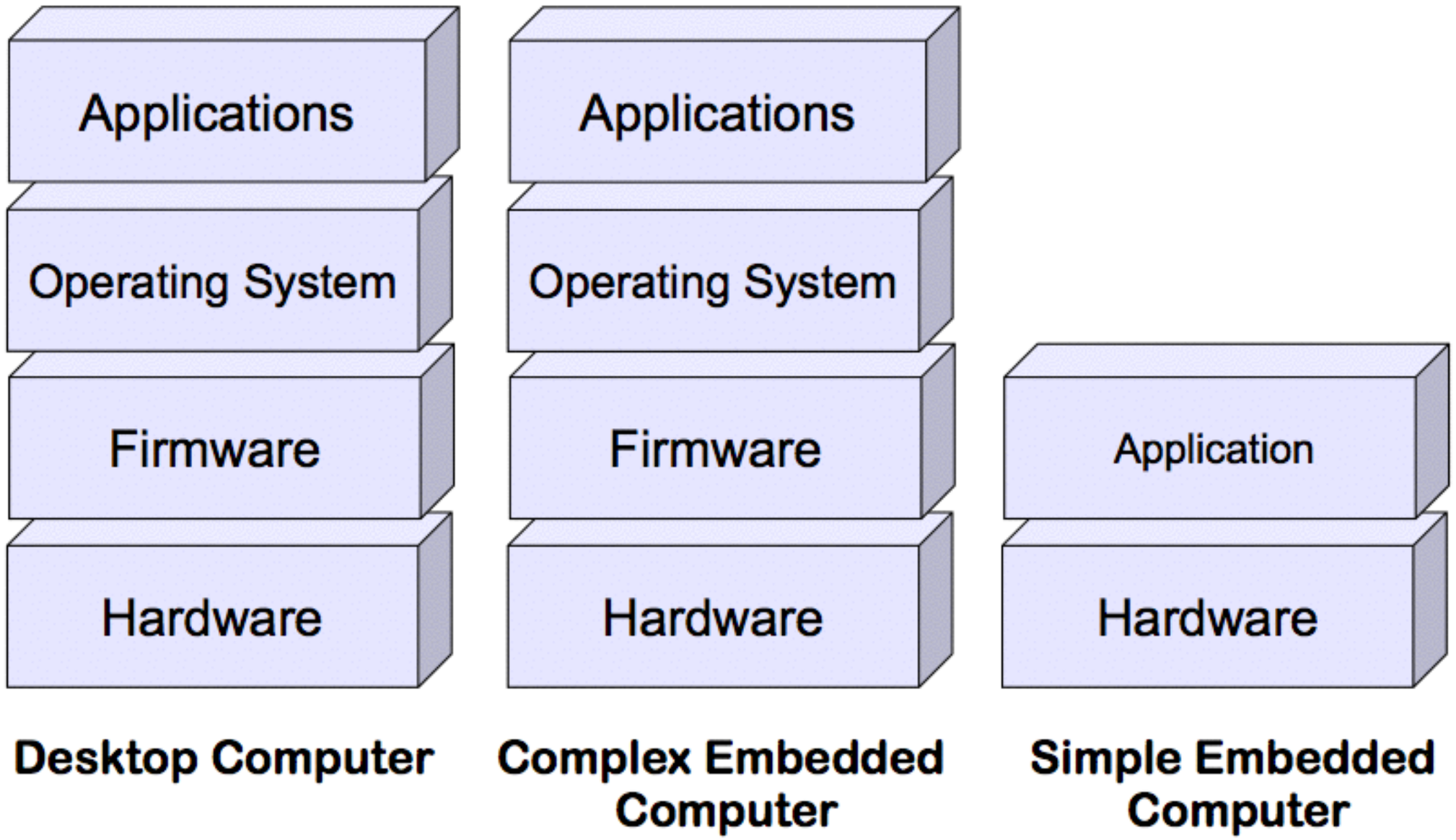
Embedded Software System

Agenda

- Embedded Software Architectures
- Real time operating systems (RTOS)

Software Layers



Survey Of Embedded Software Architectures

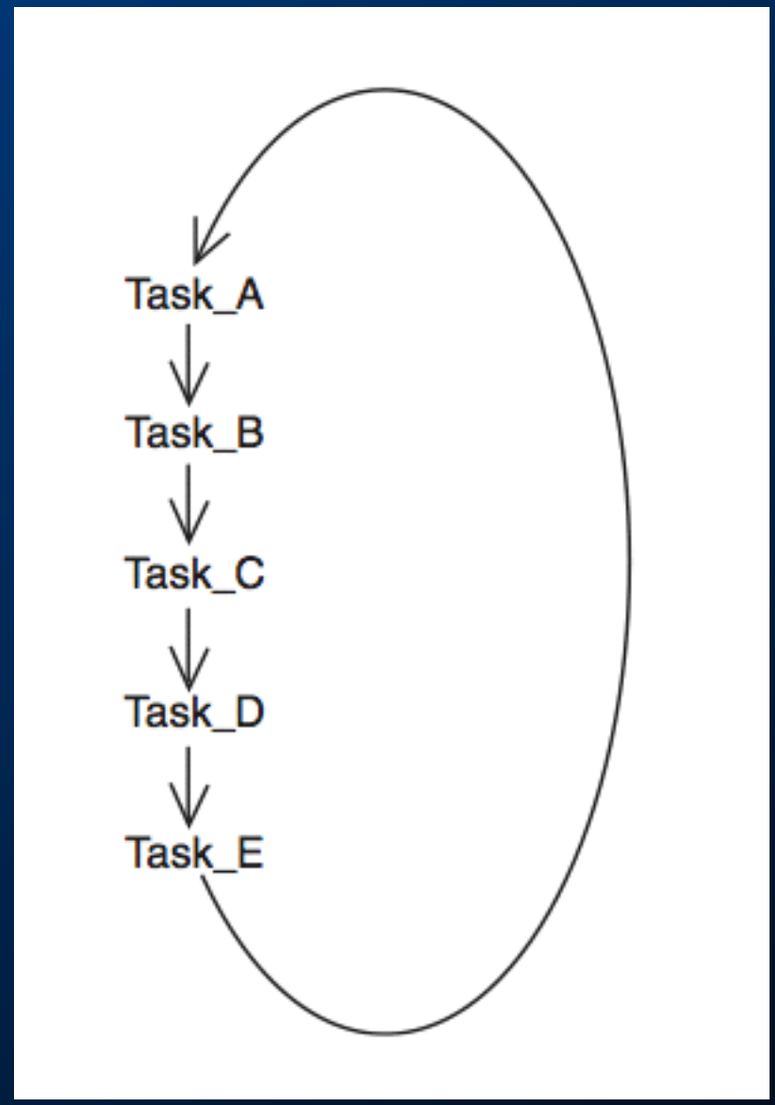
- Round Robin
- State Machine
- Round Robin with Interrupts
- Just interrupts
- Function Queue Scheduling
- Real time operating systems (RTOS)

Round Robin

Round Robin / Control Loop

Everything is a function call from the main loop

```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all devices have been serviced, then start over again
    }
}
```



Round Robin

- Low priority tasks need to be slowed down

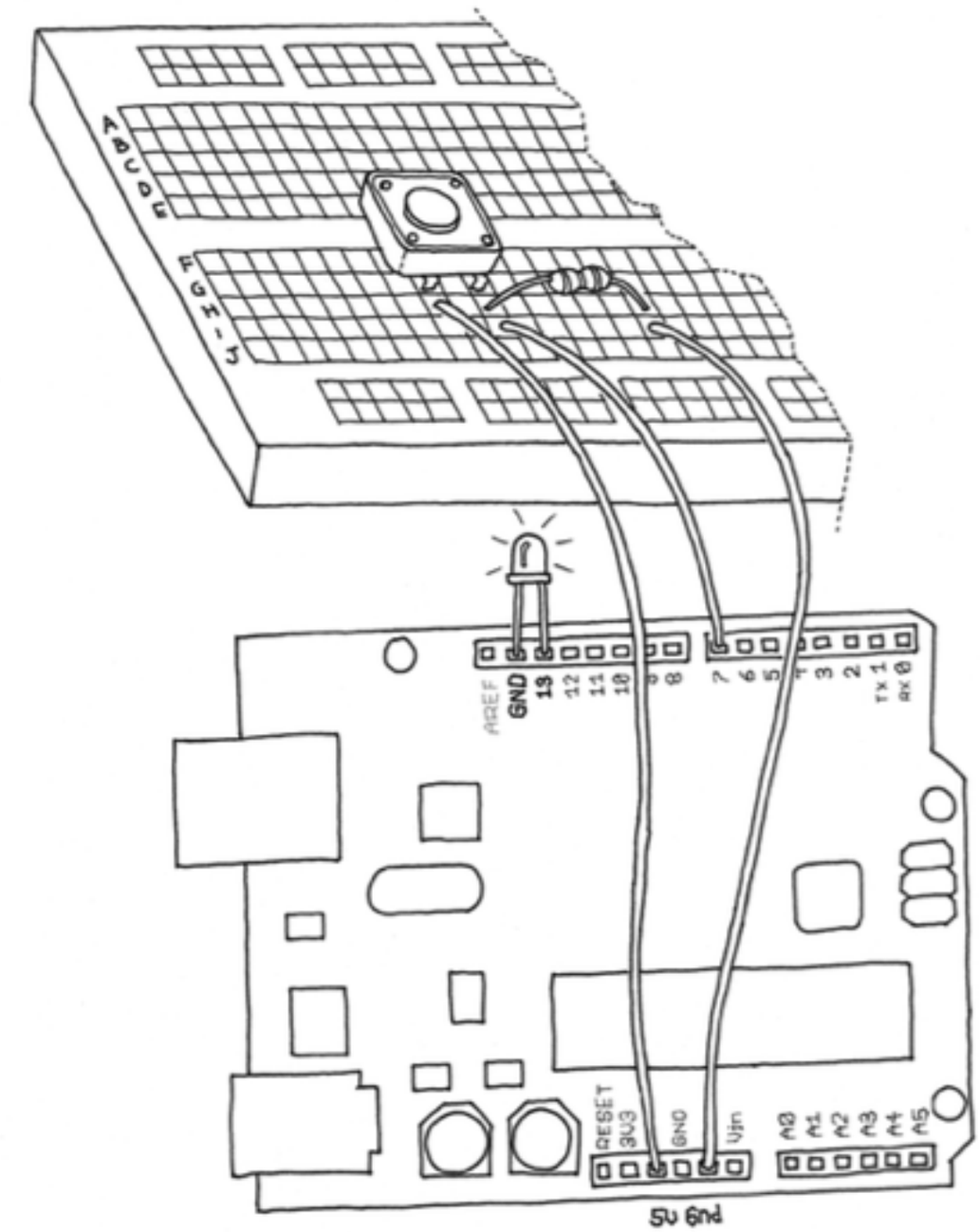
```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_A requires service)
            service device_A
        if (device_C requires service)
            service device_C
        if (device_A requires service)
            service device_A
        ... and so on, making sure high-priority device_A is always serviced on time
    }
}
```

Round Robin

- Priority – None, everything runs in sequence.
- Response time – The sum of all tasks.
- Impact of changes – Significant. Changing the execution time of tasks or adding tasks impacts all other tasks.
- Simplicity, no shared data problems.

Arduino – Turn on LED while the button is pressed

```
const int LED = 13; // the pin for the LED
const int BUTTON = 7; // the input pin where the
                      // pushbutton is connected
int val = 0; // val will be used to store the state
             // of the input pin
void setup() {
  pinMode(LED, OUTPUT); // tell Arduino LED is an
output
  pinMode(BUTTON, INPUT); // and BUTTON is an
input
  }
void loop(){
  val = digitalRead(BUTTON); // read input value and
store it
  // check whether the input is HIGH (button pressed)
  if (val == HIGH) {
    digitalWrite(LED, HIGH); // turn LED ON
  }else{
    digitalWrite(LED, LOW);
  }
}
```



Arduino – Watchdog

```
#include <avr/wdt.h>
#define TIMEOUT WDTO_8S      // predefine time, refer avr/wdt.h
const int ledPin =  13;      // the number of the LED pin

void setup(){
    // disable the watchdog
    //wdt_disable();
    pinMode(ledPin,OUTPUT);
    // LED light once after start or if timeout
    digitalWrite(ledPin,HIGH);
    delay(1000);
    // enable the watchdog
    wdt_enable(TIMEOUT);
}

 void loop(){
    // process runing
    digitalWrite(ledPin,LOW);
    delay(9000); //if timeout trig the reset
    //feed dog
    wdt_reset();
}
```

State Machine

```
while(1) {
        switch(state) {
                case IDLE:
                        check_buttons();
                        LEDisplay_hex(NUM1);
                        if (BUTTON1 | BUTTON2 | BUTTON3)
                                state=SHOW;
                        break;
                case SHOW:
                        NUM1=0;
                        if (BUTTON1) NUM1 += 0x0001;
                        if (BUTTON2) NUM1 += 0x0010;
                        if (BUTTON3) NUM1 += 0x0100;
                        state=IDLE;
                        break;
        }
}
```

State Machine

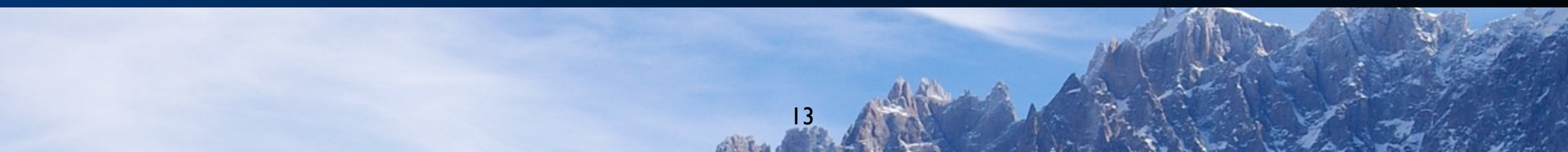
- Similar to round robin, but only the current state gets executed.
- Each state determines the next state (non-sequential execution).

State Machine

- Priority – Each state determines the priority of the next state.
- Response time – The sum of all tasks.
- Impact of changes – Significant.
  Changing the execution time of tasks or adding tasks impacts all other tasks.
- Simplicity - No shared data problems.

Round Robin with Interrupts

```
BOOL flag_A = FALSE; /* Flag for device_A follow-up processing */
/* Interrupt Service Routine for high priority device_A */
ISR_A(void) {
    ... handle urgent requirements for device_A in the ISR,
    then set flag for follow-up processing in the main loop ...
        flag_A = TRUE;
}
void main(void) {
    while(TRUE) {
        if (flag_A)
            flag_A = FALSE
            ... do follow-up processing with data from device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all high and low priority devices have been serviced
        }
}
```



Round Robin with Interrupts

- Priority – Interrupts get priority over main loop
  - Priority of interrupts as well
- Response time –
  - The sum of all tasks or
    - Interrupt execution time
- Impact of changes – Less significant for interrupt service routines. Same as Round Robin as main loop.
- Shared data – must deal with data shared with interrupt service routines

Just Interrupts

```
SET_VECTOR(P3AD, button_isr);
SET_VECTOR(TIMER1, display_isr);
while(1) {
;
}
```

Just interrupts

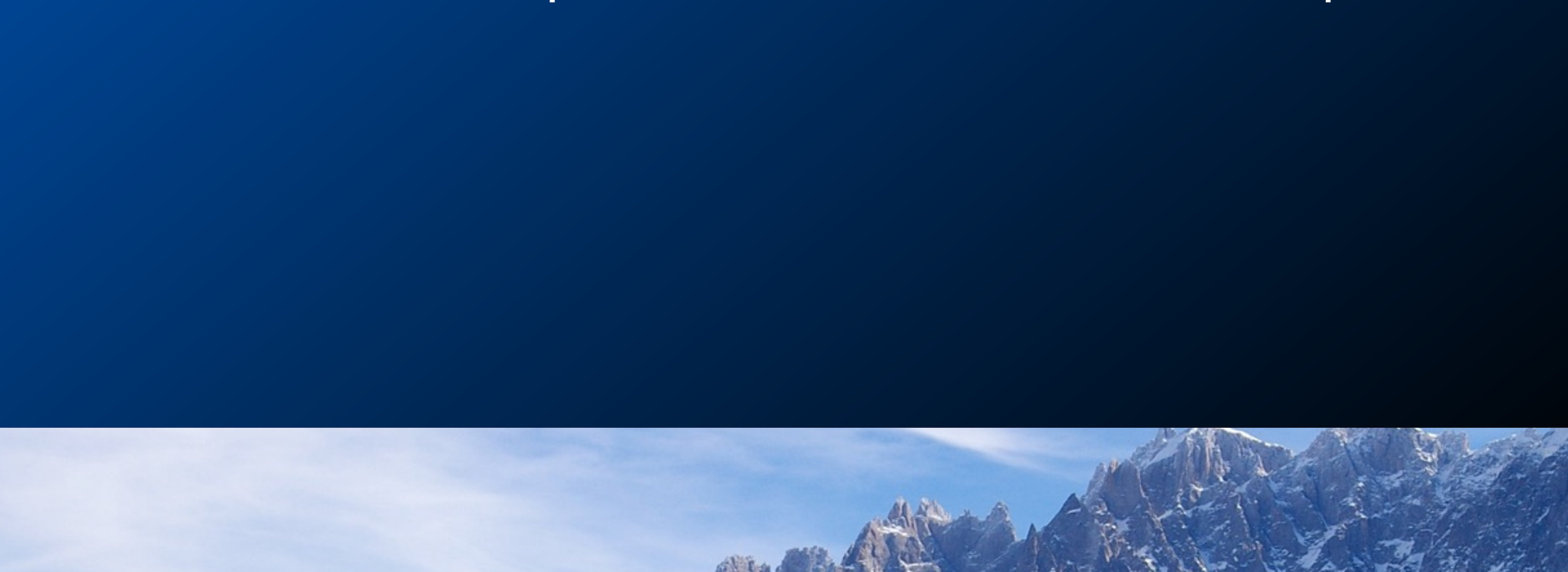
- Can have problems if too many ISRs
- If a high priority interrupt takes longer to execute than lower priority interrupts, then some will get missed.
  - Or you need to deal with nested interrupts.

