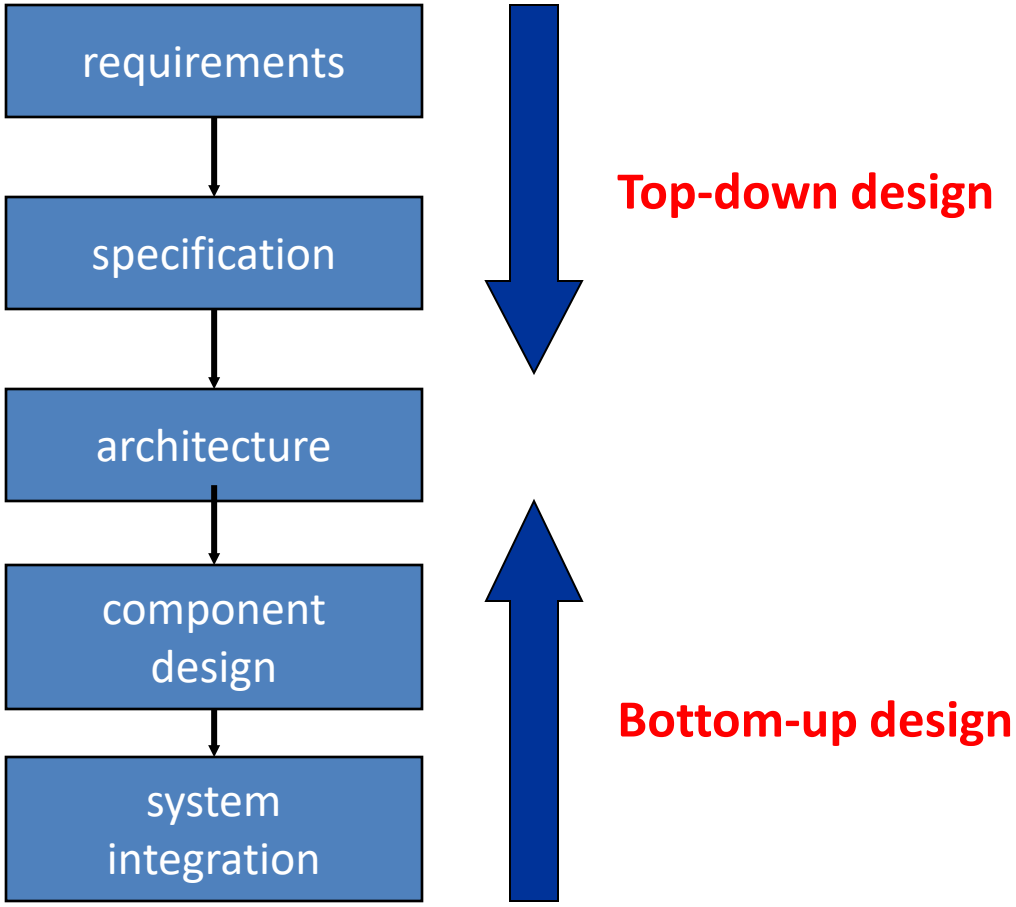
- Application-Specific Integrated Circuits (ASICs)
- Microprocessors
- Field-Programmable Gate Arrays (FPGAs)

- Why should we use microprocessors?
  - Reprogrammability and low development cost >> low performance/watt

Microprocessors

- Performance
  - Con: Programmable architecture is fundamentally slow!
    - Fetch, decode instructions
  - Pro: Highly optimized architecture and manufacturing
    - Pipelines; cache; clock frequency; circuit density; manufacturing technology
- Power
  - Processors perform poorly in terms of performance/watt!
  - Power management can alleviate the power problem.
- Flexibility, development cost and time
  - Let software do the work!

Design Methodologies



Microprocessors

- von Neumann
  - Same memory holds data, instructions.
  - A single set of address/data buses between CPU and memory
- Harvard
  - Separate memories for data and instructions.
  - Two sets of address/data buses between CPU and memory
- RISC vs. CISC
  - CISC: Many addressing modes and instructions; High code density.
  - RISC: Compact, uniform instructions: facilitate pipelining, poor memory footprint

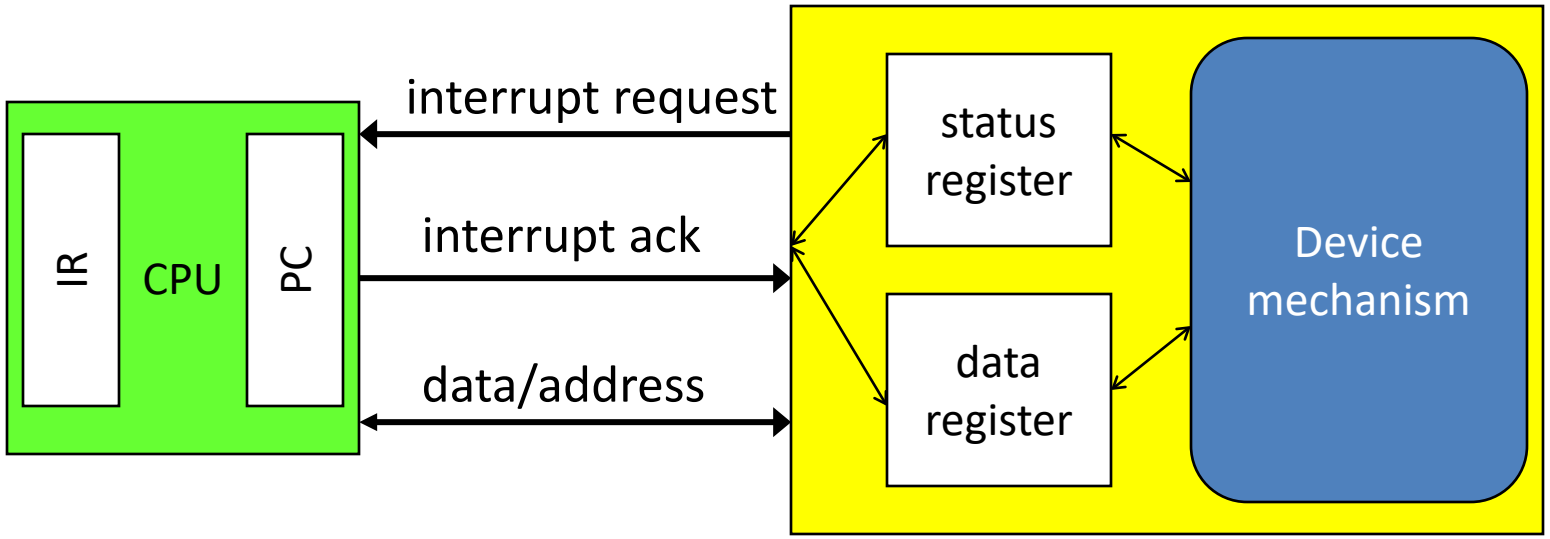
Busy-Wait I/O Programming

- Simplest way to program I/O devices.
  - Devices are usually slower than CPU and require more cycles
  - CPU has to wait for device to finish before starting next one
  - Use **`peek`** instruction to test when device is finished
  - **Test-and-set**

```
//send a string to device using Busy-Wait handshaking
current_char = mystring;
while (*current_char != '\0') {
   //send character to device (data register)
   poke(OUT_CHAR,*current_char);
   //wait for device to finish by checking its status
   while (peek(OUT_STATUS) != 0);
   //advance character pointer to next one
   current_char++;
}
```

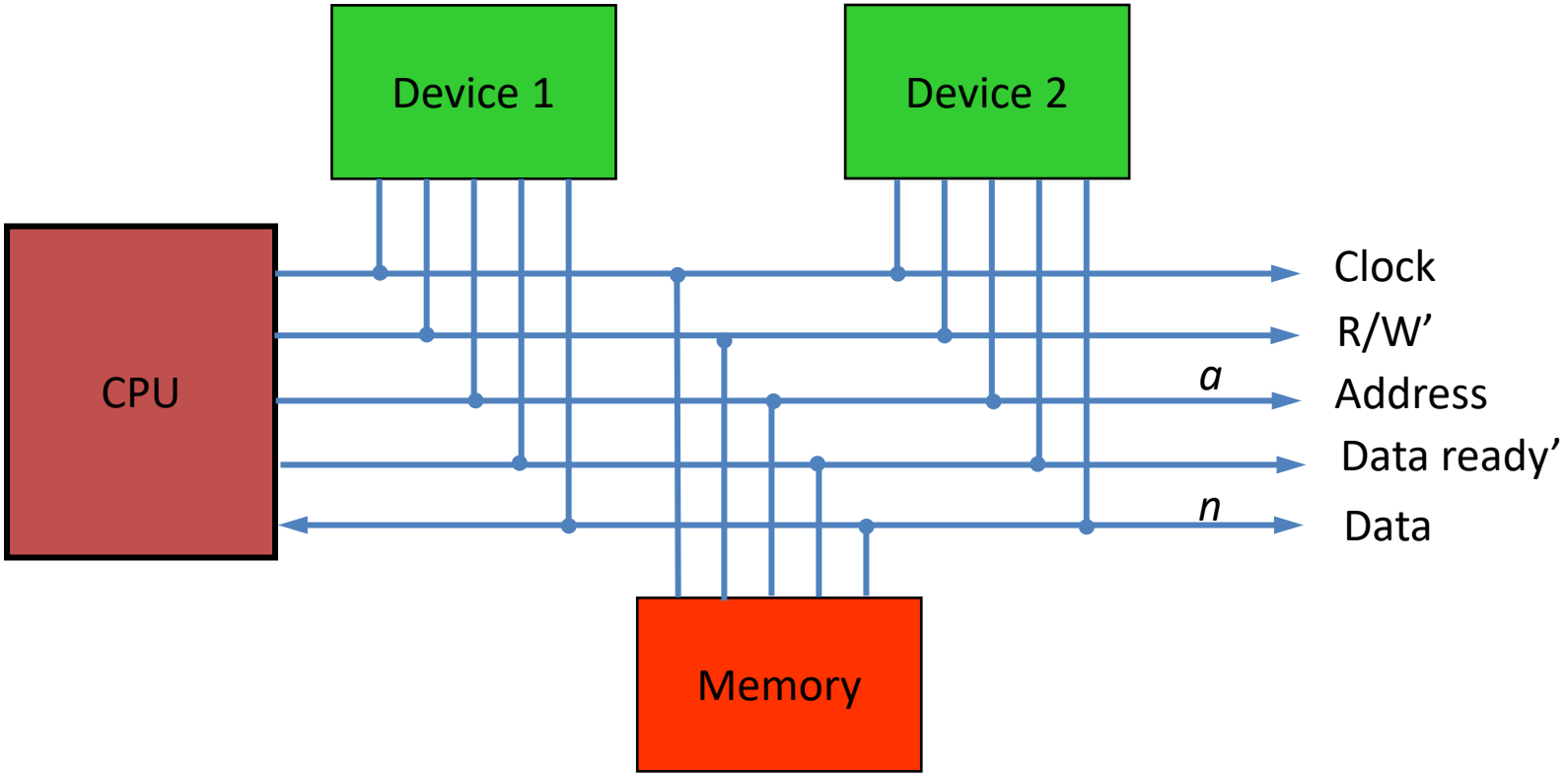
Interrupt-based I/O

- Busy-wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow to change the flow of control in the CPU.
  - Call interrupt handler (i.e. device driver) to handle device.