Just Interrupts

- Priority – Interrupts get priority
- Response time –
  - Interrupt execution time
- Impact of changes – Little significant for interrupt service routines.
- Shared data – Must deal with data shared with interrupt service routines

Function Queue Scheduling

- Function pointers are added to a queue.
- The main loop cycles through the queue and executes tasks.
- Tasks or interrupts add new tasks to the function queue.

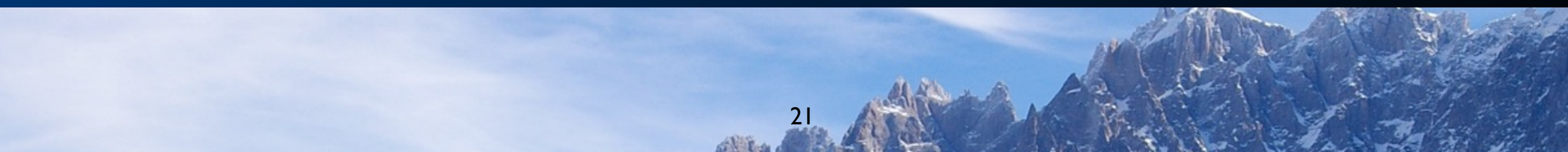


Function Queue Scheduling

```
#define MAX_TASKS 20
typedef int(*FuncPtr)();
FuncPtr tasks[MAX_TASKS]
int current_task = 0;

void add_task(FuncPtr func) {
        int n;
        for(n=current_task+1;n<MAX_TASKS-1;n++) {
            if(tasks[n]==NULL) {
                tasks[n]=func;
                return;
            }
        }
        for(n=0;n<current_task;n++) {
            if(tasks[n]==NULL) {
                tasks[n]=func;
                return;
            }
    }
```

```
id display_task() {
    LEDisplay_hex(NUM1);
    add_task(button_task);
}

void button_task() {
      check_buttons();
      NUM1=0;
      if (BUTTON1)     NUM1 += 0x0001;
      if (BUTTON2)     NUM1 += 0x0010;
      if (BUTTON3)     NUM1 += 0x0100;

      add_task(display_task);
}
```

```
main()  {
        LEDisplay_init();
        LEDisplay_clear();
        init_buttons();

        add_task(button_task);

        while(1) {
                if(tasks[current_task]==NULL) {
                ;
                }
                else {
                    (*tasks[current_task])();
                    tasks[current_task]=NULL;
                }
                current_task++;
                if(current_task>=MAX_TASKS) current_task=0;
        }
}
```



Function Queue Scheduling

- Priority – Interrupts have priority. Tasks execute in sequence
- Response time – Execution time of the longest task
- Impact of changes – Low. Interrupts manage priority functions. Queue manages lower priority.
- Shared data – must deal with data shared with interrupt service routines

Function Queue Improvements

- Include time scheduling

```
typedef int(*FuncPtr);

typedef struct {
    long timer;
    int status;
    FuncPtr;
} Task;
Task task_list[MAX_TASKS];
```

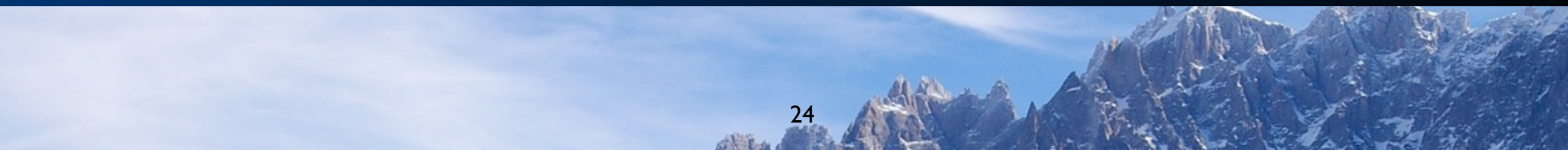
An interrupt decrements all task timers. When it reaches 0 its available for execution

Function Queue Improvements

Include task priority

```
typedef int(*FuncPtr);


typedef struct {
    int priority;
    FuncPtr;
} Task;
Task task_list[MAX_TASKS];
```

Highest priority tasks get moved to the head of the queue.



Function Pointers

- The Function Pointer Tutorial
  - http://www.newty.de/zip/e_fpt.pdf

Preemptive multitasking or multi-threading

- In this type of system, a low-level piece of code switches between tasks or threads based on a timer (connected to an interrupt). This is the level at which the system is generally considered to have an "operating system" kernel. Depending on how much functionality is required, it introduces more or less of the complexities of managing multiple tasks running conceptually in parallel.
- To access to shared data must be controlled by some synchronization strategy, such as message queues, semaphores or a non-blocking synchronization scheme.
- Because of these complexities, it is common for organizations to use a real-time operating system (RTOS), allowing the application programmers to concentrate on device functionality rather than operating system service.

Microkernels

- A microkernel is a logical step up from a real-time OS. The usual arrangement is that the operating system kernel allocates memory and switches the CPU to different threads of execution. User mode processes implement major functions such as file systems, network interfaces, etc.
- In general, microkernels succeed when the task switching and intertask communication is fast and fail when they are slow.

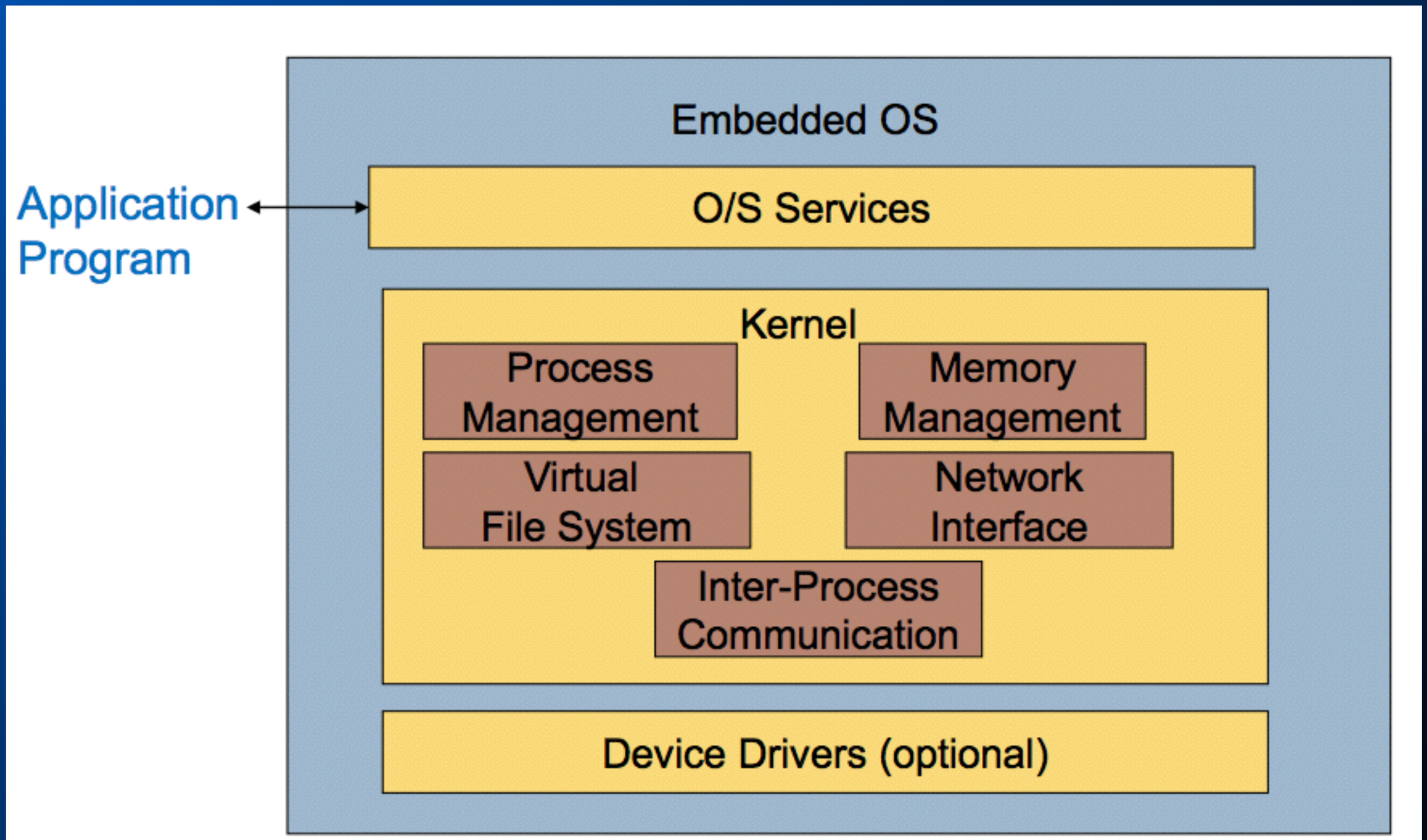
Monolithic kernels

- a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment.
    - examples: embedded Linux and Windows CE
- Despite the increased cost in hardware, this type of embedded system is increasing in popularity, especially on the more powerful embedded devices such as wireless routers and GPS navigation systems. Here are some of the reasons:
    - Ports to common embedded chip sets are available.
    - They permit re-use of publicly available code for device drivers, web servers, firewalls, and other code.
    - Development systems can start out with broad feature-sets, and then the distribution can be configured to exclude unneeded functionality, and save the expense of the memory that it would consume.
    - Many engineers believe that running application code in user mode is more reliable and easier to debug, thus making the development process easier and the code more portable.
    - Features requiring faster response that can be guaranteed can often be placed in hardware.

Agenda

- Embedded Software Architectures
- Real time operating systems (RTOS)

General OS model (Linux-like)



Operating systems

- The operating system controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated.
  - how processes communicate.
- The most important resource is the CPU itself.
  - CPU access controlled by the scheduler.

RTOS and GPOS

- 相似的功能
  - 多任务级别
  - 软件和硬件资源管理
- 为应用提供基本的OS服务
- 从软件应用抽象硬件

RTOS从GPOS中分离出来的不同功能

- 嵌入式应用上下文中具有更好的可靠性
- 满足应用需要的剪裁能力
- 更快的特性
- 减少内存需求
- 为实时嵌入式系统提供可剪裁的调度策略
- 支持无盘化嵌入式系统，允许从ROM或RAM上引导和运行
- 对不同硬件平台具有更好的可移植性

Real-time operating system (RTOS) features

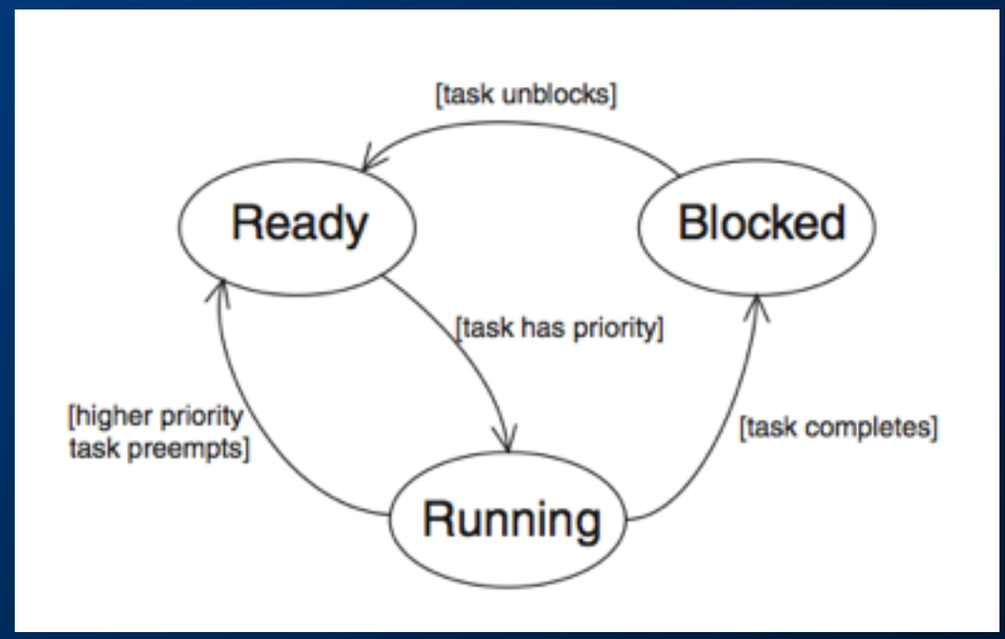
- reliability
- predictability, deterministic
- performance
- compactness
- scalability

Commercial RTOSs (partial)

- AMX (KADAK)
- C Executive (JMI Software)
- RTX (CMX Systems)
- eCos (Red Hat)
- INTEGRITY (Green Hills Software)
- LynxOS (LynuxWorks)
- µC/OS-II (Micrium)
- Neutrino (QNX Software Systems)
- Nucleus (Mentor Graphics)
- POSIX (IEEE Standard)
- FreeRTOS.org
- RTOS-32 (OnTime Software)
- OS-9 (Microware)
- OSE (OSE Systems)
- pSOSystem (Wind River)
- QNX (QNX Software Systems)
- Quadros (RTXC)
- RTEMS (OAR)
- ThreadX (Express Logic)
- Linux/RT (TimeSys)
- VRTX (Mentor Graphics)
- VxWorks (Wind River)

Real-time Operating System (RTOS)

- Why use one?
  - flexibility
  - response time



- The elemental component of a real-time operating system is a task, and it's straightforward to add new tasks or delete obsolete ones because there is no main loop: The RTOS schedules when each task is to run based on its priority.
- The part of the RTOS called a scheduler keeps track of the state of each task, and decides which one should be running.

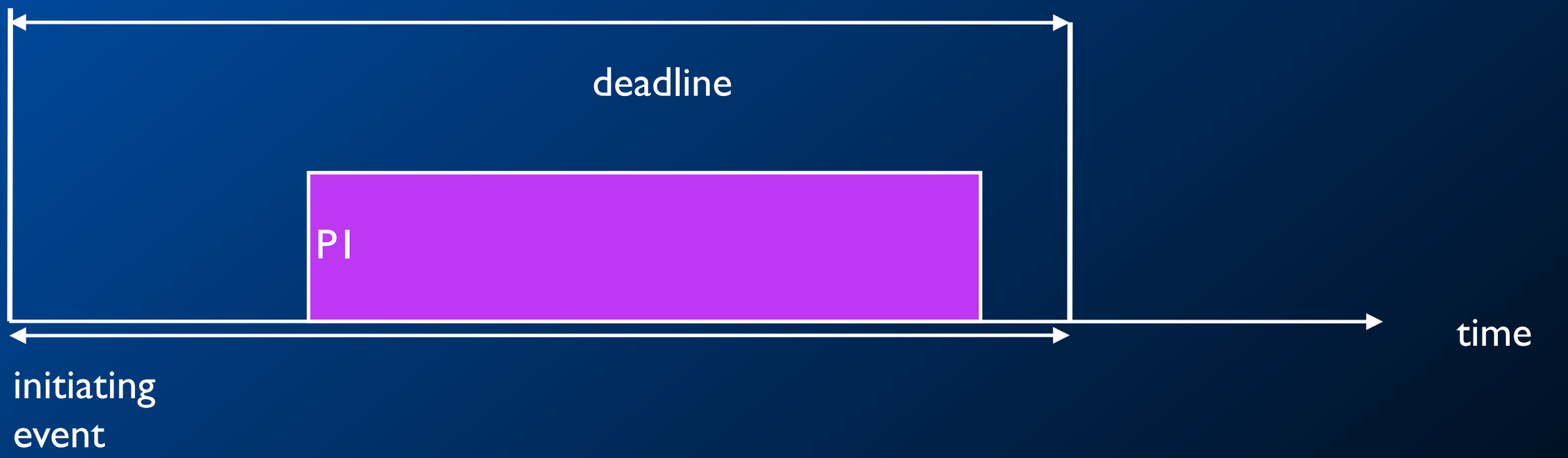
Real-time systems

- Perform a computation to conform to external timing constraints.
- Deadline frequency:
  - Periodic.
  - Aperiodic.
- Deadline type:
  - Hard: failure to meet deadline causes system failure.
  - Soft: failure to meet deadline causes degraded response.
  - Firm: late response is useless but some late responses can be tolerated.

Timing specifications on processes

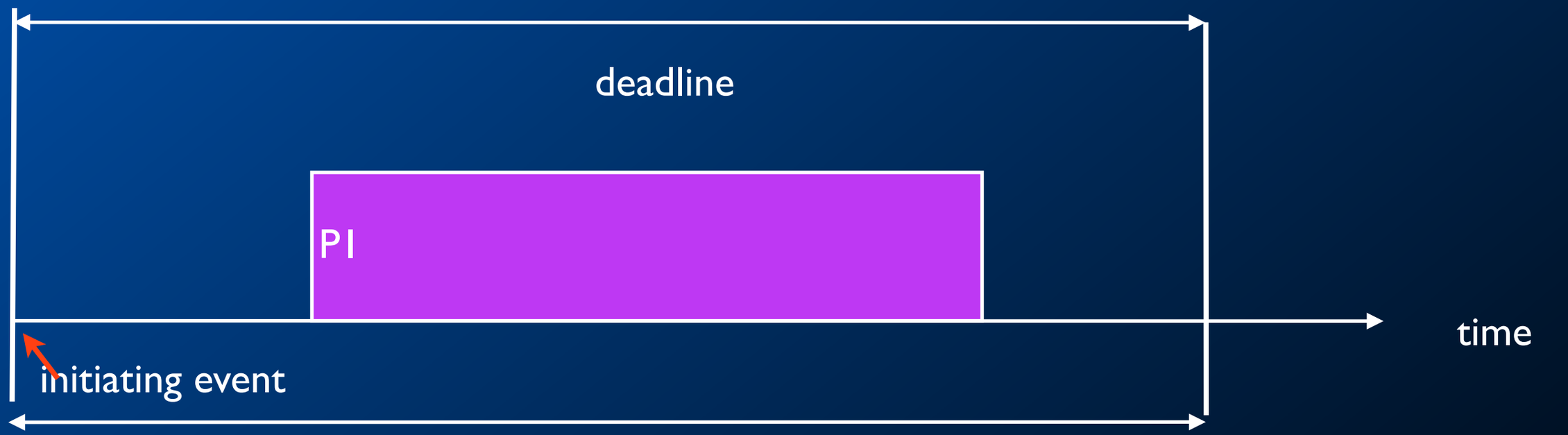
- Release time: time at which process becomes ready.
- Deadline: time at which process must finish.

Release times and deadlines



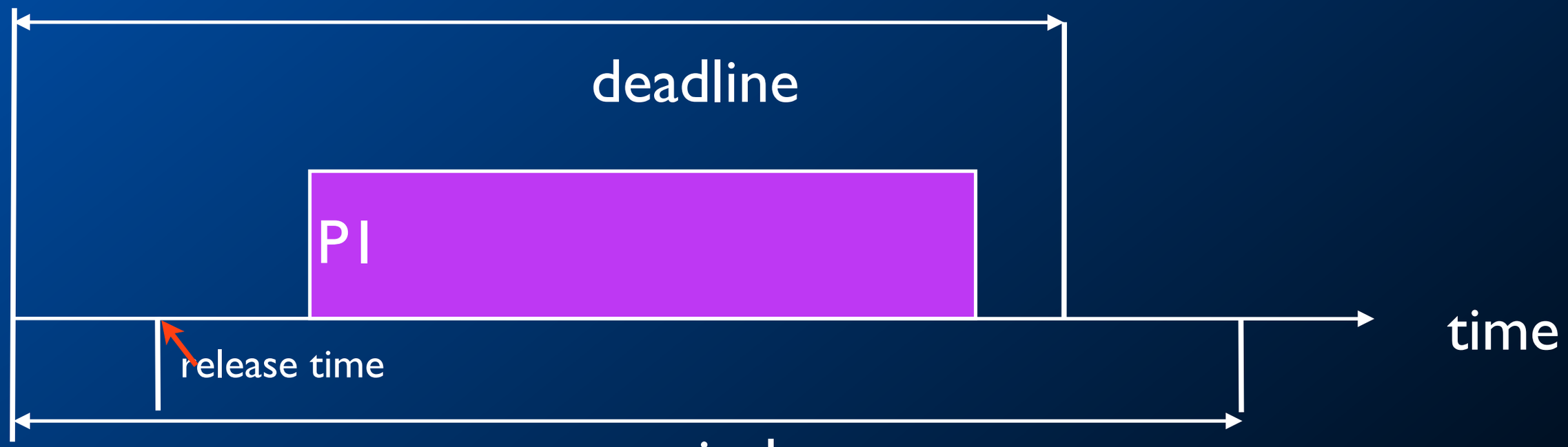
aperiodic process

Release times and deadlines



periodic process initiated
at start of period

Release times and deadlines



periodic process initiated by event

Rate requirements on processes

- Period: interval between process activations.
- Rate: reciprocal of period.
- Initiation rate may be higher than period---several copies of process run at once.

Timing violations

- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

Example: Space Shuttle software error

- Space Shuttle's first launch was delayed by a software timing error:
  - Primary control system PASS and backup system BFS.
  - BFS failed to synchronize with PASS.
    - Change to one routine added delay that threw off start time calculation.
    - 1 in 67 chance of timing problem.

Process execution characteristics

- Process execution time Ti.
  - Execution time in absence of preemption.
  - Possible time units: seconds, clock cycles.
  - Worst-case, best-case execution time may be useful in some cases.
- Sources of variation:
  - Data dependencies.
  - Memory system.
  - CPU pipeline.

Utilization

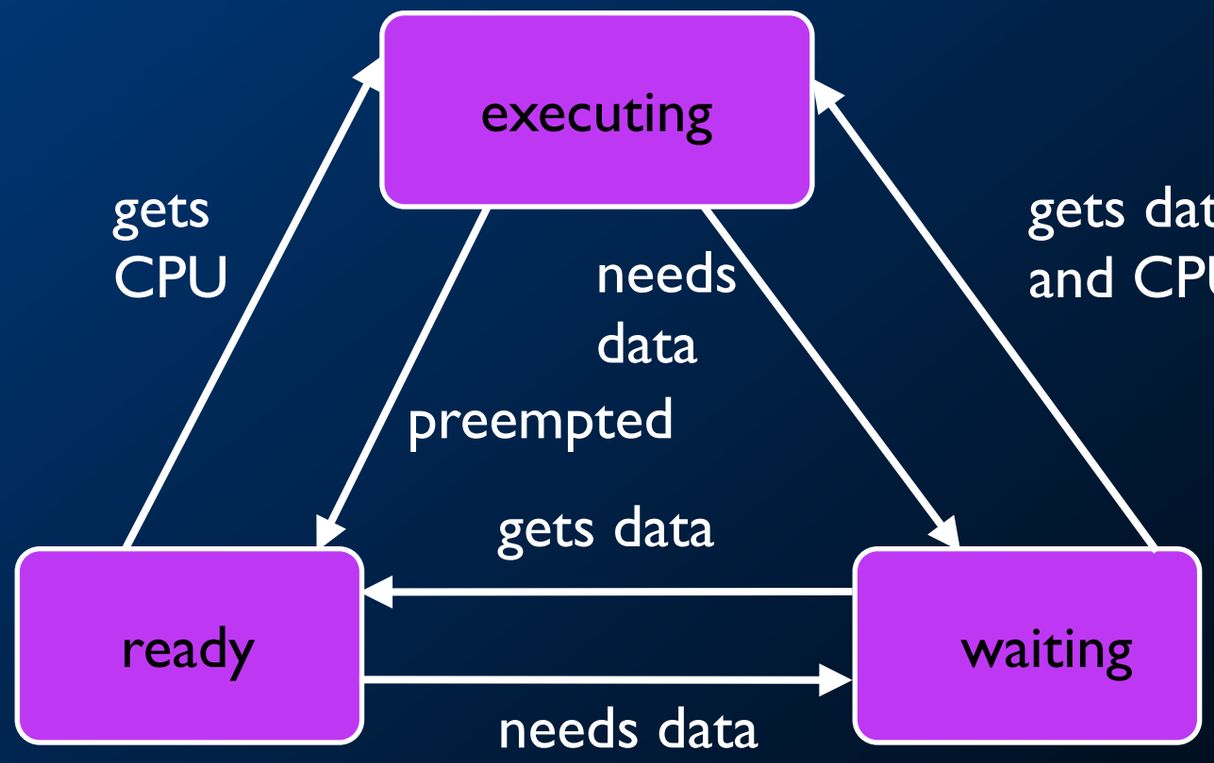
- CPU utilization:
  - Fraction of the CPU that is doing useful work.
  - Often calculated assuming no scheduling overhead.
- Utilization:
  - U = (CPU time for useful work)/ (total available CPU time)

    $= [\Sigma_{t1 \le t \le t2} T(t)] / [t2 - t1]$

    $= T/t$

State of a process

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.



The scheduling problem

- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?

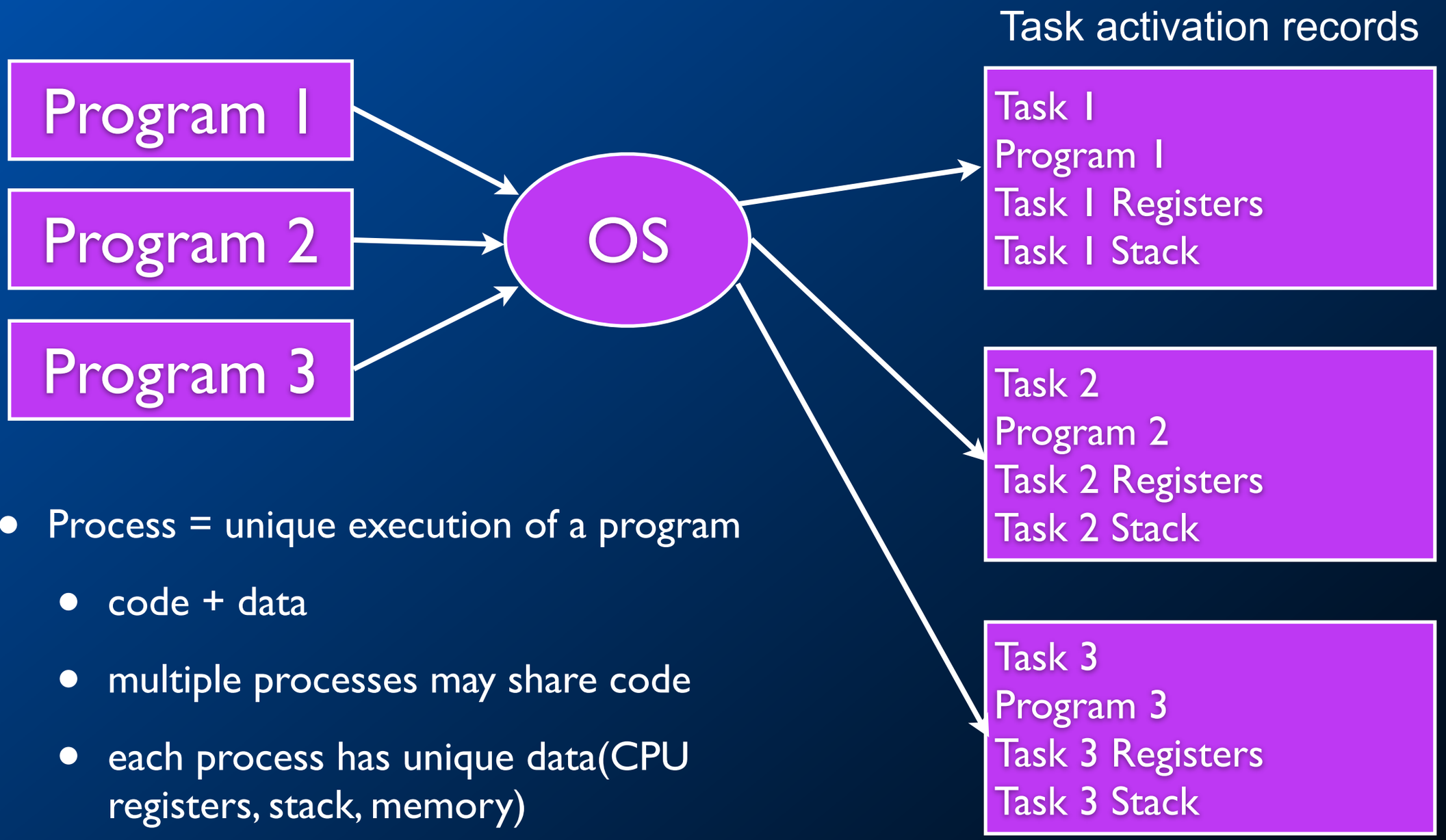
Embedded vs. general-purpose scheduling

- Workstations try to avoid starving processes of CPU access.
  - Fairness = access to CPU.
- Embedded systems must meet deadlines.
  - Low-priority processes may not run for a long time.

OS process management

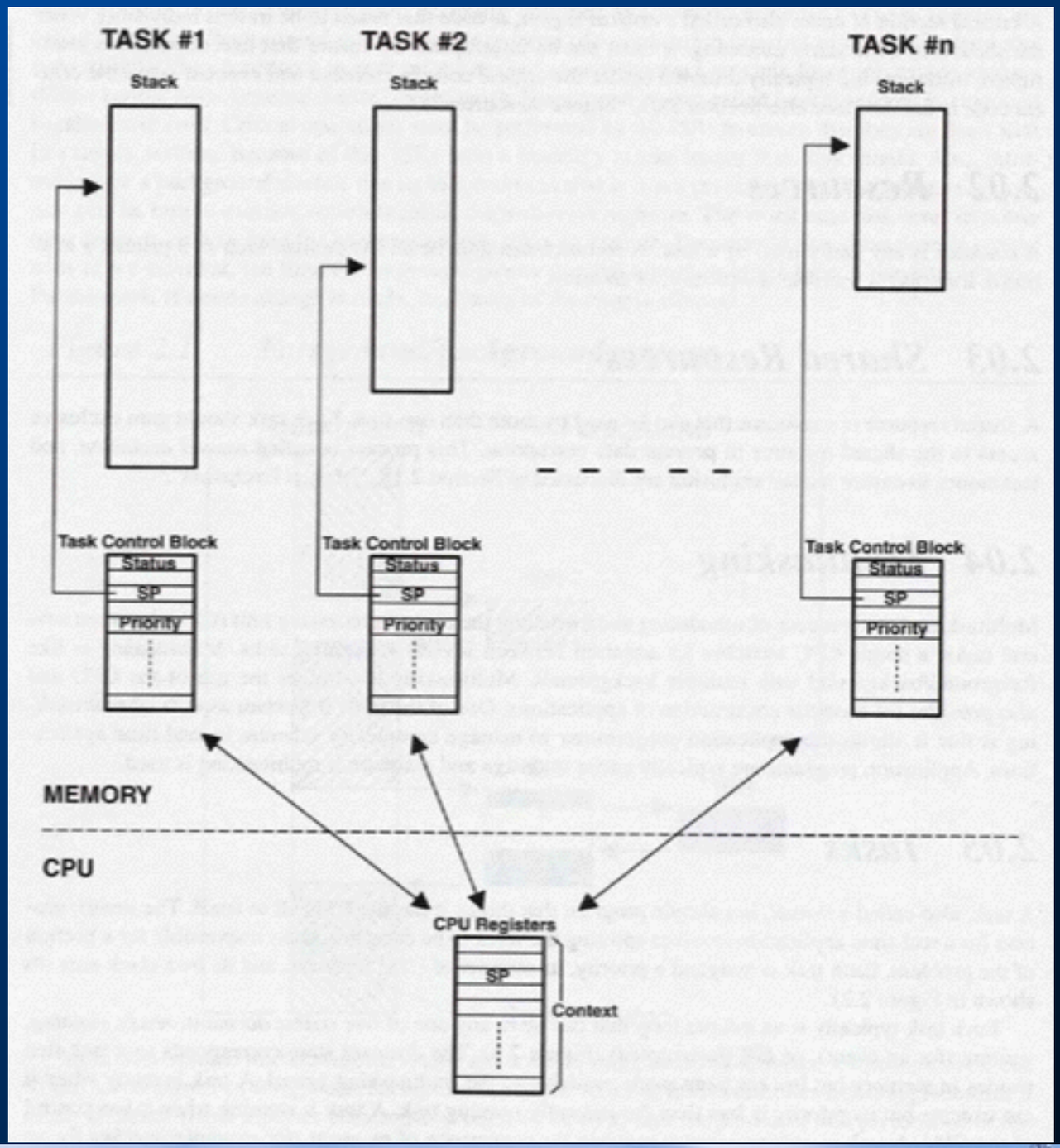
- OS needs to keep track of:
  - process priorities;
  - scheduling state;
  - process activation record.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.
    - Example: incoming telephone call