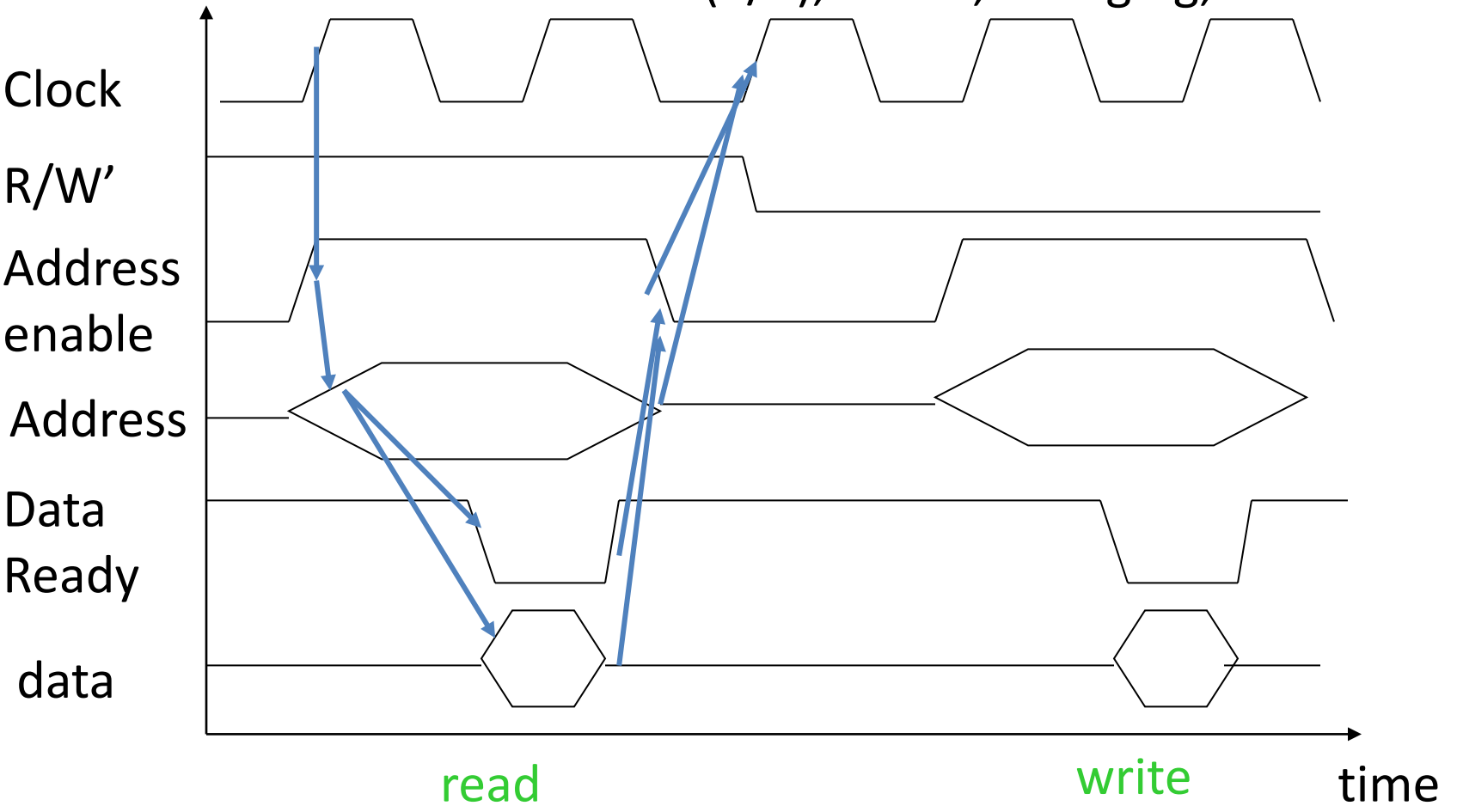
Microprocessor Bus

- Bus is a set of wires and a protocol for the CPU to communicate with memory and devices
- Five major components to support reads and writes



Typical Bus Access

- Timing diagram syntax:
  - Tri-state: Constant value (0/1), stable, changing, unknown.



I/O Interfaces

- Parallel v.s. Serial
  - Parallel
    - Wider bandwidth
    - More wires indicate more overhead
    - Simple I/O operation
  - Serial
    - 1-bit transfer per time unit
    - Less wires indicate less overhead
    - Complex I/O protocol
- Serial I/O interfaces
  - SPI
  - I2C
  - Master/slave mode