Multitasking OS

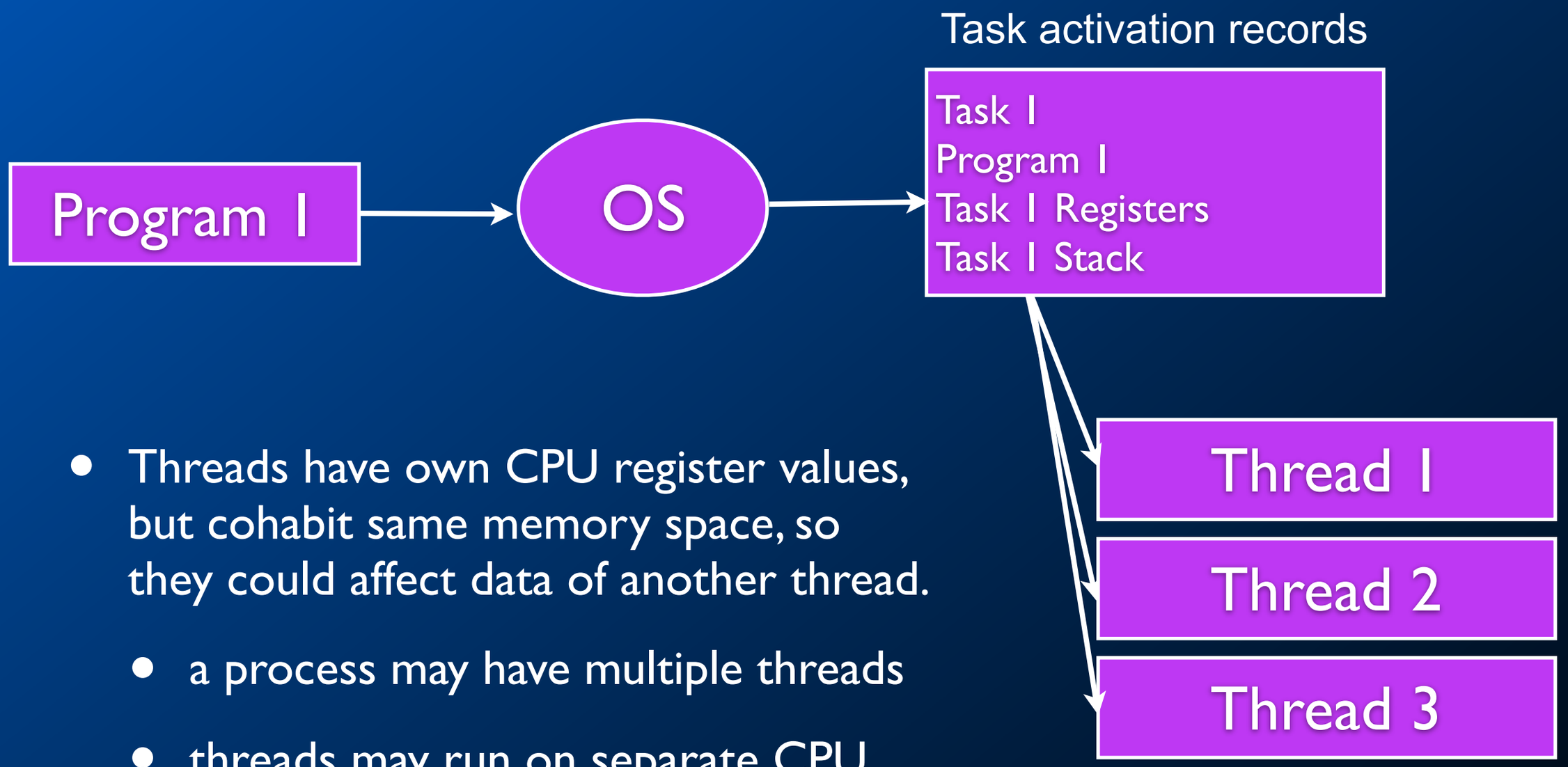


- Process = unique execution of a program
  - code + data
  - multiple processes may share code
  - each process has unique data(CPU registers, stack, memory)
  - process defined by its "activation record"

Multitasking OS



Process threads (lightweight processes)



- Threads have own CPU register values, but cohabit same memory space, so they could affect data of another thread.
  - a process may have multiple threads
  - threads may run on separate CPU cores

Typical process/task activation records (task control blocks)

- Task ID
- Task state (running, ready, blocked)
- Task priority
- Task starting address
- Task stack
- Task CPU registers
- Task data pointer
- Task time (ticks)

When can a new thread be dispatched?

- Under non-preemptive scheduling:
  - When the current thread completes.
- Under Preemptive scheduling:
  - Upon a timer interrupt
  - Upon an I/O interrupt (possibly)
  - When a new thread is created, or one completes.
  - When the current thread blocks on or releases a mutex
  - When the current thread blocks on a semaphore
  - When a semaphore state is changed
  - When the current thread makes any OS call
    - file system access
    - network access
    - ...

The Focus Today:
How to decide which thread to schedule?

Considerations:

- Preemptive vs. non-preemptive scheduling
- Periodic vs. aperiodic tasks
- Fixed priority vs. dynamic priority
- Priority inversion anomalies
- Other scheduling anomalies

Metrics

- How do we evaluate a scheduling policy?
  - Ability to satisfy all deadlines.
  - CPU utilization---percentage of time devoted to useful work.
  - Scheduling overhead---time required to make scheduling decision.
  - lateness.
  - total completion time.

Preemptive Scheduling

Assume all threads have priorities

- either statically assigned (constant for the duration of the thread)
- or dynamically assigned (can vary).

Assume further that the kernel keeps track of which threads are "enabled" (able to execute, e.g. not blocked waiting for a semaphore or a mutex or for a time to expire).

Preemptive scheduling:

- At any instant, the enabled thread with the highest priority is executing.
- Whenever any thread changes priority or enabled status, the kernel can dispatch a new thread.

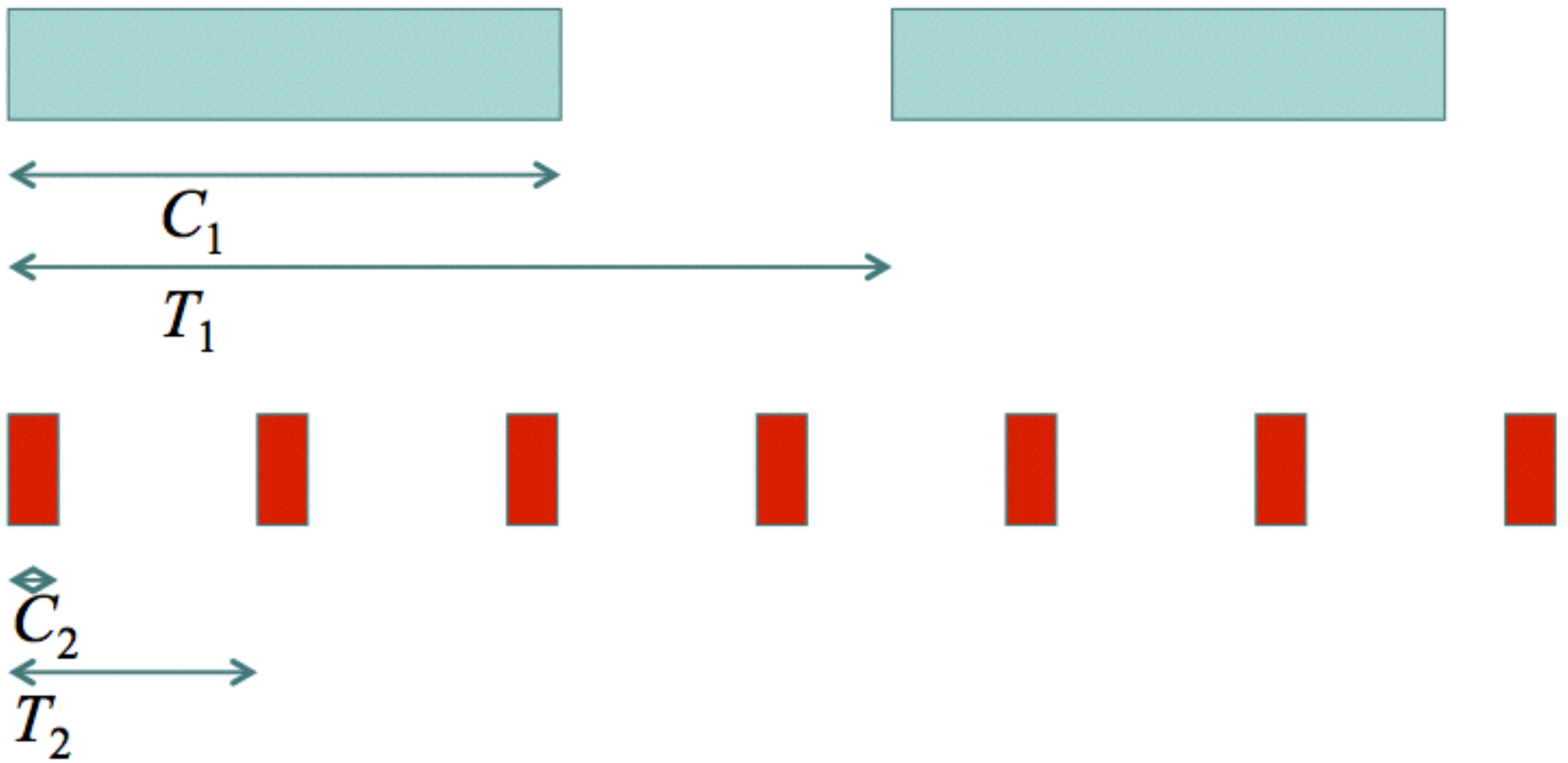
Rate Monotonic Scheduling

- Assume n tasks invoked periodically with:
  - periods T1, ... ,Tn (impose real-time constraints)
  - All tasks are independent.
  - worst-case execution times (WCET) C1, ... ,Cn
    - assumes no mutexes, semaphores, or blocking I/O
  - no precedence constraints
    - fixed priorities
    - The time required for context switching is negligible
    - preemptive scheduling
- Theorem: If any priority assignment yields a feasible schedule, then priorities ordered by period (smallest period has the highest priority) also yields a feasible schedule.
- RMS is optimal in the sense of feasibility.
- Liu and Leland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 20(1), 1973.

Feasibility for RMS

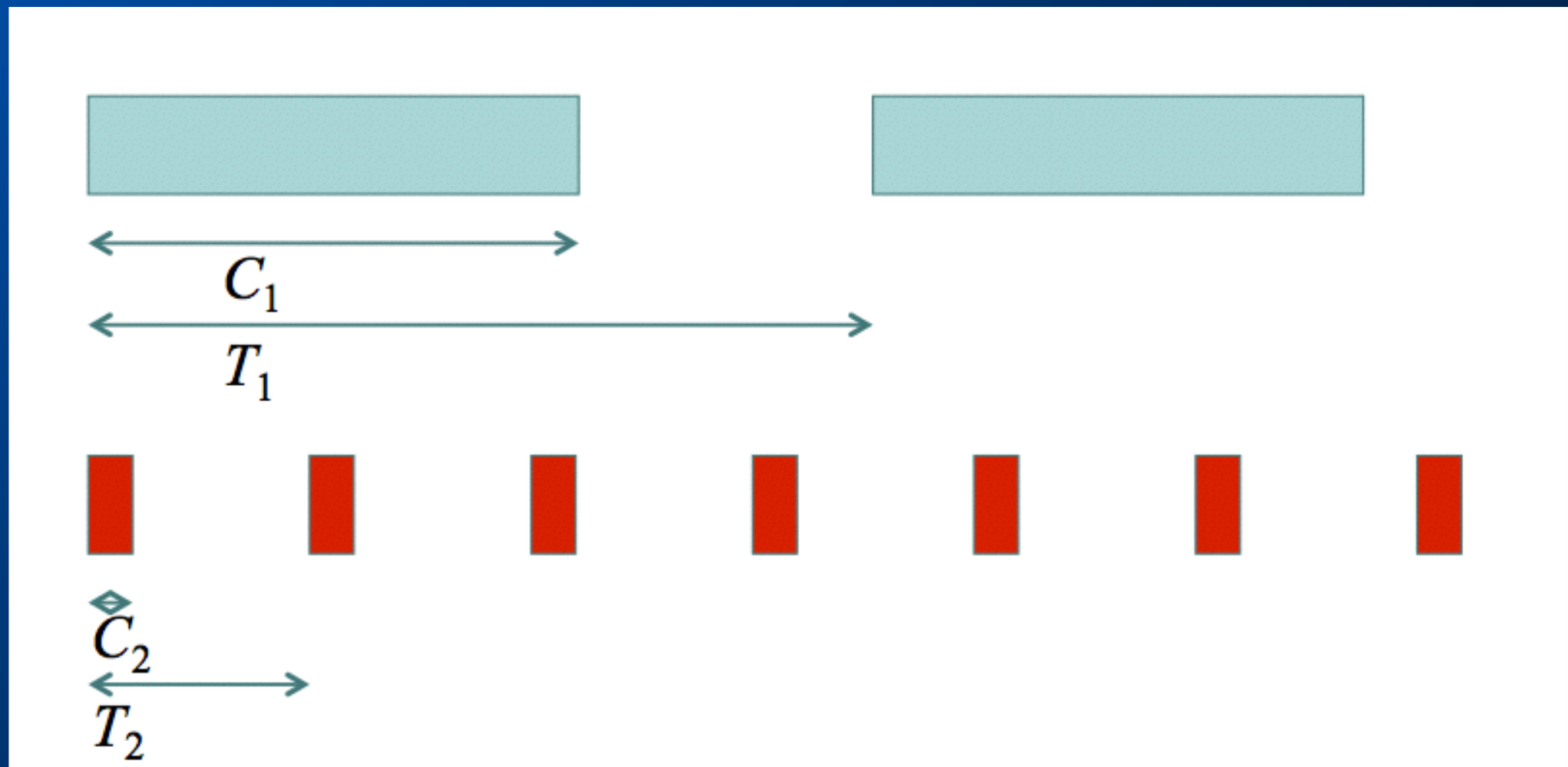
- Feasibility is defined for RMS to mean that every task executes to completion once within its designated period.

Showing Optimality of RMS: Consider two tasks with different periods



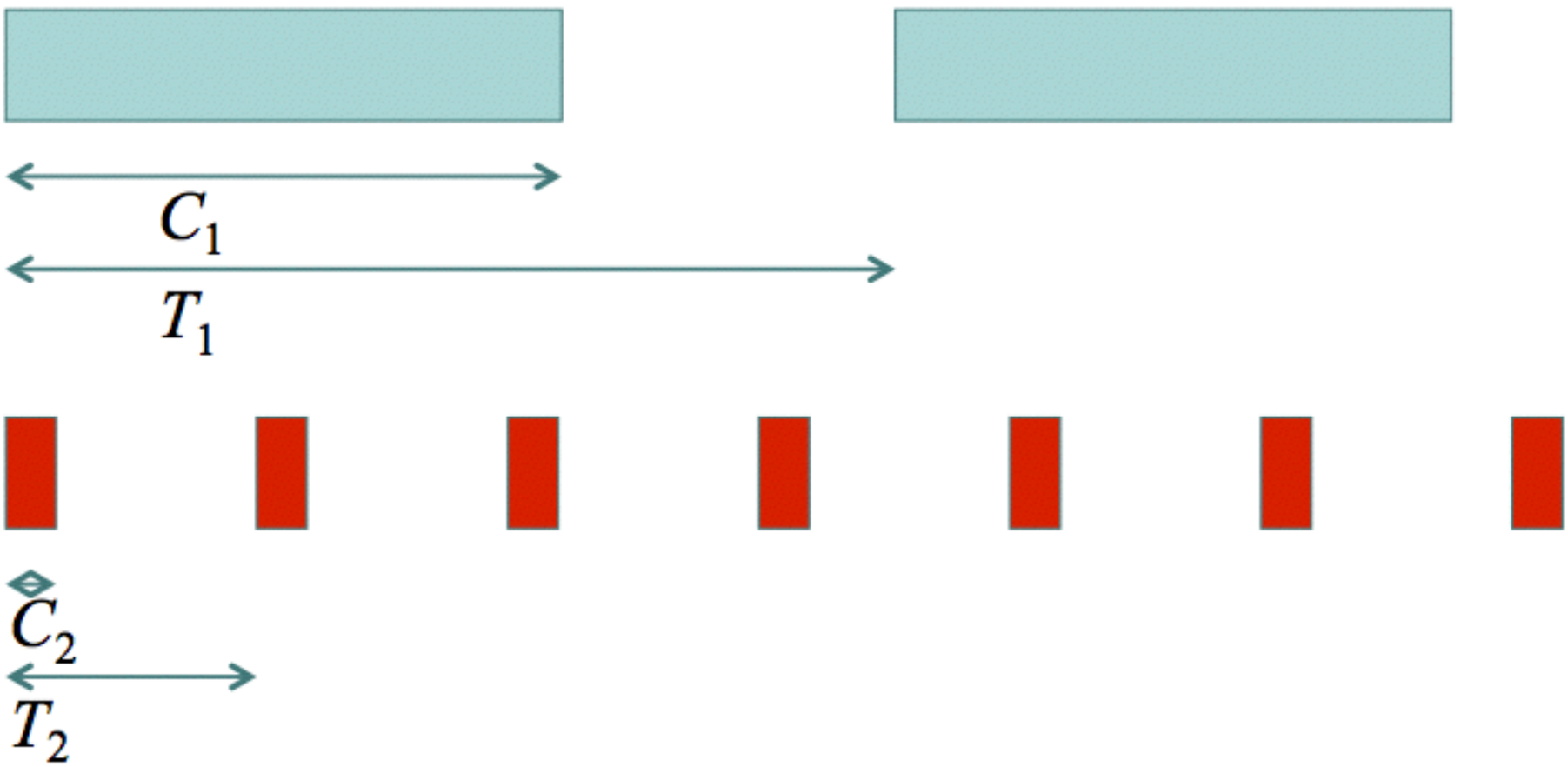
Is a non-preemptive schedule feasible?

Showing Optimality of RMS: Consider two tasks with different periods



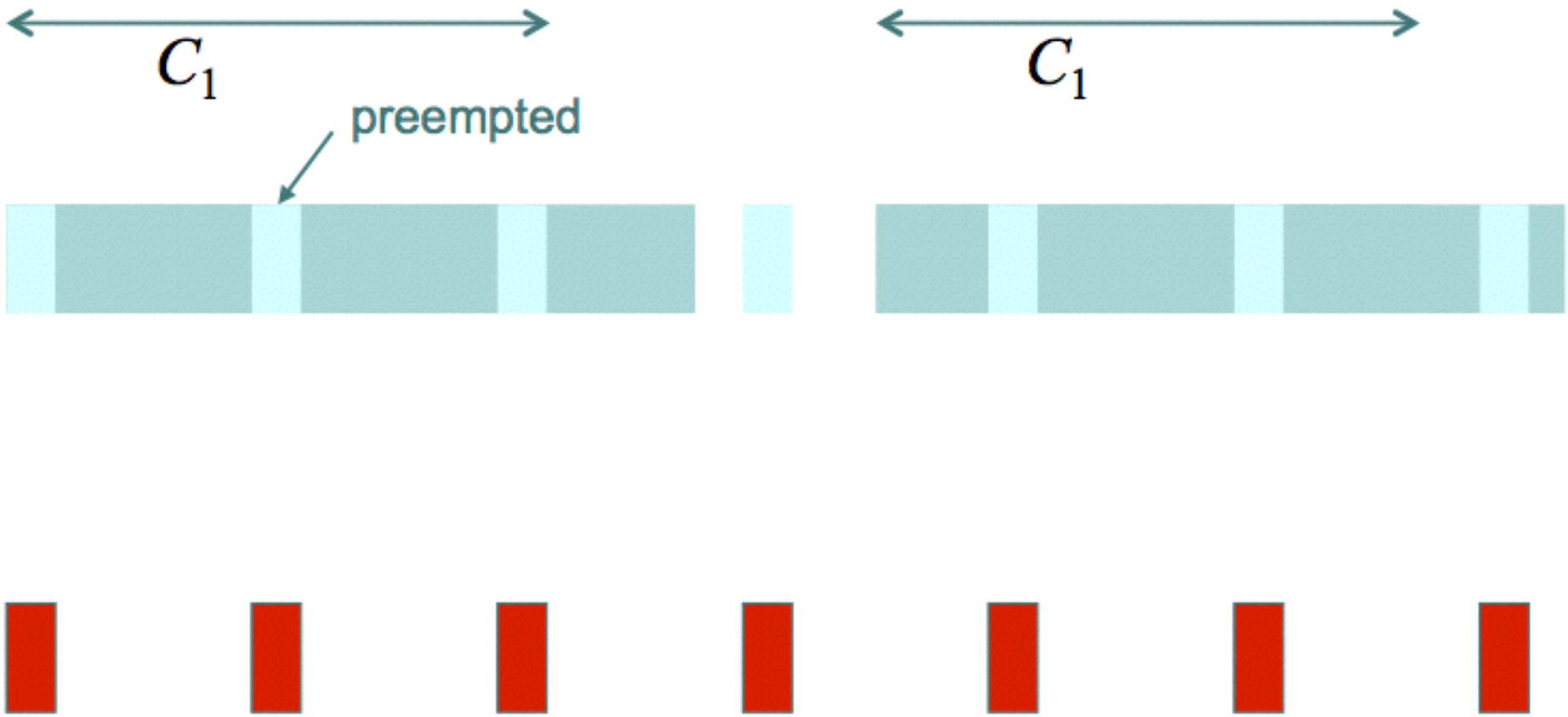
Non-preemptive schedule is not feasible. Some instance of the Red Task (2) will not finish within its period if we do non-preemptive scheduling.

Showing Optimality of RMS: Consider two tasks with different periods



What if we had a preemptive scheduling with higher priority for red task?

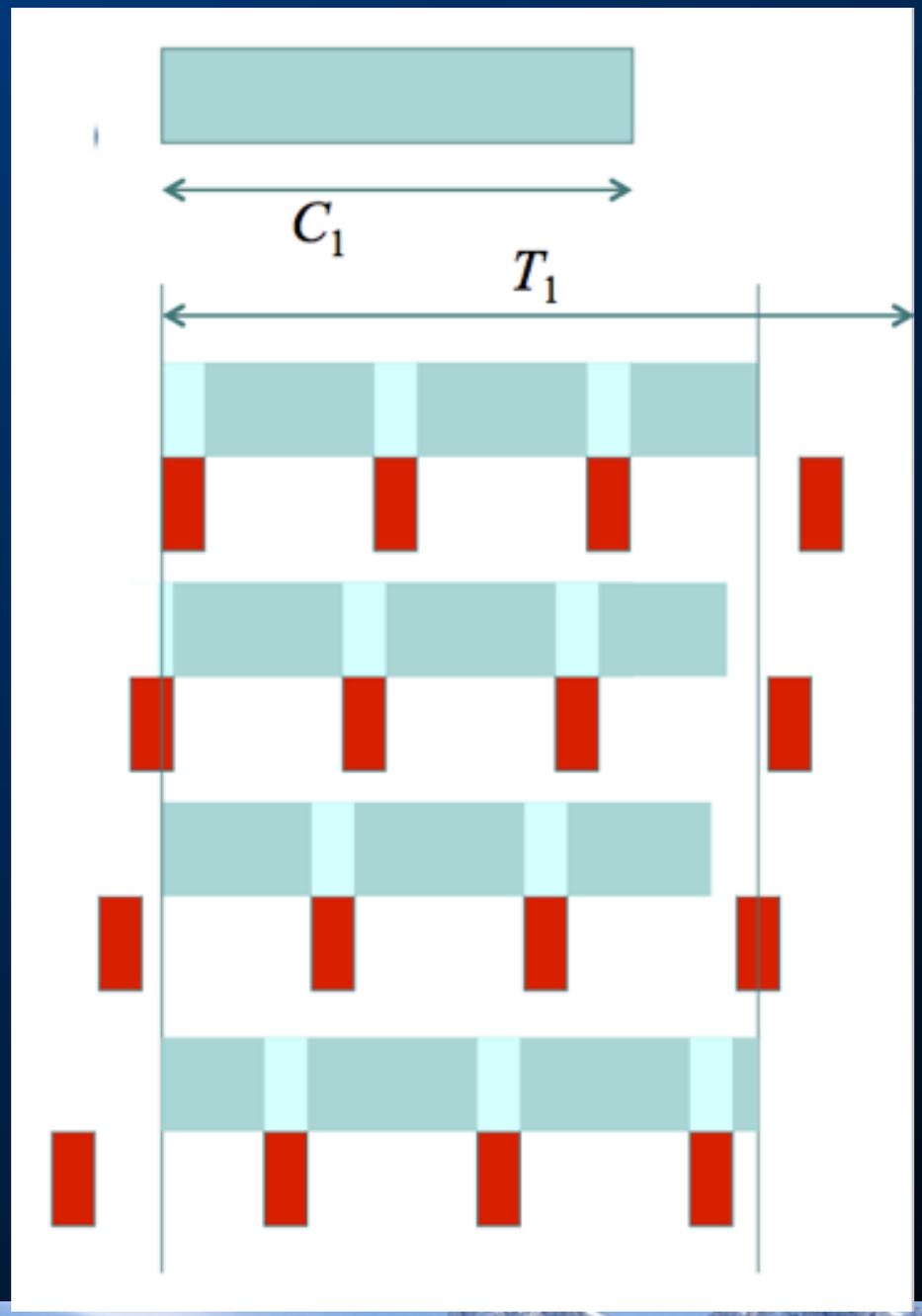
Showing Optimality of RMS: Consider two tasks with different periods



Preemptive schedule with the red task having higher priority is feasible.
Note that preemption of the blue task extends its completion time.

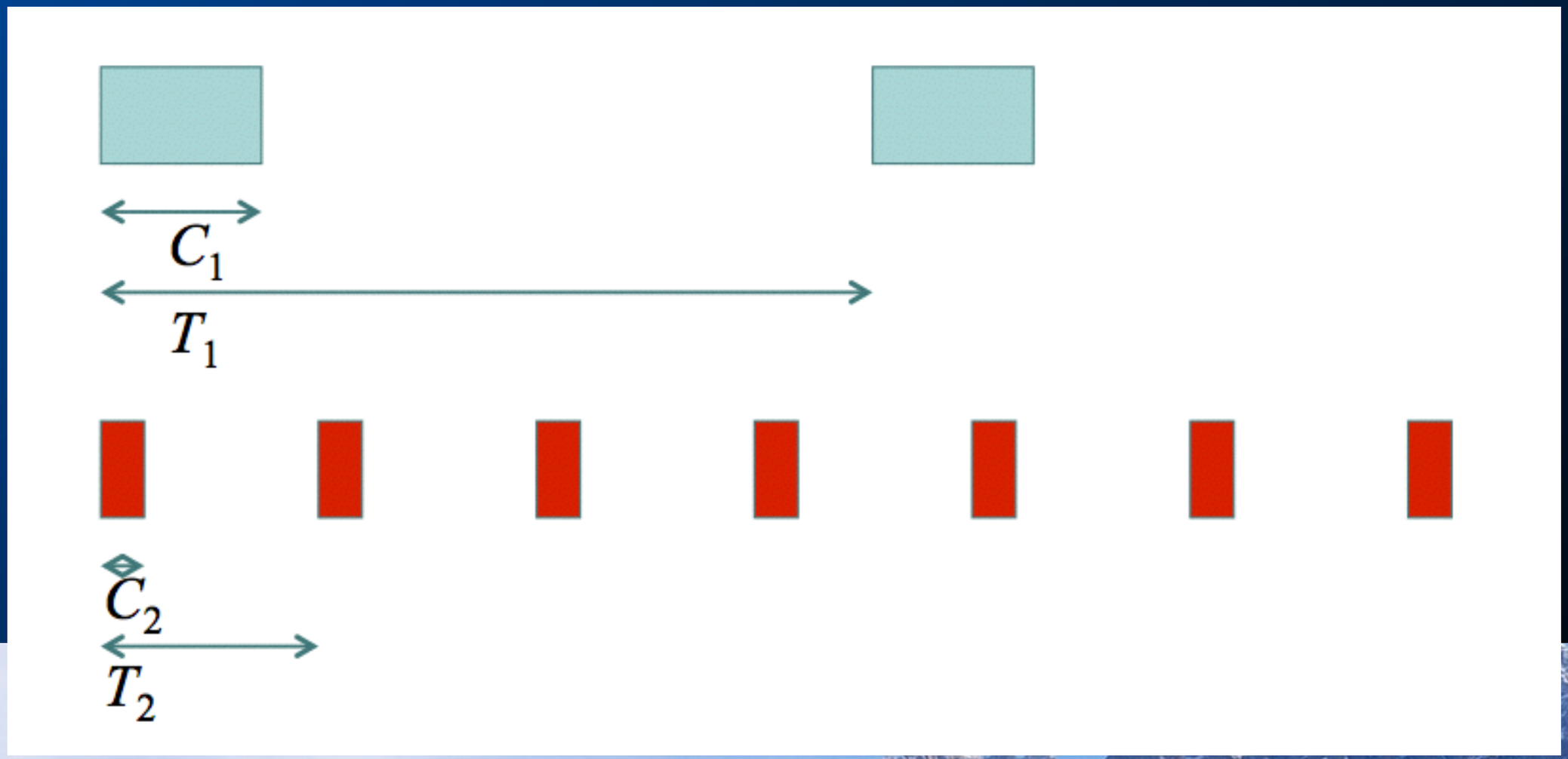
Showing Optimality of RMS: Alignment of tasks

- Completion time of the lower priority task is worst when its starting phase matches that of higher priority tasks.
- Thus, when checking schedule feasibility, it is sufficient to consider only the worst case: All tasks start their cycles at the same time.



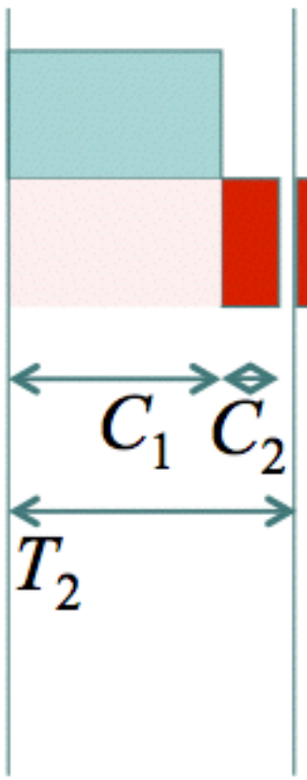
Showing Optimality of RMS: (for two tasks)

- It is sufficient to show that if a non-RMS schedule is feasible, then the RMS schedule is feasible.
- Consider two tasks as follows:



Showing Optimality of RMS: (for two tasks)

- The non-RMS, fixed priority schedule looks like this:



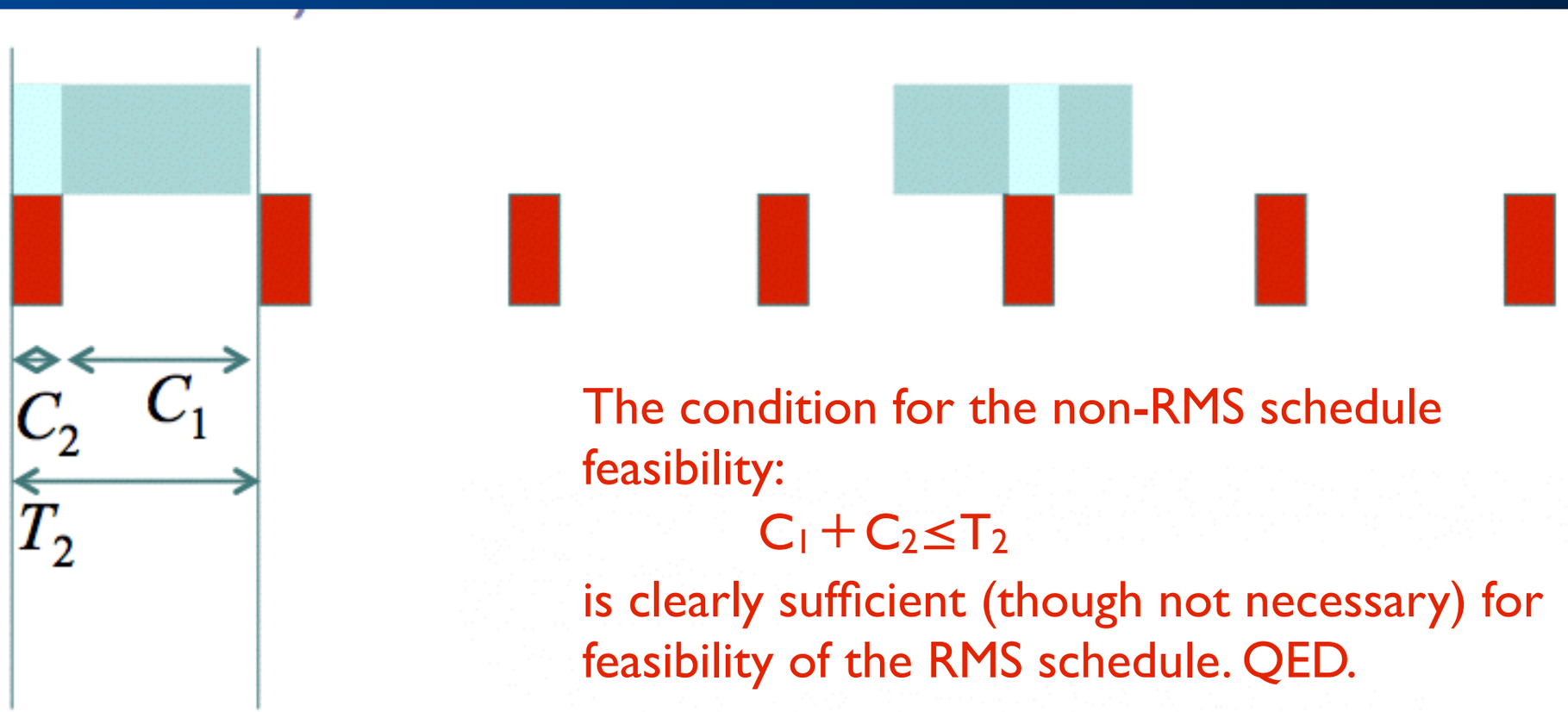
From this, we can see that the non-RMS schedule is feasible if and only if

$$C_1 + C_2 \leq T_2$$

We can then show that this condition implies that the RMS schedule is feasible.

Showing Optimality of RMS: (for two tasks)

- The RMS schedule looks like this: (task with smaller period moves earlier)



The condition for the non-RMS schedule feasibility:

$$C_1 + C_2 \leq T_2$$

is clearly sufficient (though not necessary) for feasibility of the RMS schedule. QED.