ARM Architecture

- Dominant architecture for embedded systems

ARM Instruction Sets

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
 CMP    r3,#0
  BEQ    skip
  ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
 ADDNE r0,r1,r2
```

Memory System and Caches

- Memory is slower than CPU
  - CPU clock rates increase faster than memory
- Caches are used to speed up memory
  - Cache is a small but fast memory that holds copies of the contents of main memory
  - More expensive than main memory, but faster
- Memory Management Units (MMU)
  - Memory size is not large enough for all application?
  - Provide a larger virtual memory than physical memory

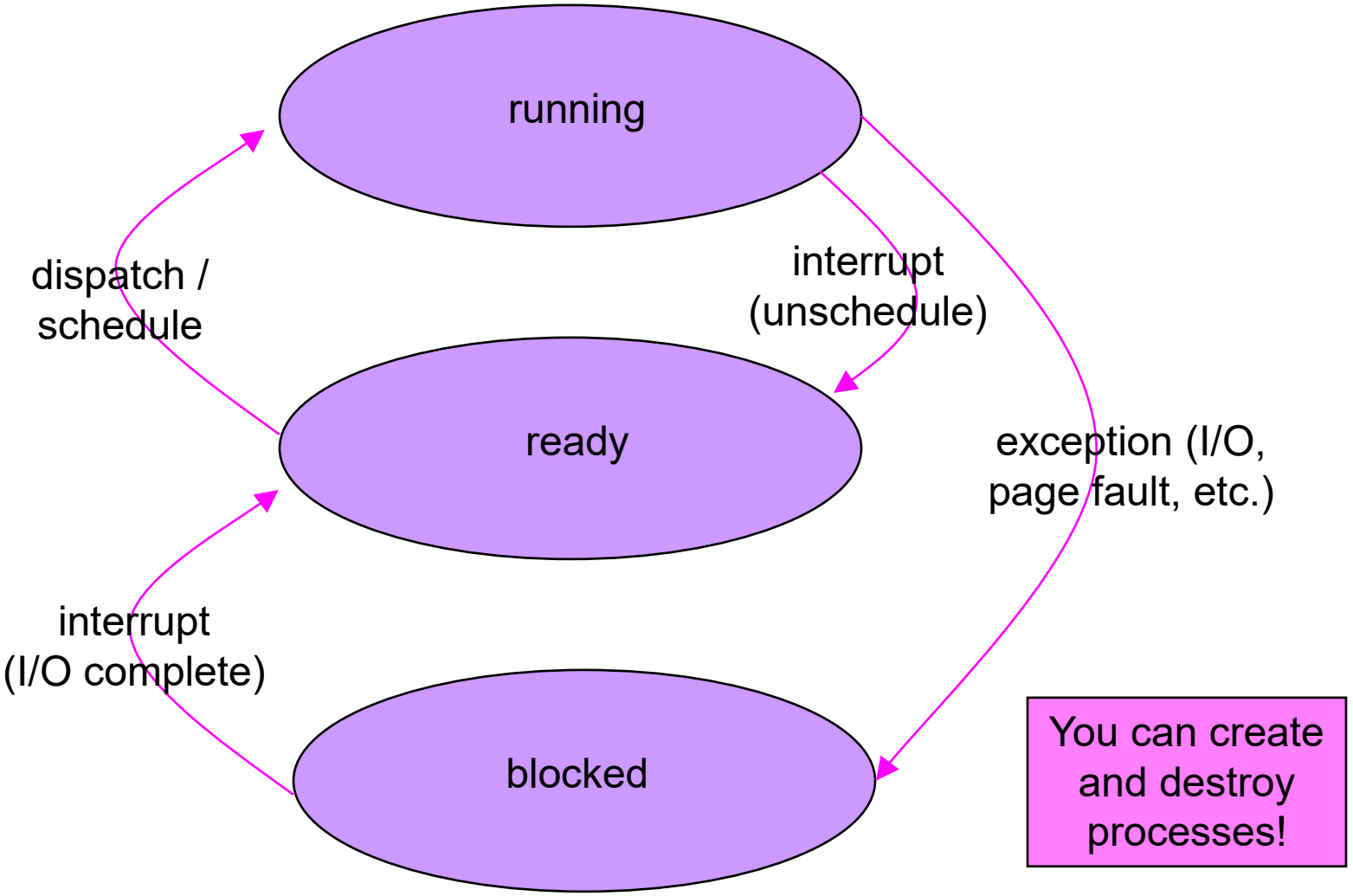
Memory Devices

- Types of memory devices
  - RAM (Random-Access Memory)
    - Address can be read in any order, unlike magnetic disk/tape
    - Usually used for data storage
    - DRAM vs. SRAM.
  - ROM (Read-Only Memory)
    - Usually used or program storage
    - Mask-programmed vs. field-programmable.

Key OS Issues

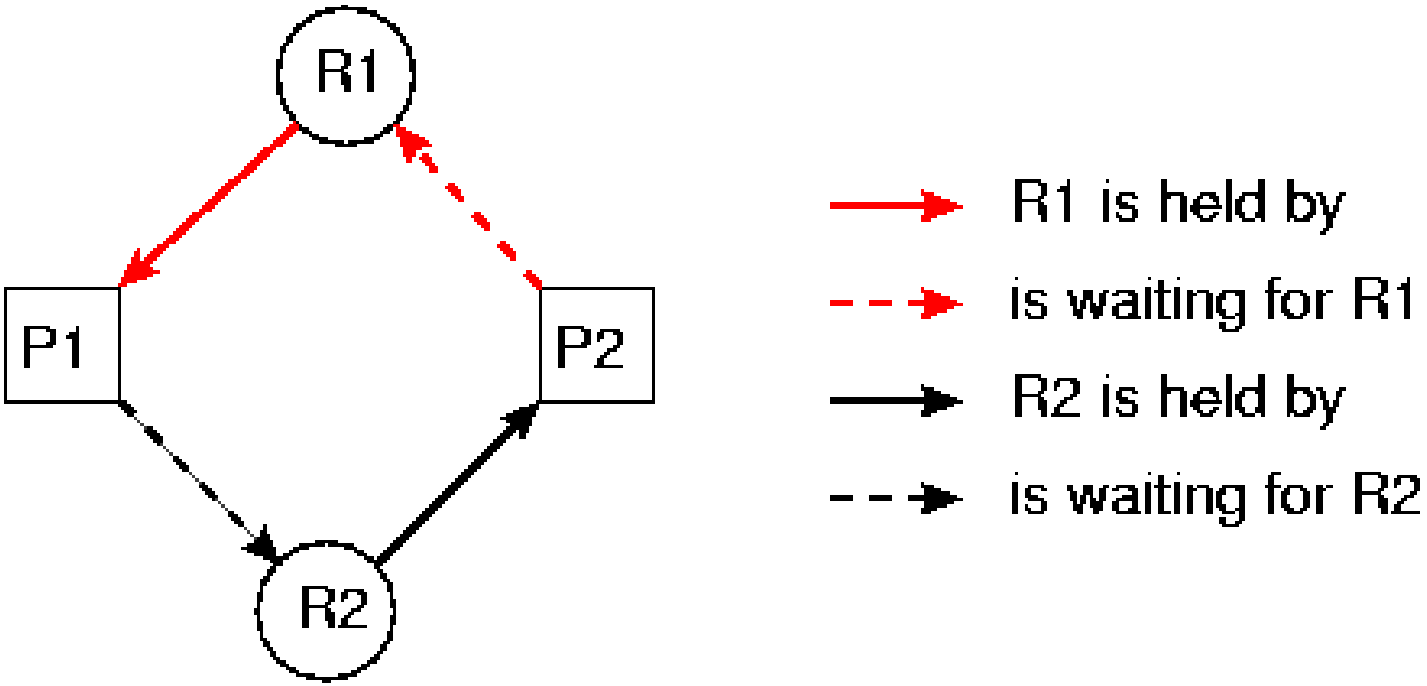
- Multi-Programming
  - keeps multiple runnable jobs loaded in memory at once
  - overlaps I/O of a job with computing of another
    - while one job waits for I/O completion, OS runs instructions from another job
  - goal: optimize system throughput
    - perhaps at the cost of response time…
- Timesharing
  - multiple terminals into one machine
  - each user has illusion of entire machine to him/herself
  - optimize response time, perhaps at the cost of throughput
  - Timeslicing
    - divide CPU equally among the users

A Process's Lifecycle



Deadlock

- Resource graph
  - A deadlock exists if there is an *irreducible cycle* in the resource graph



Second Half of the semester

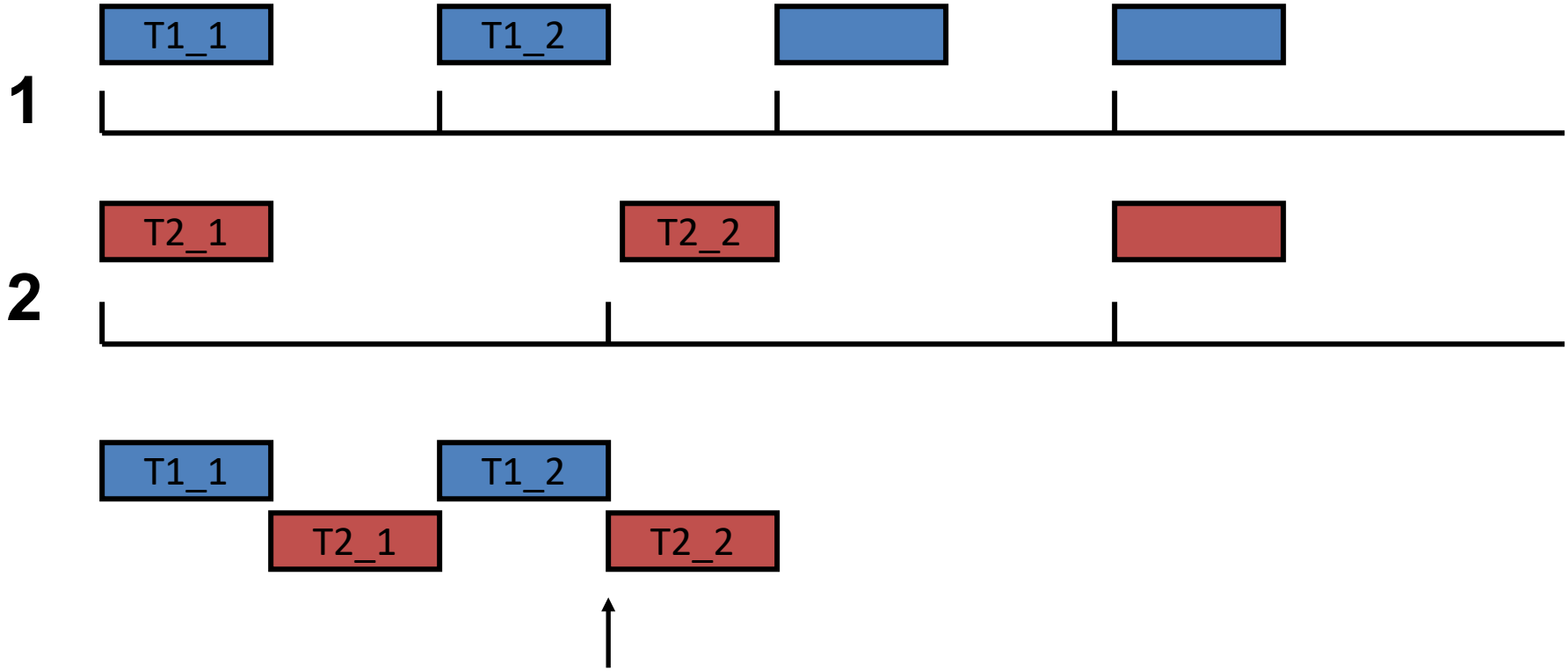
- Real-Time Scheduling