Comments

- This proof can be extended to an arbitrary number of tasks (though it gets much more tedious).
- This proof gives optimality only w.r.t. feasibility. It says nothing about other optimality criteria.
- Practical implementation:
  - Timer interrupt at greatest common divisor of the periods
  - Multiple timers

RMS

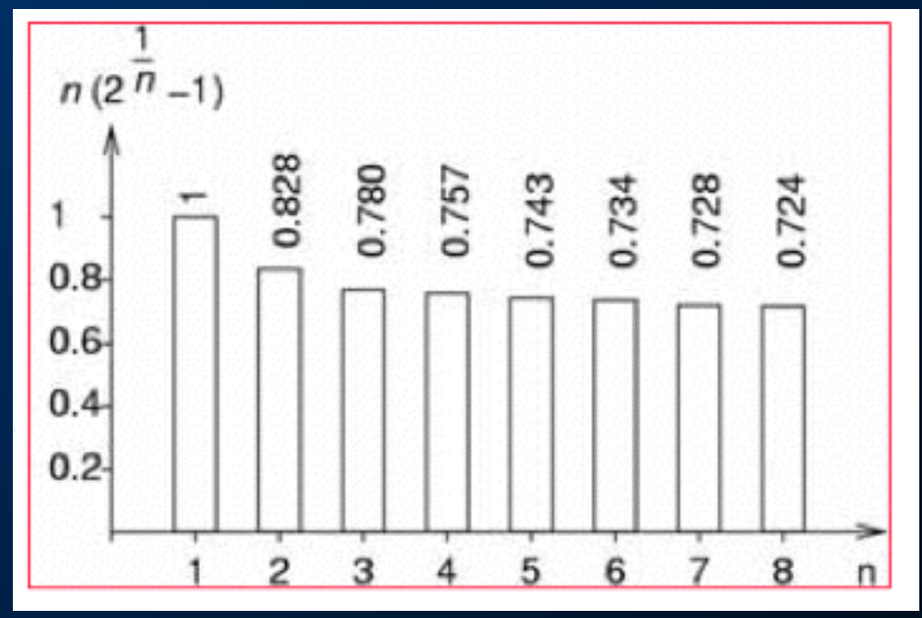
- Schedulability analysis: A set of periodic tasks is schedulable with RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

- This condition is sufficient but not necessary.
- The term

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- denotes the processor utilization factor U which is the fraction of processor time spent in the execution of the task set.



Deadline Driven Scheduling: 1. Jackson's Algorithm: EDD (1955)

- Given n independent one-time tasks with deadlines $d_1, \ldots, d_n$, schedule them to minimize the maximum lateness, defined as

$$L_{\max} = \max_{1 \le i \le n} \{ f_i - d_i \}$$

  where f_i is the finishing time of task i. Note that this is negative iff all deadlines are met.

- Earliest Due Date (EDD) algorithm: Execute them in order of non-decreasing deadlines.

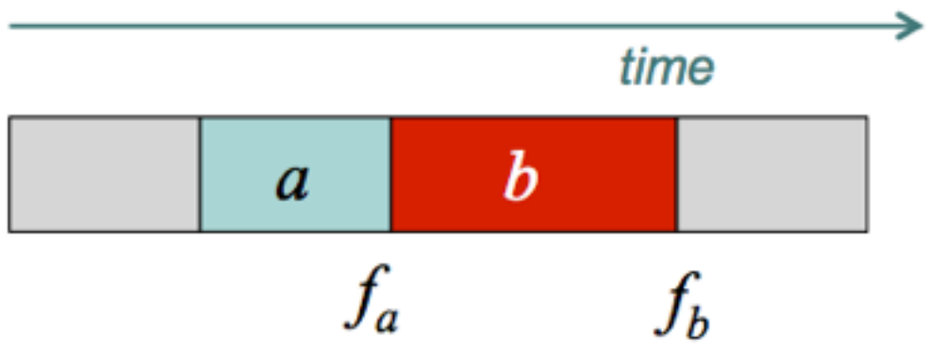
- Note that this does not require preemption.

Theorem: EDD is Optimal in the Sense of Minimizing Maximum Lateness

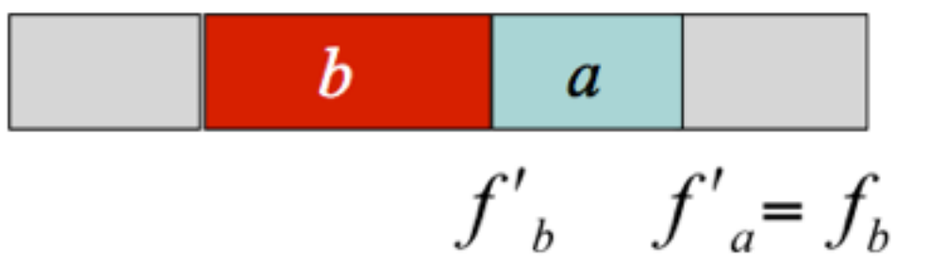
- To prove, use an interchange argument. Given a schedule *S* that is not EDD, there must be tasks *a* and *b* where *a* immediately precedes *b* in the schedule but $d_a > d_b$. Why?
- We can prove that this schedule can be improved by interchanging *a* and *b*. Thus, no non-EDD schedule is achieves smaller max lateness than EDD, so the EDD schedule must be optimal.

Consider a non-EDD Schedule *S*

- There must be tasks *a* and *b* where a immediately precedes *b* in the schedule but $d_a > d_b$



$$L_{\max} = \max\{f_a - d_a, f_b - d_b\} = f_b - d_b$$



$$L'_{\max} = \max\{f'_a - d_a, f'_b - d_b\}$$

Theorem: $L'_{max} \leq L_{max}$.
Hence, S' is no worse than S.

Case 1: $f'_a - d_a > f'_b - d_b$.
Then: $L'_{max} \leq f'_a - d_a = f_b - d_a \leq L_{max}$
(because: $d_a > d_b$).

Case 2: $f'_a - d_a \leq f'_b - d_b$.
Then: $L'_{max} \leq f'_b - d_b \leq L_{max}$
(because: $f'_b < f_b$).

Deadline Driven Scheduling: 1. Horn's algorithm: EDF (1974)

- Extend EDD by allowing tasks to "arrive" (become ready) at any time.
- Earliest deadline first (EDF): Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all arrived tasks is optimal w.r.t. minimizing the maximum lateness.
- Proof uses a similar interchange argument.

Using EDF for Periodic Tasks

- The EDF algorithm can be applied to periodic tasks as well as aperiodic tasks.
  - Simplest use: Deadline is the end of the period.
  - Alternative use: Separately specify deadline (relative to the period start time) and period.

RMS vs. EDF? Which one is better?

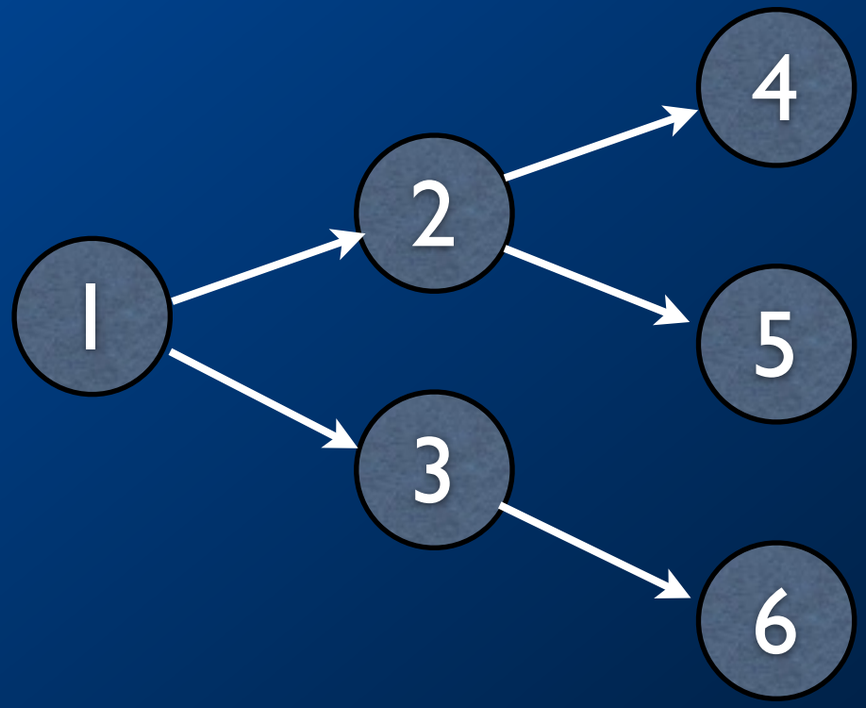
- What are the pros and cons of each?

Comparison of EDF and RMS

- Favoring RMS
  - Scheduling decisions are simpler (fixed priorities vs. the dynamic priorities required by EDF. EDF scheduler must maintain a list of ready tasks that is sorted by priority.)
- Favoring EDF
  - Since EDF is optimal w.r.t. maximum lateness, it is also optimal w.r.t. feasibility. RMS is only optimal w.r.t. feasibility. For infeasible schedules, RMS completely blocks lower priority tasks, resulting in unbounded maximum lateness.
  - EDF can achieve full utilization where RMS fails to do that
  - EDF results in fewer preemptions in practice, and hence less overhead for context switching.
  - Deadlines can be different from the period.

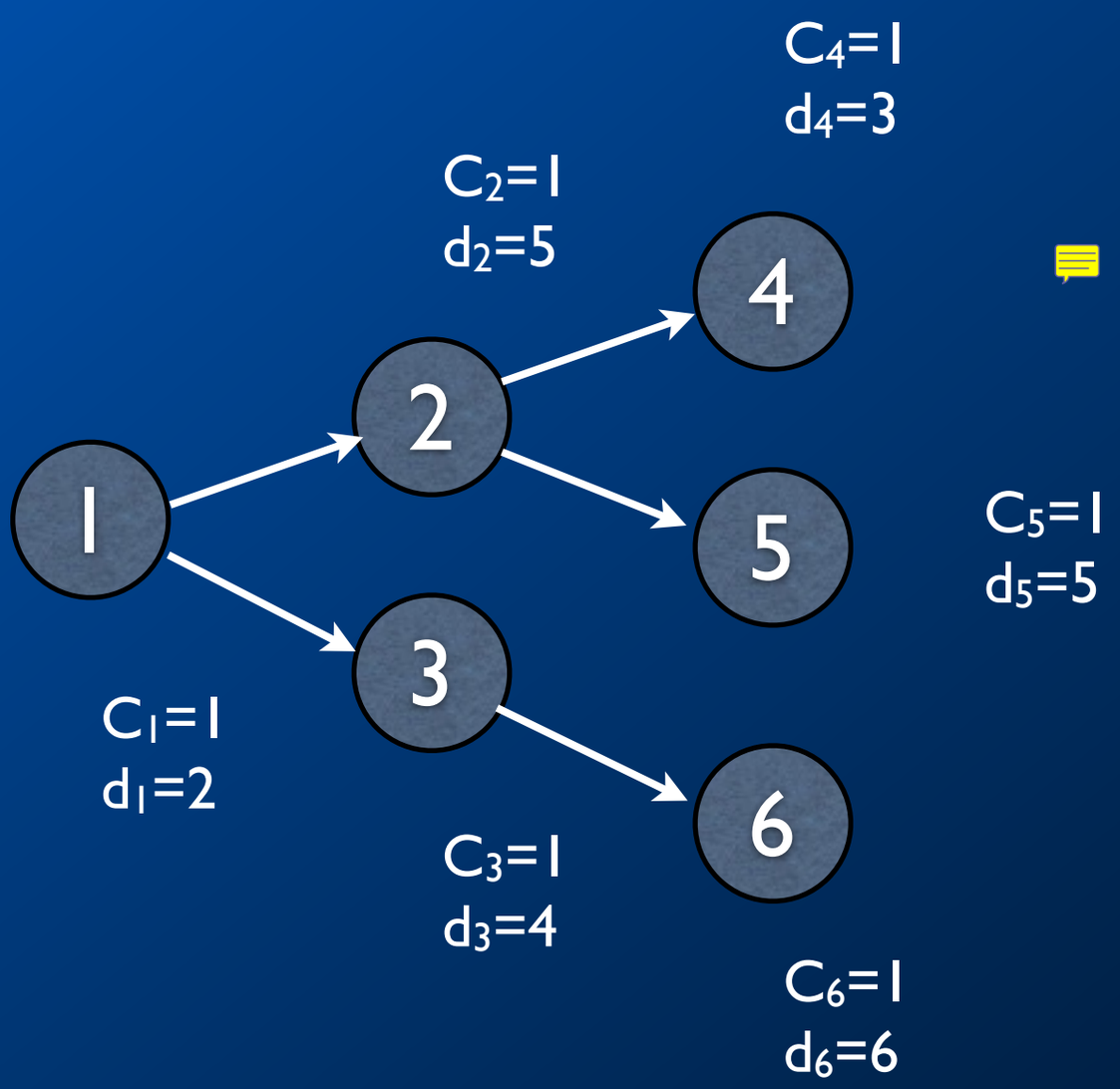
Precedence Constraints

- A directed acyclic graph (DAG) shows precedences, which indicate which tasks must complete before other tasks start.



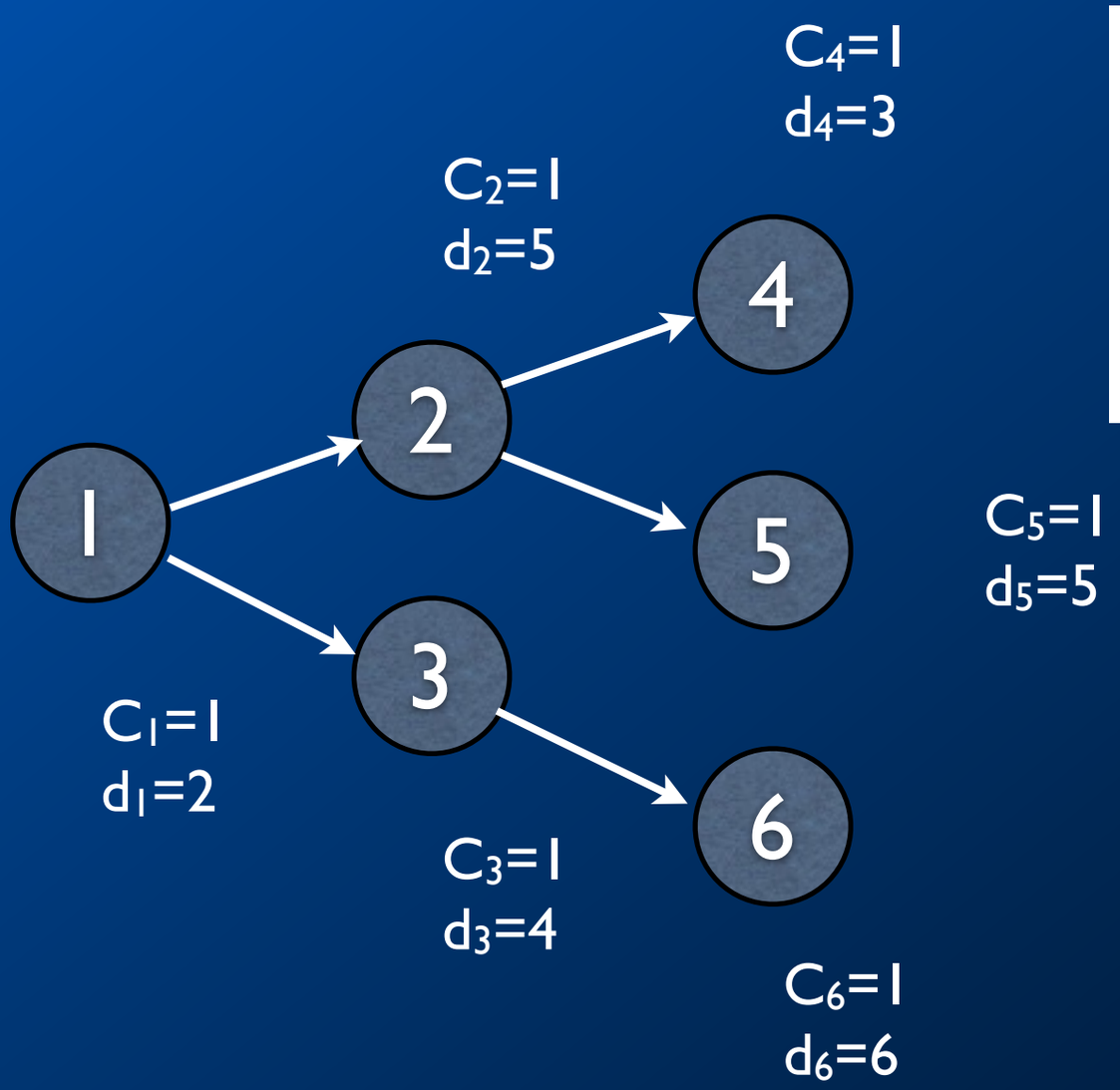
DAG, showing that task 1 must complete before tasks 2 and 3 can be started, etc.

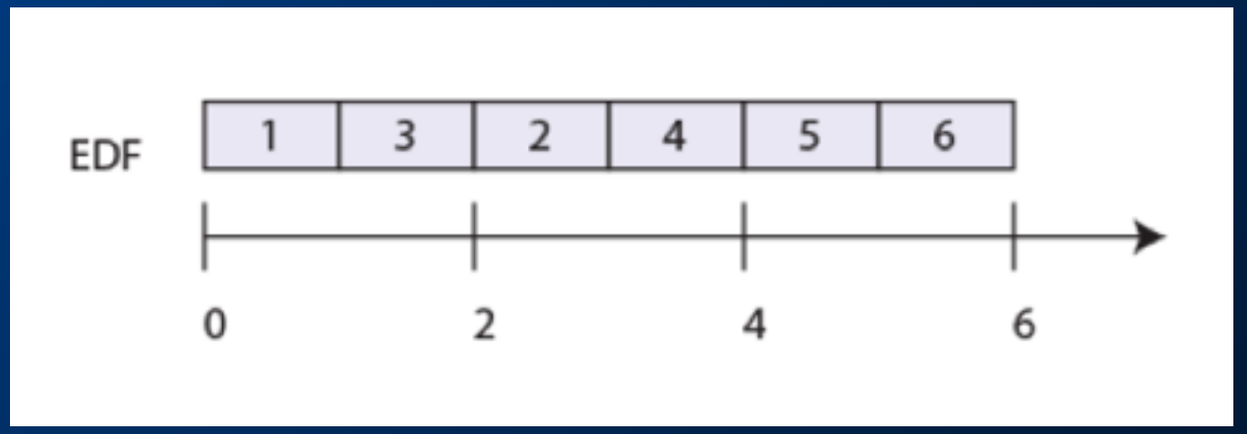
Example: EDF Schedule



Is this feasible? Is it optimal?

EDF is not optimal under precedence constraints

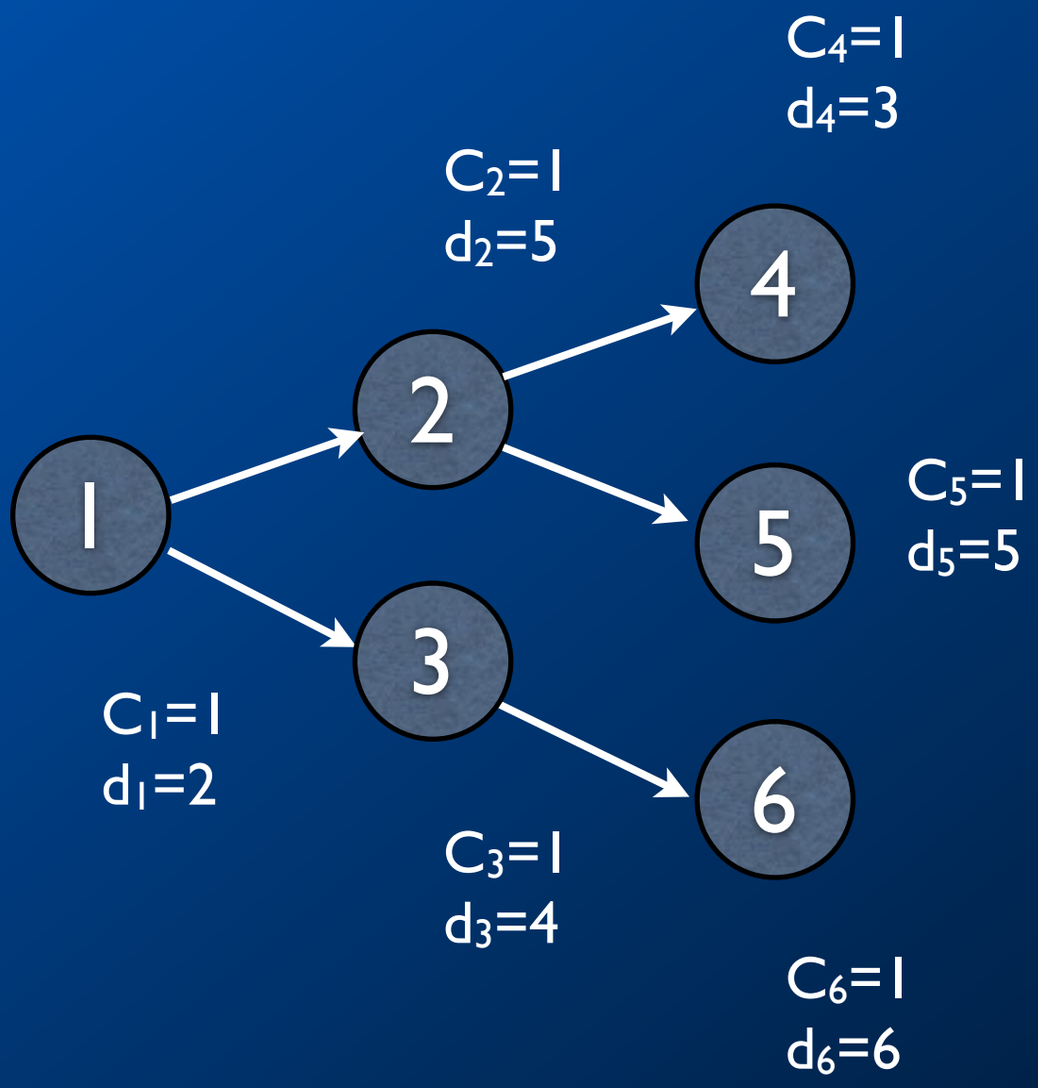




The EDF schedule chooses task 3 at time 1 because it has an earlier deadline. This choice results in task 4 missing its deadline.

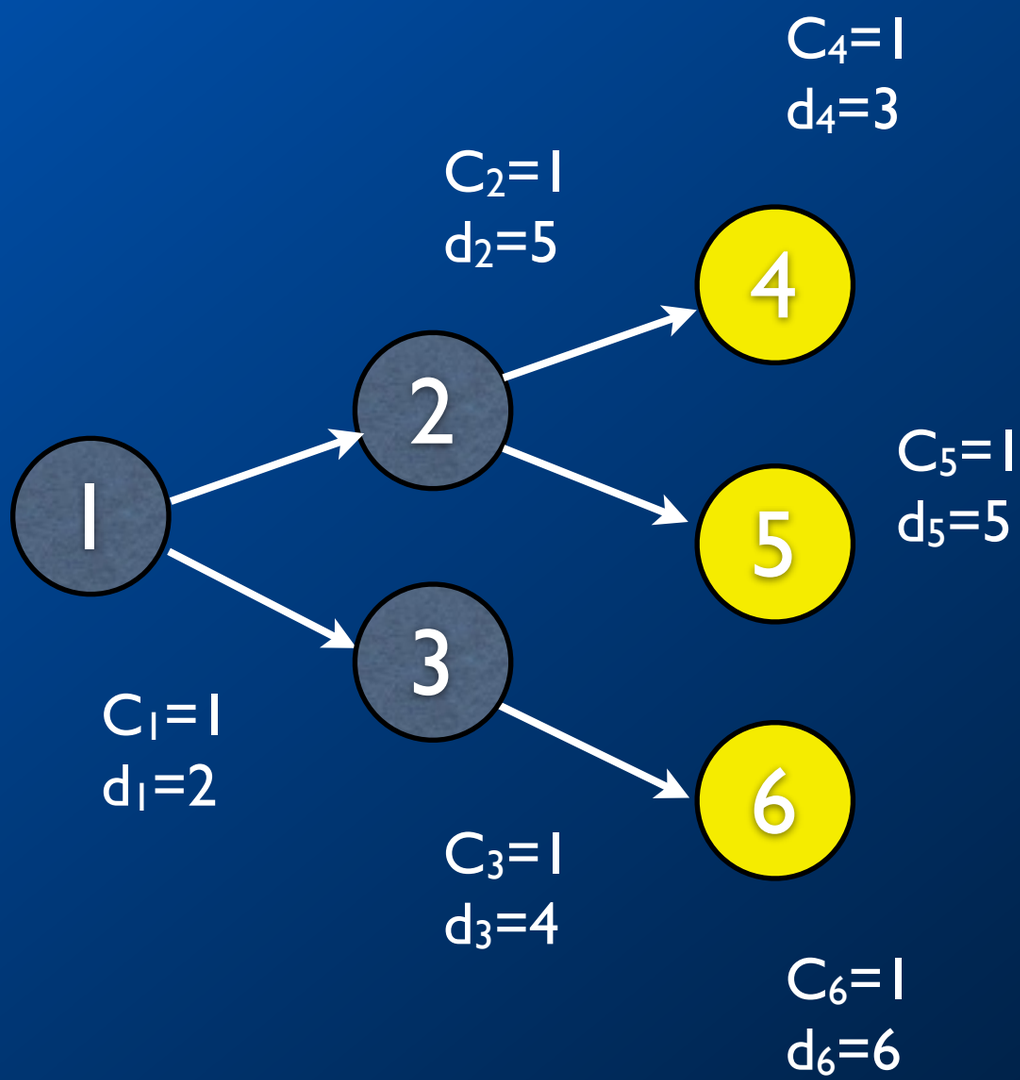
Is there a feasible schedule?

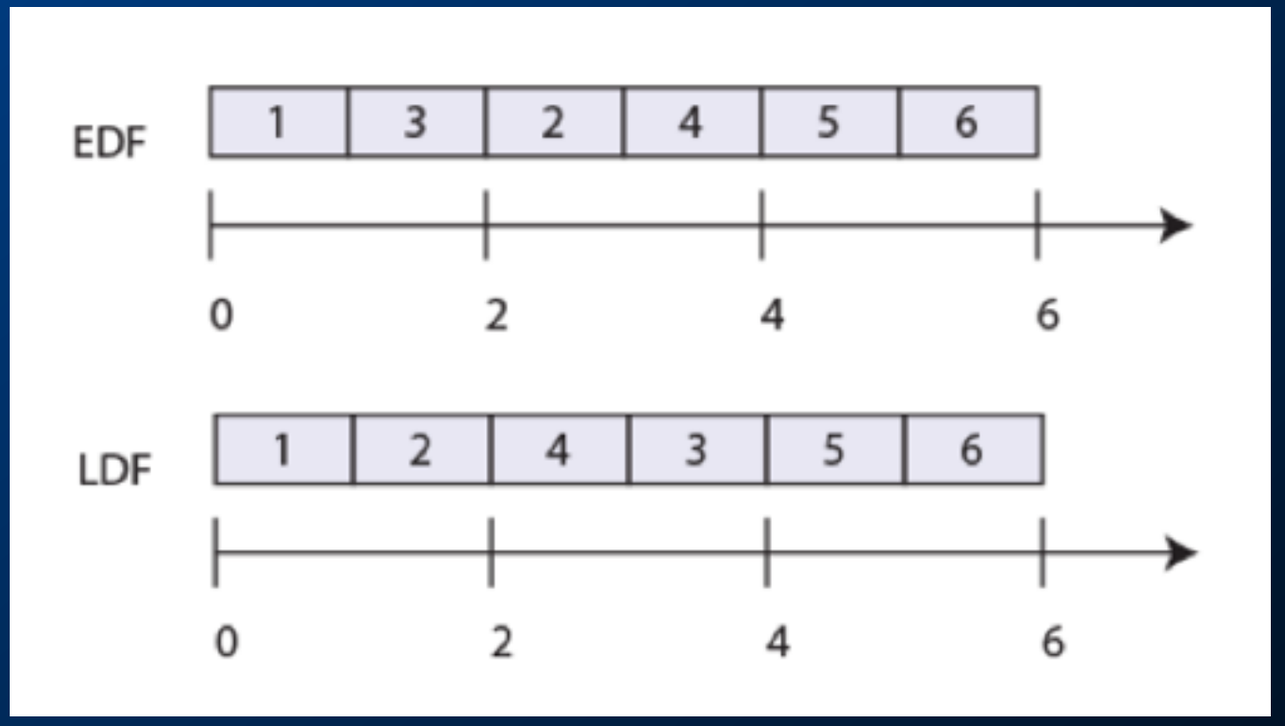
LDF is optimal under precedence constraints



The LDF schedule shown at the bottom respects all precedences and meets all deadlines.

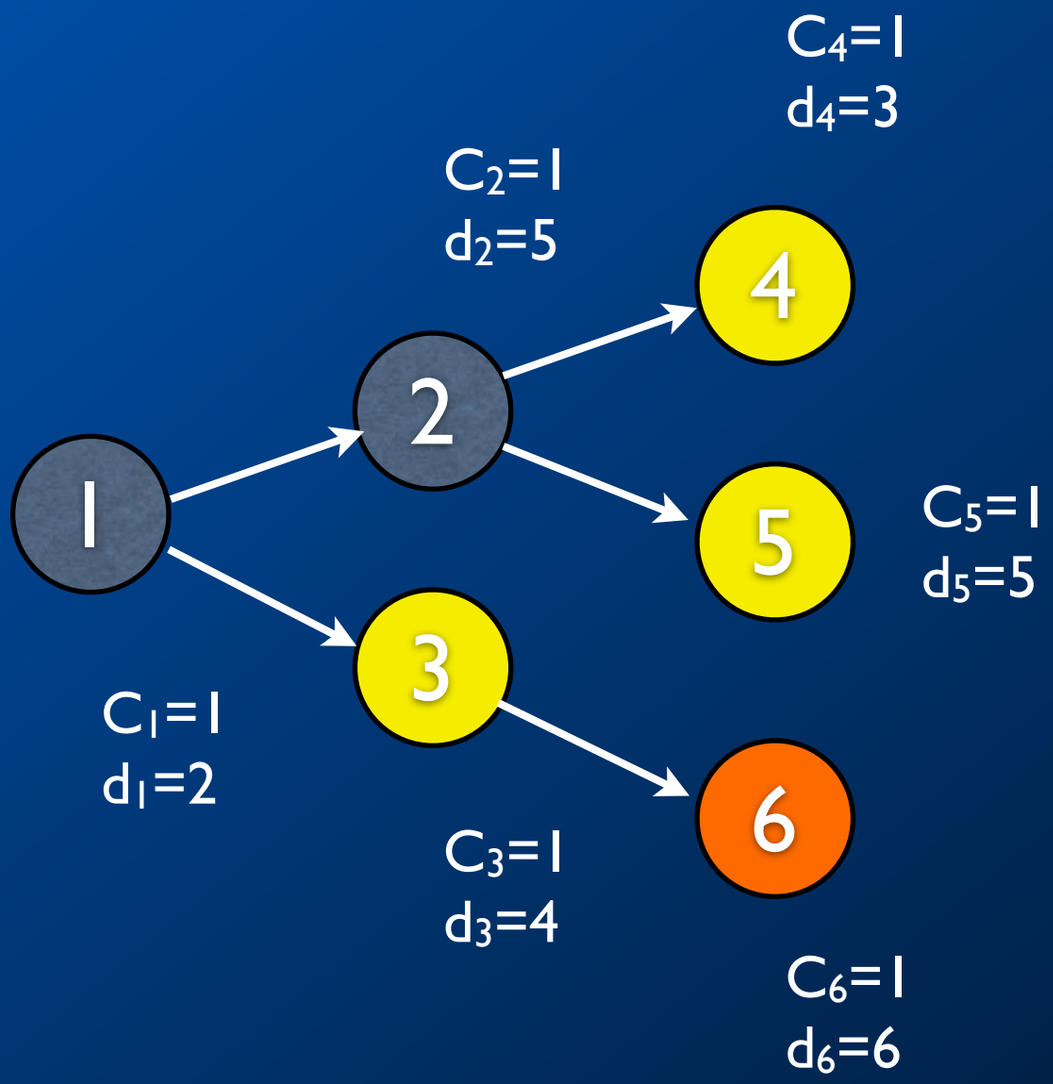
Latest Deadline First (LDF)(Lawler, 1973)