- Program optimizations
- Power management
- Safety and Reliability

Real-time Scheduling

- Terminologies and timing parameters
  - Task, job, subtask
- Metrics to evaluate scheduling algorithms
  - Schedulability, overhead
- Optimal scheduling algorithms
  - When relative deadline = period: RMS, EDF, utilization bound
  - When relative deadline < period: EDF, processor demand analysis
- CPU utilization analysis and bound
- Priority inversion
  - Sources, unbounded priority inversion, priority inheritance
- End-to-end scheduling framework
  - Task allocation: bin packing
  - Synchronization protocol: greedy protocol, release guard
  - Subdeadline assignment: ultimate deadline, proportional deadline

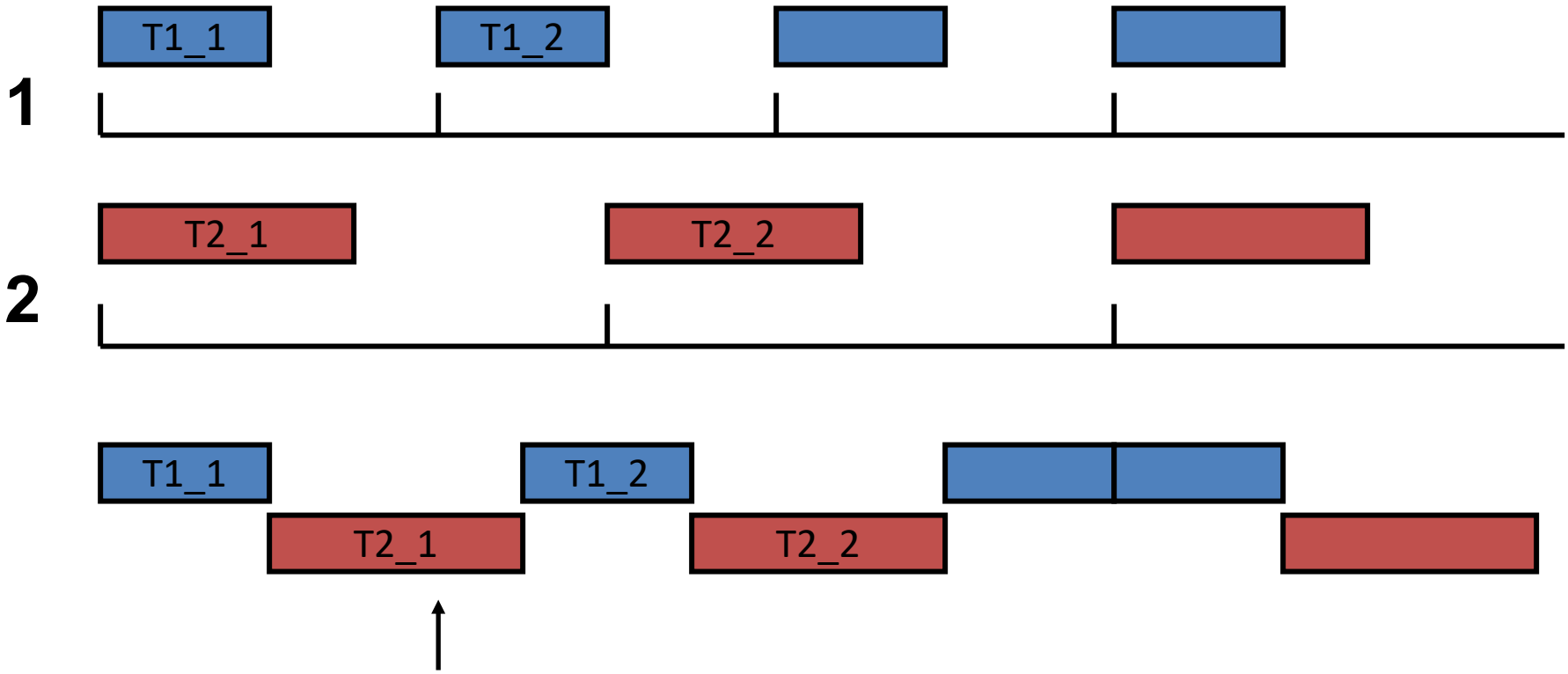
RMS Meeting the Deadline

- T1 = (10,20), T2 = (**10**,30), utilization is **83%**

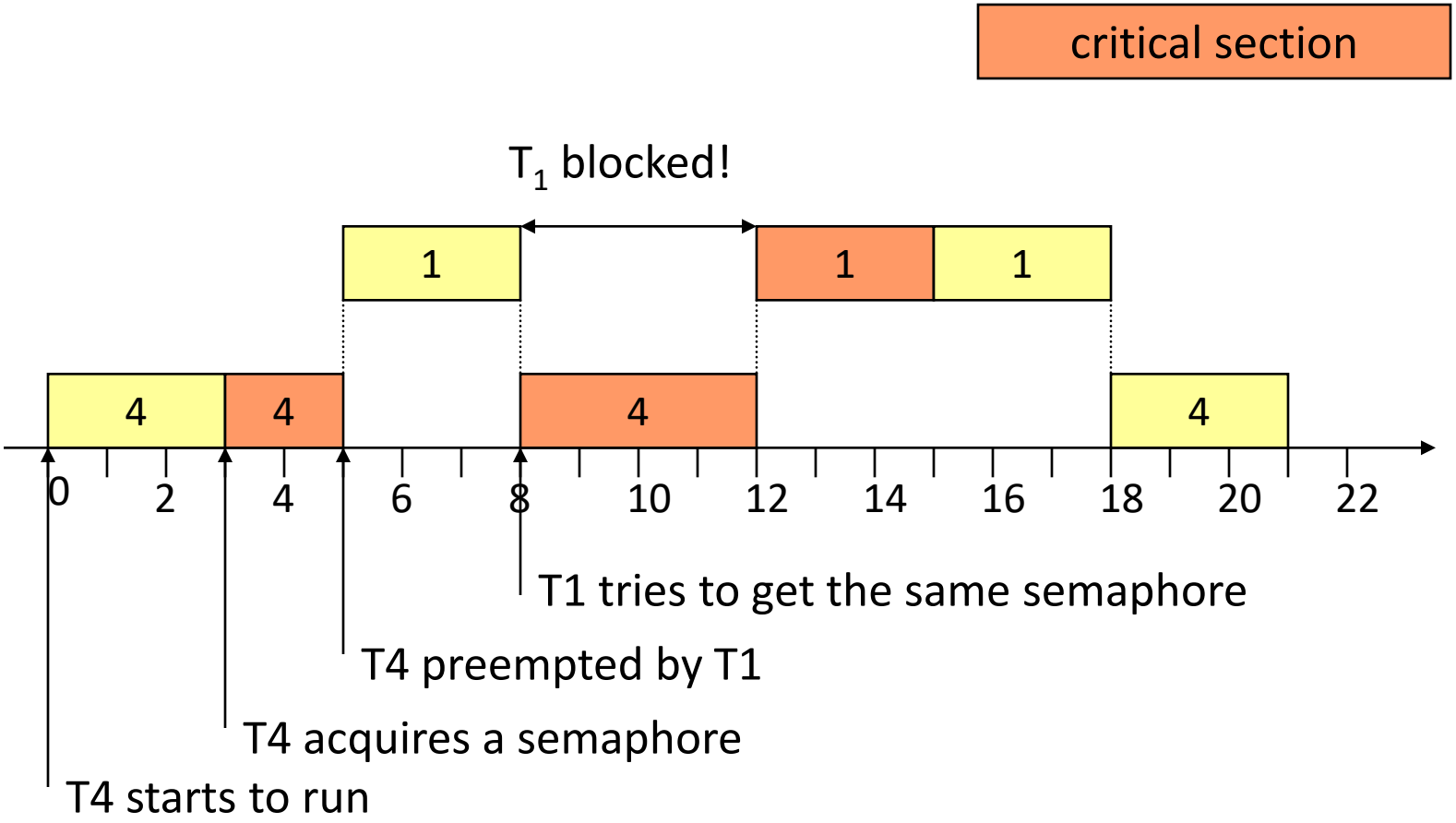


EDF Meeting a Deadline

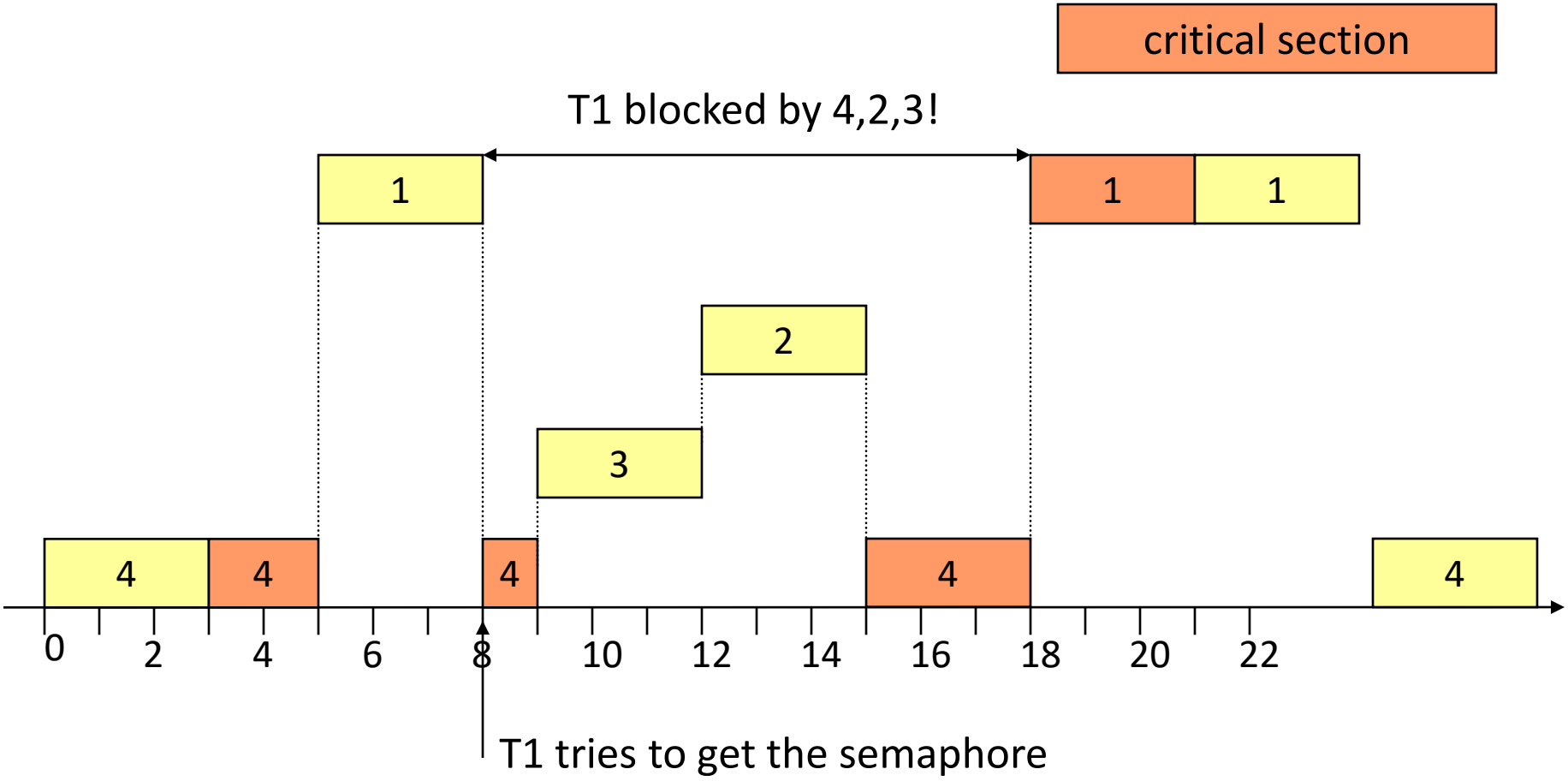
- T1 = (10,20), T2 = (15,30), utilization is 100%



Priority Inversion

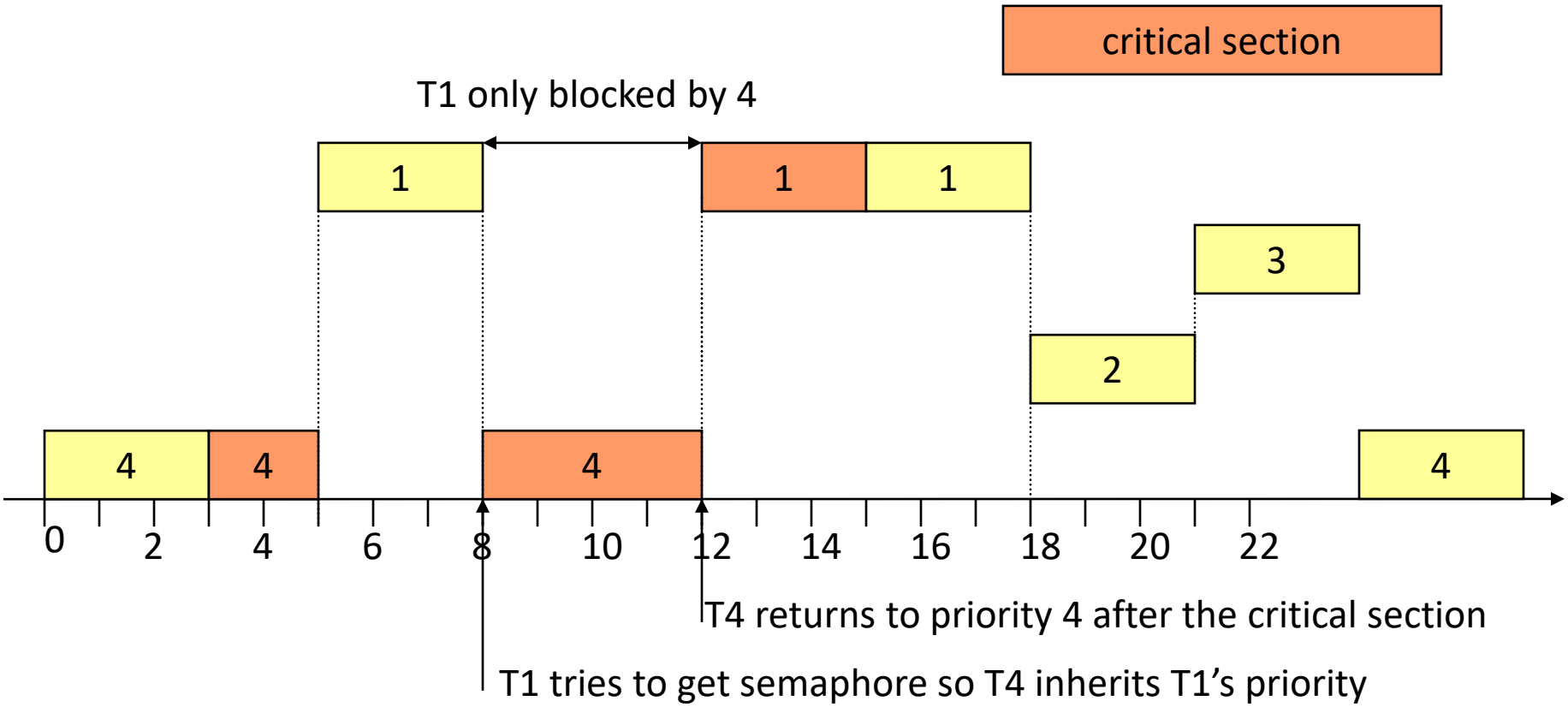


Unbounded Priority Inversion



Solution: Priority Inheritance

- Let the low-priority task inherit the priority of the blocked high-priority task.

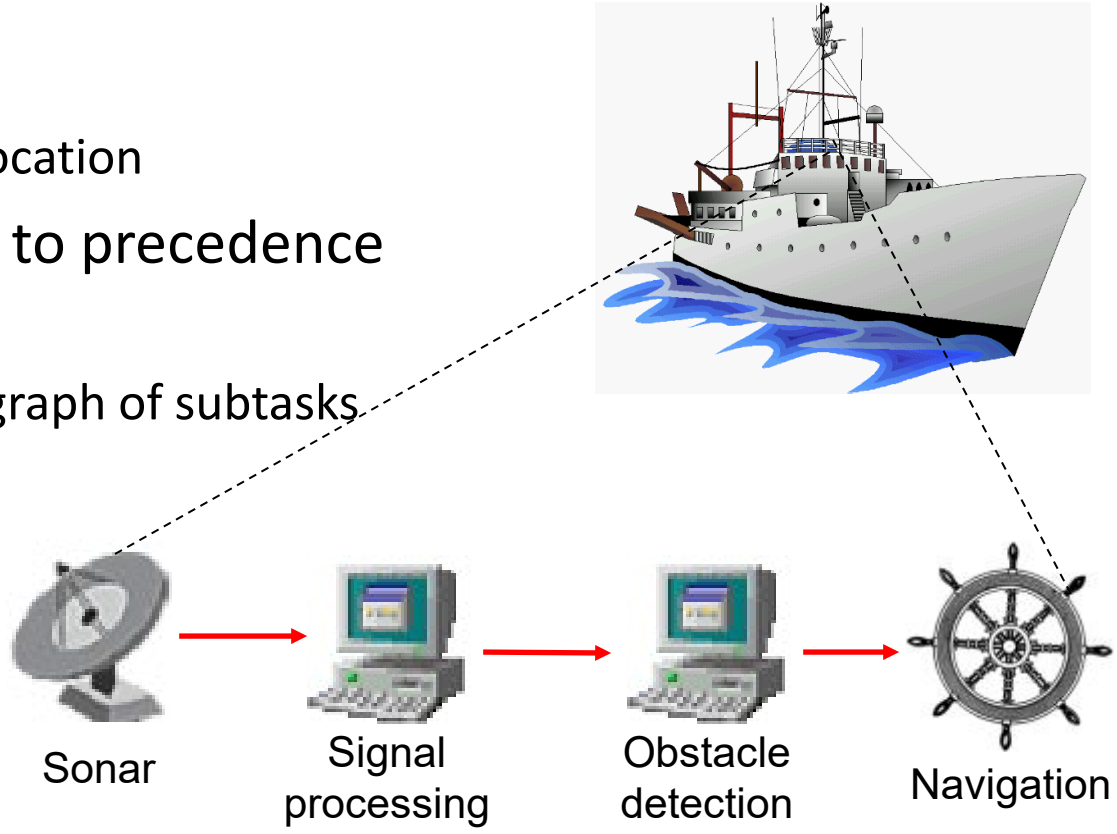


Multi-Processor Systems

- Tight coupling among processors.
- Communicate through shared memory and on-board bus.
- Scheduled by a common scheduler/OS.
  - Global scheduling
  - Partitioned scheduling
- States of all processors available to each other.

End-to-End Task Model

- An (end-to-end) task is composed of multiple subtasks running on multiple processors
  - Message/event
  - Remote method invocation
- Subtasks are subject to precedence constraints
  - Task = a chain/tree/graph of subtasks
  - E.g. ship navigation

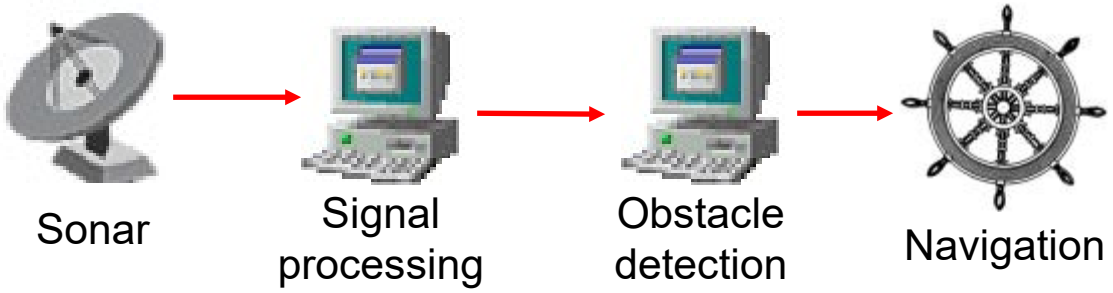


End-to-End Scheduling Framework

1. Task allocation
2. Synchronization protocol
3. Subdeadline assignment
4. Schedulability analysis

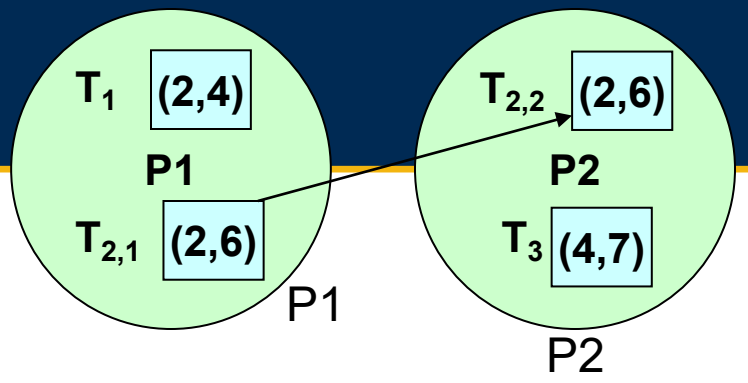
Greedy Protocol

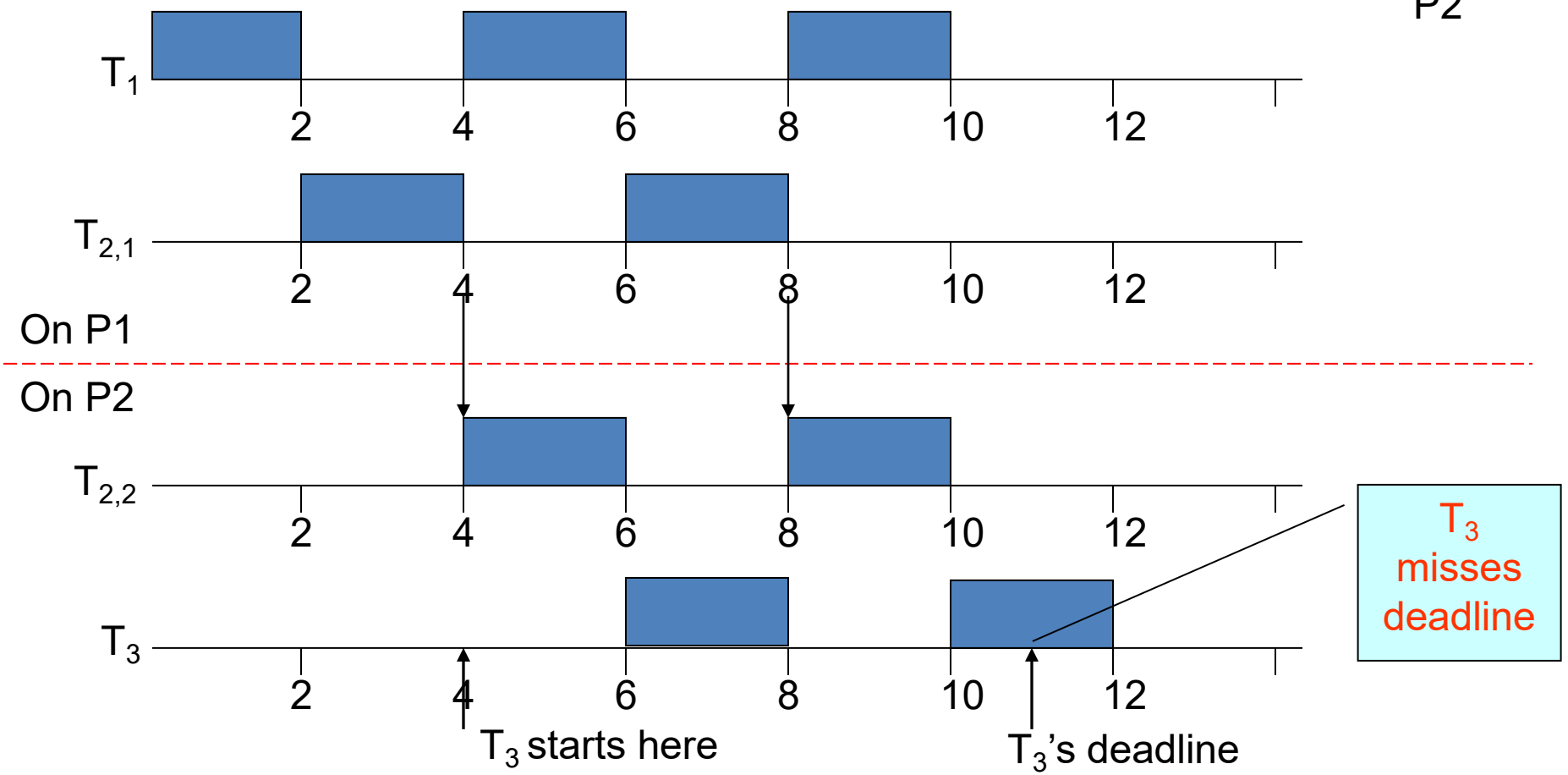
- After a subtask is finished, the next subtask starts immediately
- Release job $J_{i,j;k}$ as soon as $J_{i,j-1;k}$ is completed
- Subsequent subtasks may not be periodic under a greedy protocol
  - Difficult for schedulability analysis
  - High-priority tasks arrive early → high worst-case response time for lower-priority tasks



Greedy Protocol Illustrated

- Task: (C,P)





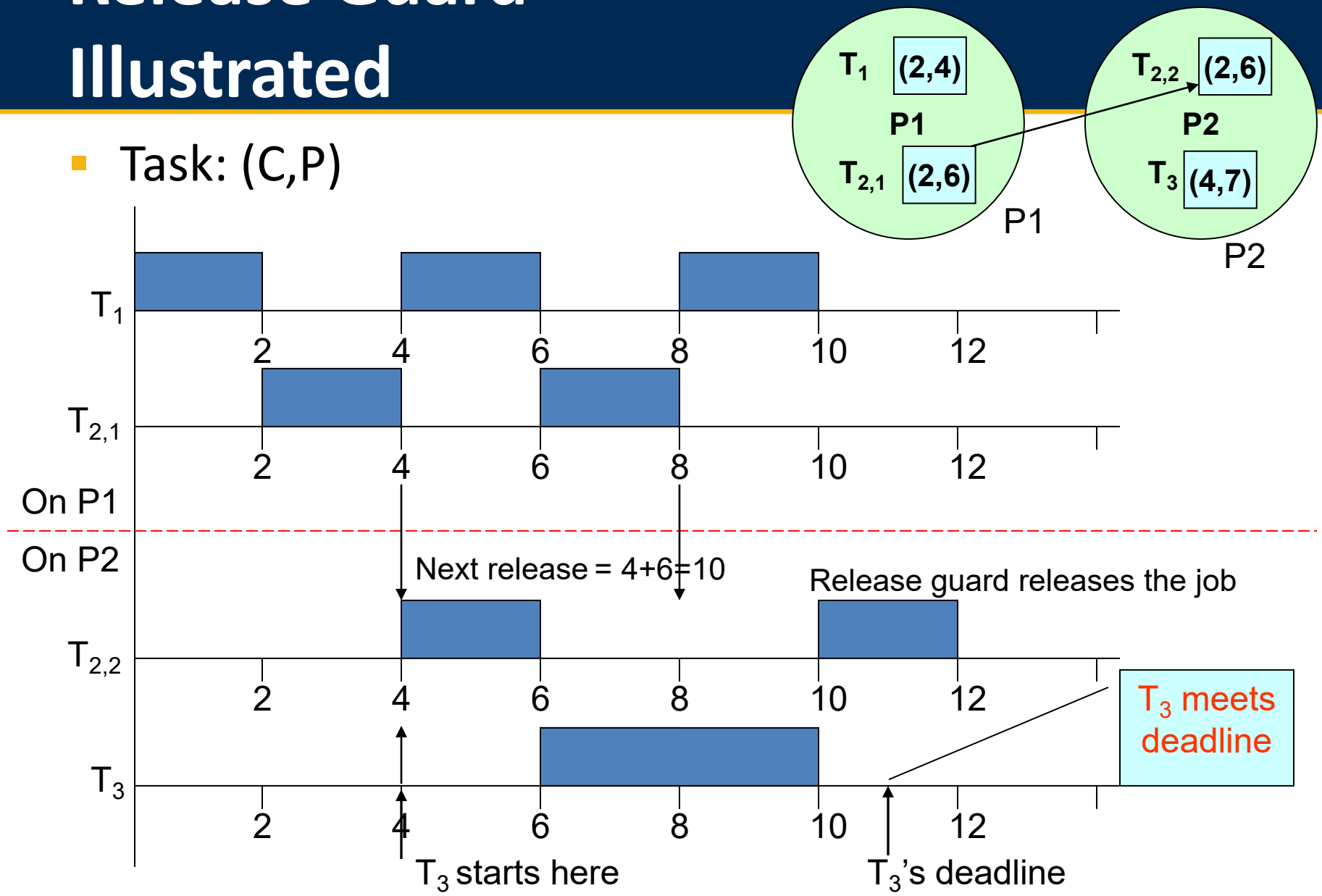
Release Guard

- After a subtask is finished, the next subtask may wait for a while before release
- Every subtask (if not a first subtask) has a release guard, which
  - waits for the preceding subtask for a result/event
  - then releases the job
    - at the point of exact one period from the last release time (Rule1)

    OR
    - whenever the processor becomes idle (Rule 2)
- Release guard strategy improves worst response time without affecting schedulability

Release Guard Illustrated

- Task: (C,P)



Basic Compilation Optimization

- Expression simplification
- Dead code elimination
- Function inlining
- Loop optimizations
- Register allocation

1. Expression Simplification

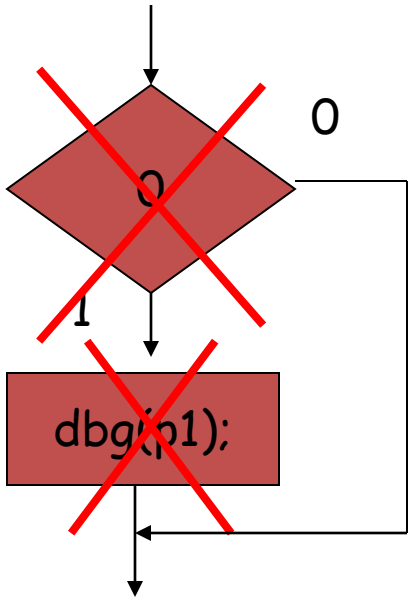
- Algebraic:
  - `a*b + a*c` ⇨ `a*(b+c)`
  - Three operations change to two operations
- Constant folding:
  - `for (i=0; i<8+1; i++)`
  - ⇨ `for (i=0; i<9; i++)`
- Strength reduction of constant multiplication
  - `a*2`⇨`a<<1; a*3`⇨`a*(2+1)`⇨`a<<1+a;`
  - `a*1000`⇨`a*(1024-16-8)`⇨`a<<10 – a<<4 – a<<3`
  - SHIFT, ADD, SUB take one cycle but multiplication takes about 10 cycles

2. Dead Code Elimination

- Dead code:

```
#define DEBUG 0
if (DEBUG) dbg(p1);
```

- Can be eliminated by analysis of control flow, constant folding



Function inlining

```
int foo(a,b,c) { return a + b - c;}
z = foo(w,x,y);
```

```
z = w + x - y;
```

- An inline function's body is inserted directly (like a substitution) in the compiled code at the point where the function is called.
- Improve performance by reducing function call overhead
- "inline" in different cases
  - TinyOS does whole-program inlining

Loop Optimizations

- Loops are good targets for optimization.
- Basic loop optimizations:
  - Code motion;
  - Reduce loop overhead: loop unrolling
  - Increase opportunities for pipelining and parallelism: loop fusion

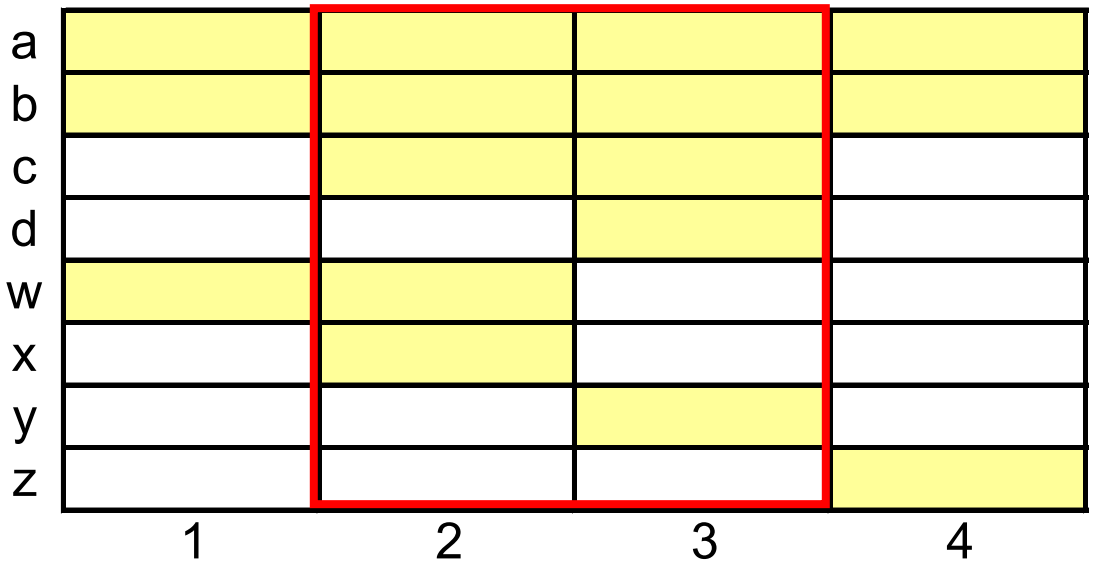
Register Allocation

- Processor registers
  - A very small amount of very fast computer memory
  - Used to speed the execution of computer programs
  - Provides quick access to most commonly used values
  - Memory hierarchy: register – cache – main memory – disk
- Reduce the number of used registers
  - Fit more frequently used variables in registers
  - Load once, use many times

Register Lifetime Graph

```
1. w = a + b;
2. x = c + w;
3. y = c + d;
4. z = a - b;
```

no. of needed registers = 5

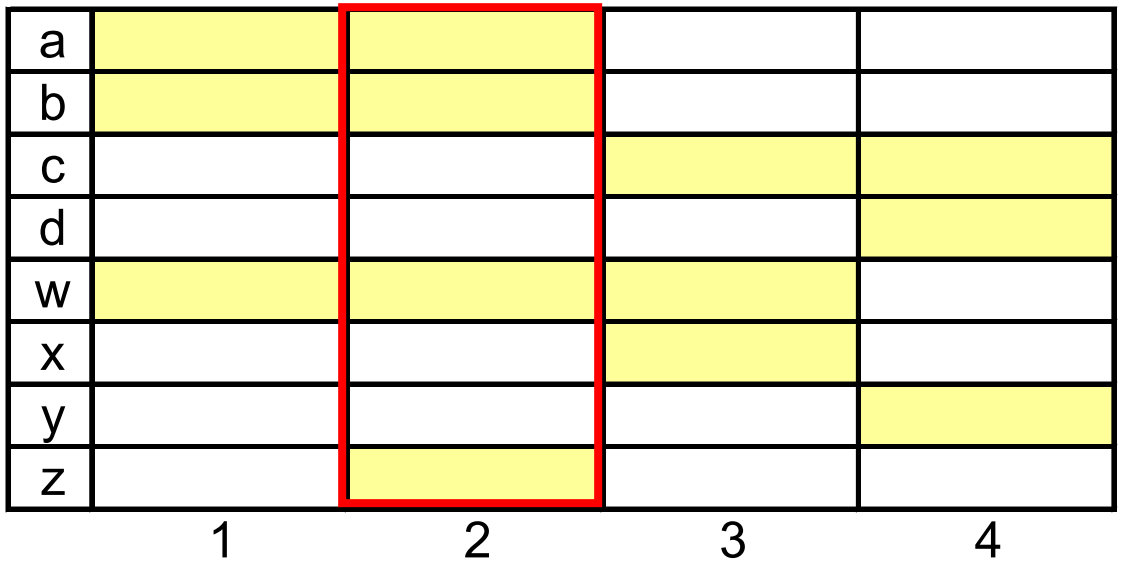


means this variable should be loaded to register

After Rescheduling

```
1. w = a + b;
2. z = a - b;
3. x = c + w;
4. y = c + d;
```

no. of needed registers = 4



Cannot change dependencies between instructions!

Power Management

- Hardware support
  - CMOS features: voltage drops, toggling, leakage
  - Clock gating, supply shutdown, dynamic voltage scaling
- Power management policy
  - Dynamic power management
  - Power state machine, break-even time T_{BE}
  - Energy saving calculation based on a known idle time
  - Predictive techniques
    - Metrics of prediction quality: safety and efficiency
    - Fixed timeout vs. predictive shutdown/wakeup
- Power manager
  - Advanced Configuration and Power Interface (ACPI)
- Holistic approach
  - Memory system, cache behavior

CMOS Power Consumption

- All digital systems are built with CMOS
  - CMOS: Complementary Metal-Oxide Semiconductor
  - A common technology for constructing integrated circuits
- Voltage drops
  - Power consumption of a CMOS is proportional to the square of the power supply voltage (V^2).
- Toggling
  - CMOS uses higher power when having more activity
- Leakage
  - Even when CMOS is inactive, some charge leaks out of the circuit's nodes

General Power-Saving Features

To deal with toggling

- Run at lower clock frequency (slower)
- Reduce activity by disabling function units when not in use.

To deal with leakage

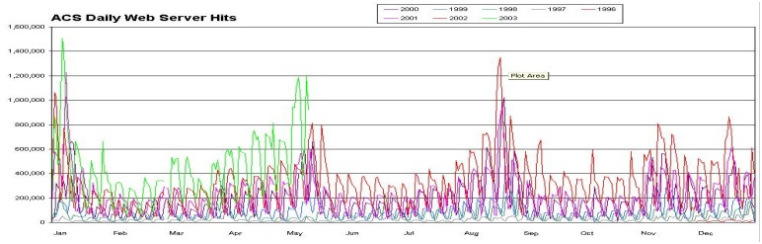
- Eliminate leakage current by disconnecting parts from power supply when not in use.

To deal with voltage drops

- Reduce power supply voltage to the lowest level that provides required performance
  - Pentium MMX: 2.8v
  - i7: ~1.5v
  - Cortex A8: 1.2v

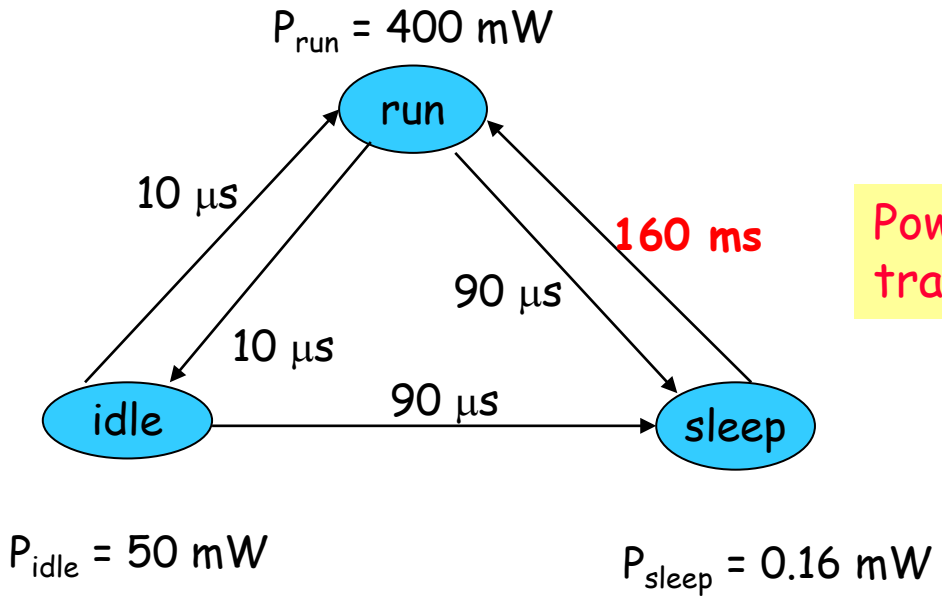
Dynamic Voltage Scaling

- Why voltage scaling?
  - Power $\propto V^2$
    - Reduce power supply voltage → save energy
  - Lower clock frequency allows lower voltage
    - Tradeoff between performance and battery lifetime
    - Overclocking? Raise voltage for better stability
- Why dynamic?
  - Power consumption is not a constant.
    - Depending on current workloads.
    - Peak computing rate is usually much higher than average.
  - Have to be dynamic to respond to power consumption variations.



Break-Even Time T_{BE}

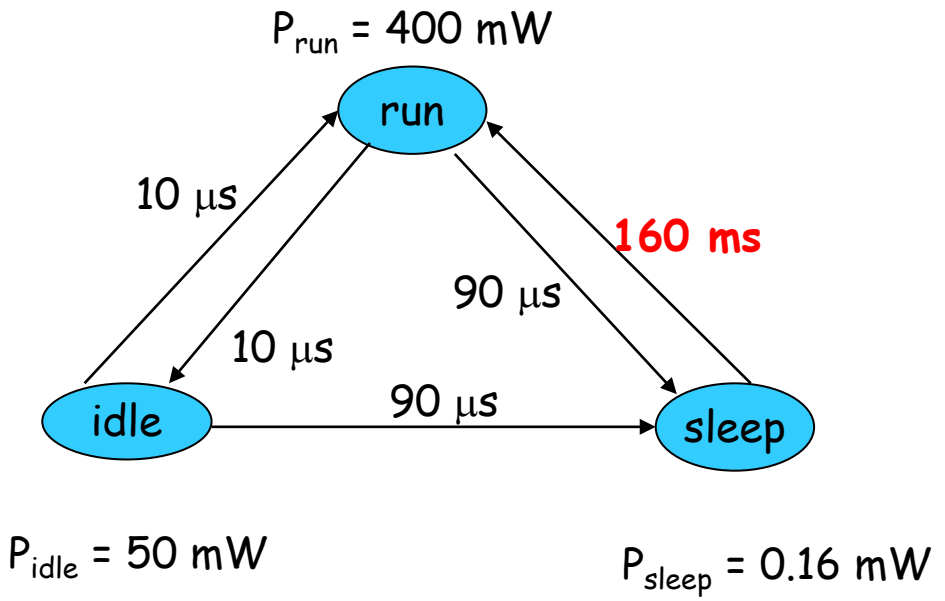
- T_{BE} of an inactive state is the total time for entering and leaving the state
  - Assumption: transition doesn't cause extra power consumption
- $T_{BE} = T_{TR} = T_{On,Off} + T_{Off,On}$
  - Ex. T_{BE} = 160 ms + 90 μs for SLEEP in SA-1100



Power consumption during transition ≈ P_{run}

Break-Even Time When $P_{TR} > P_{On}$

- $T_{BE} = T_{TR} + T_{TR}(P_{TR} - P_{On})/(P_{On} - P_{Off})$
  - $T_{TR}(P_{TR} - P_{On})$: extra energy consumed for transition
  - $/(P_{on}-P_{off})$: the idle time needed to compensate this energy consumption back
- It is easier to save power with a shorter T_{BE}
  - Shorter T_{TR}
    - IDLE in SA-1100
  - Higher difference between $P_{On} - P_{Off}$
    - SLEEP in SA-1100
  - Lower P_{TR}



Energy Saving Calculation

- Given an idle period $T_{idle} > T_{BE}$
  - $E_S(T_{idle}) = (T_{idle} - T_{TR})(P_{On} - P_{OFF}) + T_{TR}(P_{On} - P_{TR})$
    - $P_{On} > P_{TR}$: total = idle saving + transition saving
    - $P_{On} < P_{TR}$: total = idle saving - transition cost
- Achievable power saving depends on workload!
  - Distribution of idle periods

Predictive Techniques

- Don’t know how long the idle time will be?
- Predict whether idle time $T_{idle} > T_{BE}$ based on history
  - Ex: someone takes break once an hour?
  - Predicted event: $p = \{T_{idle} > T_{BE}\}$
  - Observed event: *o*
    - Triggers state transition: RUN -> IDLE/SLEEP

Metrics of Prediction Quality

- **Safety**: conditional probability Prob(p|o)
  - If an observed event happens, what's the probability of $T_{idle} > T_{BE}$?
  - Ideally, safety = 1.
- **Efficiency**: Prob(o|p)
  - If $T_{idle} > T_{BE}$, what's the probability of successfully predicting it in advance (e.g., o happens)?
- Overprediction
  - State transition too much
  - High performance penalty → poor safety
- Underprediction
  - State transition not enough
  - Wastes energy → poor efficiency

Fixed Timeout Policy

- Enter inactive state when the system has been idle for T_{TO} sec.
  - o: $T_{idle} > T_{TO}$
- Wake up in response to activity
- Assumption: If a system has been idle for T_{TO} sec, it is likely that it will continue to be idle for $T_{idle}-T_{TO} > T_{BE}$.

Selecting Best Timeout T_{TO}

- Increasing T_{TO} improves safety, but reduces efficiency at the same time
  - Safety: $Pr(T_{idle} > T_{TO} + T_{BE} \mid T_{idle} > T_{TO})$
- Highly workload dependent
- Karlin's result: $T_{TO} = T_{BE}$

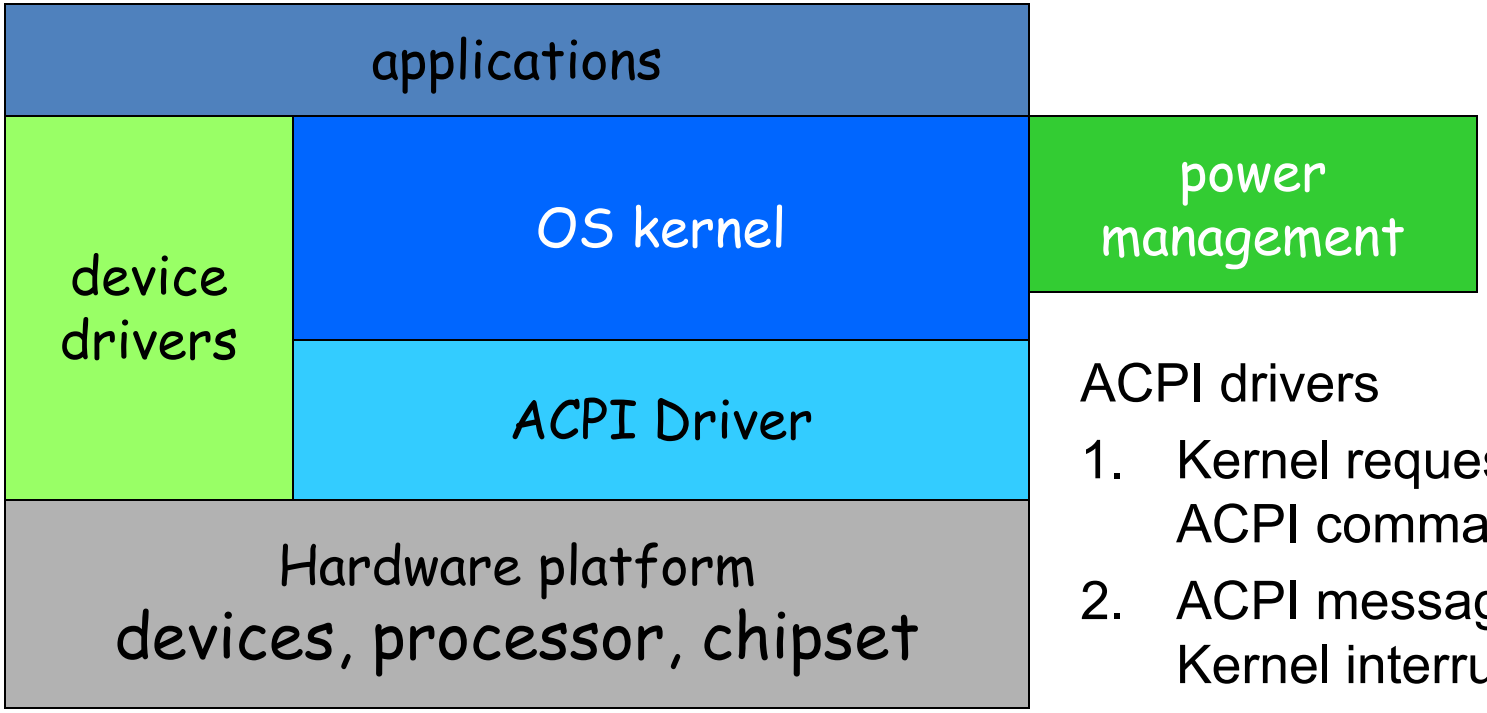
  → Energy consumption is at worst twice the energy consumed by an ideal policy

Possible Improvement

- Predictive shutdown: shutdown *immediately* when an idle period starts
  - Avoid wasting energy before reaching the timeout threshold
  - More efficient, less safe
- Predictive wakeup: wake up before the *predicted* idle time expires, even if no new activity has occurred.
  - Avoid performance penalty for wake up
  - Less efficient, safer

Advanced Configuration and Power Interface (ACPI)

- Open standard for power management services.
  - Proposed by Intel, Microsoft and Toshiba

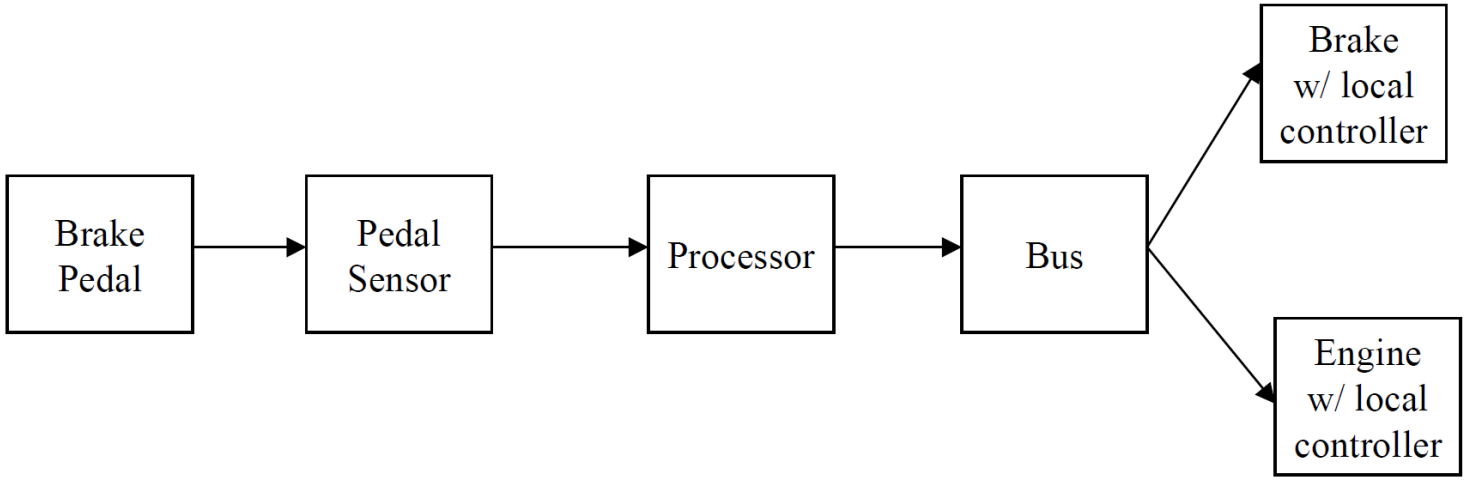


ACPI drivers

1. Kernel requests → ACPI commands
2. ACPI messages → Kernel interrupts

What is a “Safe” Embedded System?

- Again, the ABS system’s case



- Is it safe?
- What does “safe” mean?
- How can we make it safe?
- Add watch dog between brake and bus
- Add mechanical linkage from brake pedal directly to brake
- Add a third mechanical linkage…

Component vs. System

- Reliability is a component issue
- **Safety** and **Availability** are system issues
- A system can be safe even if it is unreliable!

- If a system has lots of redundancy the likelihood of a component failure (a fault) increases, but so may increase the safety and availability of that system

- Safety and Availability are different and sometimes at odds. Safety may require the shutdown of a system that may still be able to perform its function.
  - A backup system that can fully operate a nuclear power plant might always shut it down in the event of failure of the primary system.
  - The plant could remain available, but it is unsafe to continue operation

Terms

- Safety: assuming acceptable risk
- Hazard: dangerous system state that could cause damage
- Fault: conditions that lead to hazards
- Reliability
  - System is functioning if all components are functioning
    - $P_s(t) = \prod_i P_i(t)$
  - System is functioning
    - $P_s(t) = 1 - \prod_i F_i(t)$
    - $F_{i(t)} = 1 - Pi(t)$: probability of component failure

Safety Architectures

- Self checking (single channel protected design)
  - Perform periodic checksums on code and data
  - How long does it take?
  - No protection against systematic faults
- Redundancy
  - Homogeneous redundancy
  - Heterogeneous redundancy
- Diversity or Heterogeneity