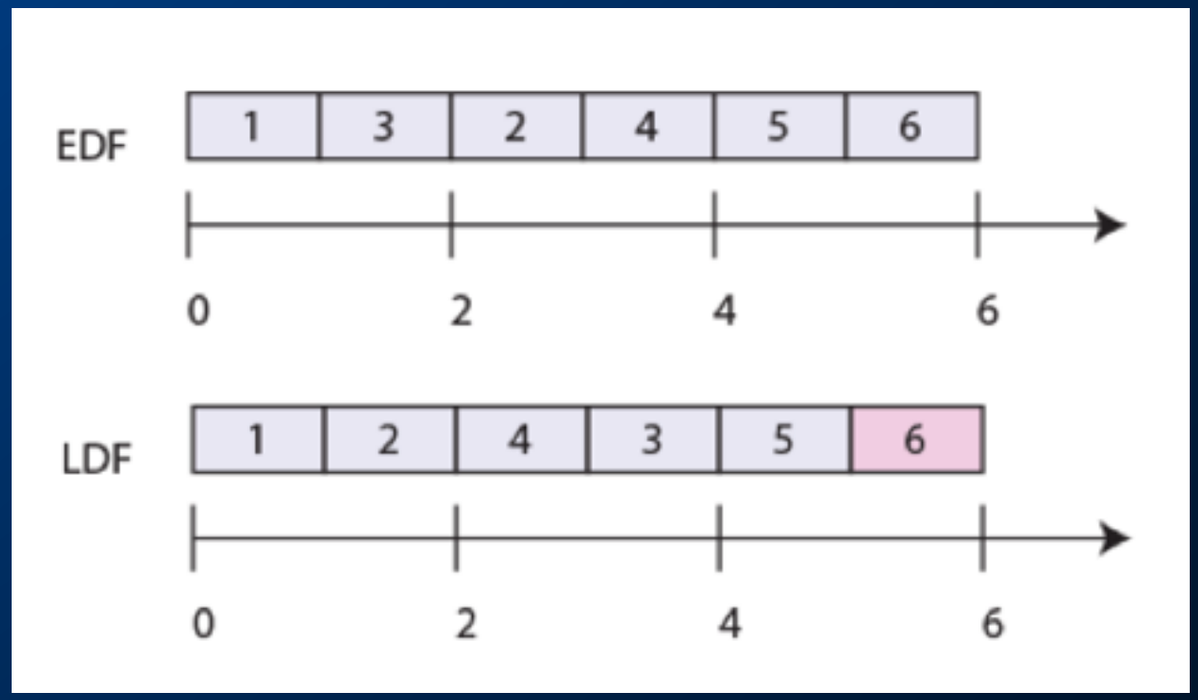




The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

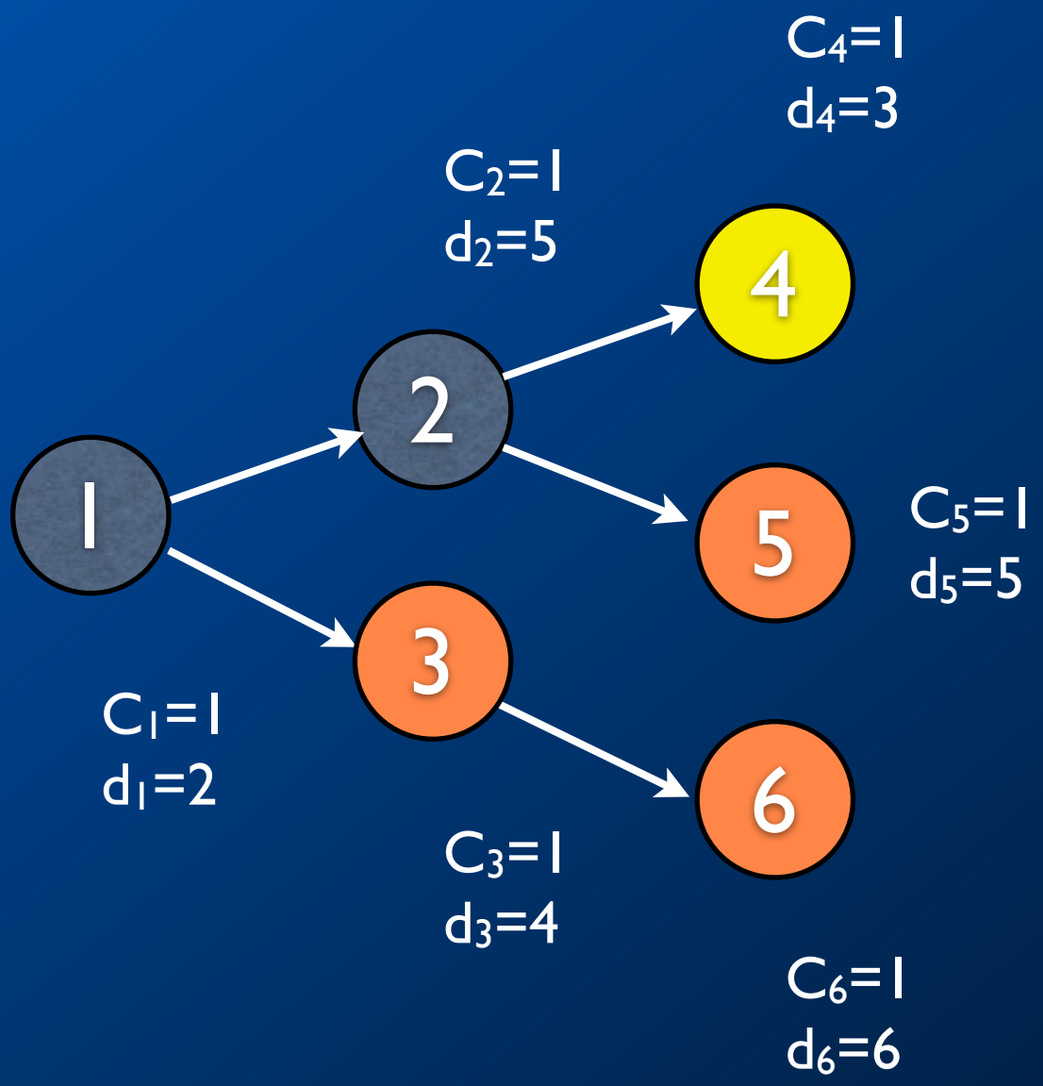
Latest Deadline First (LDF)(Lawler, 1973)

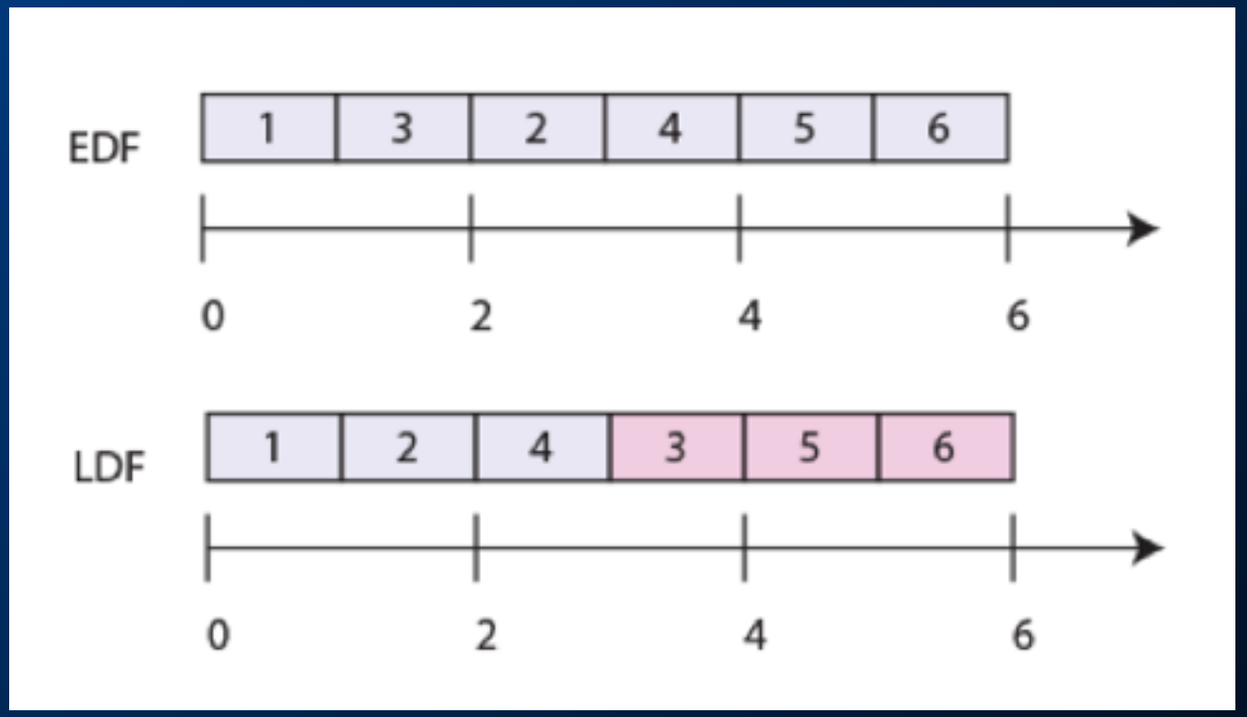




The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

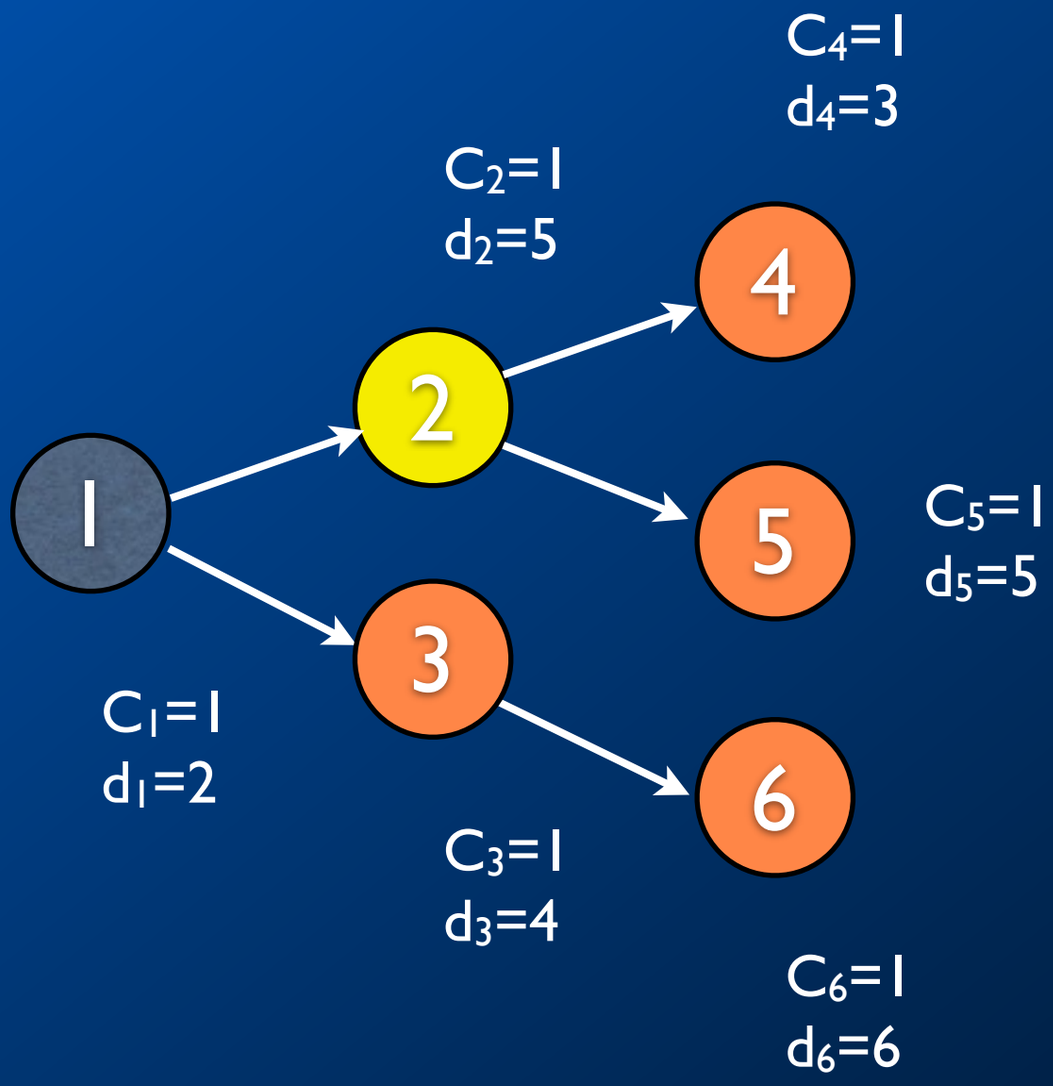
Latest Deadline First (LDF)(Lawler, 1973)

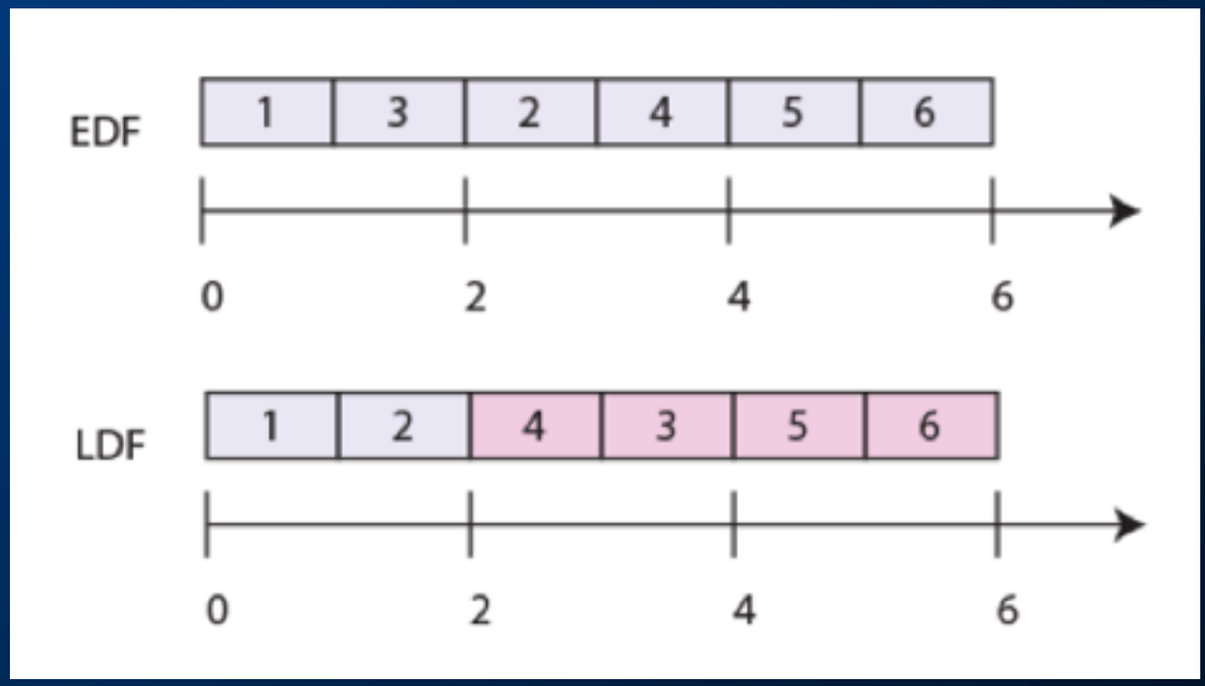




The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

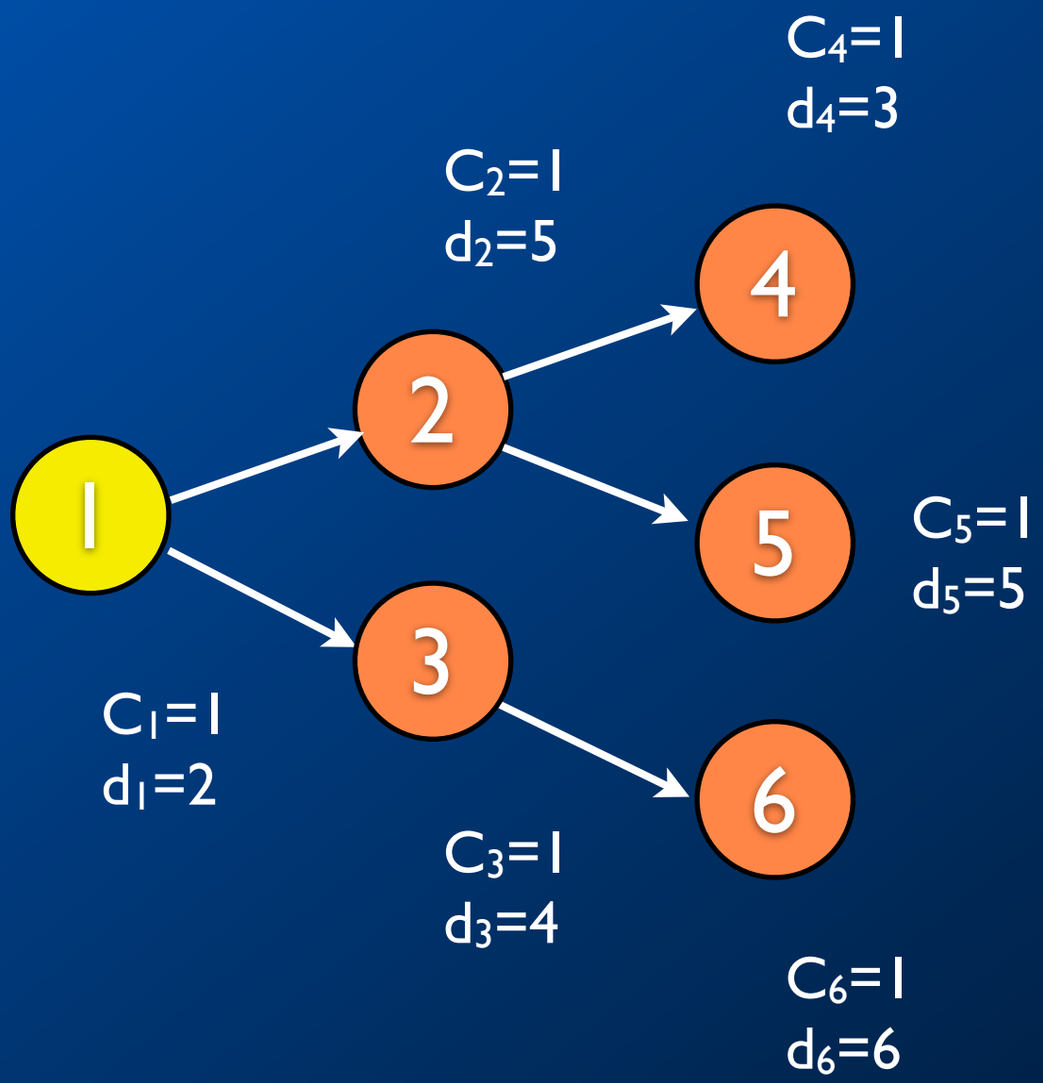
Latest Deadline First (LDF)(Lawler, 1973)

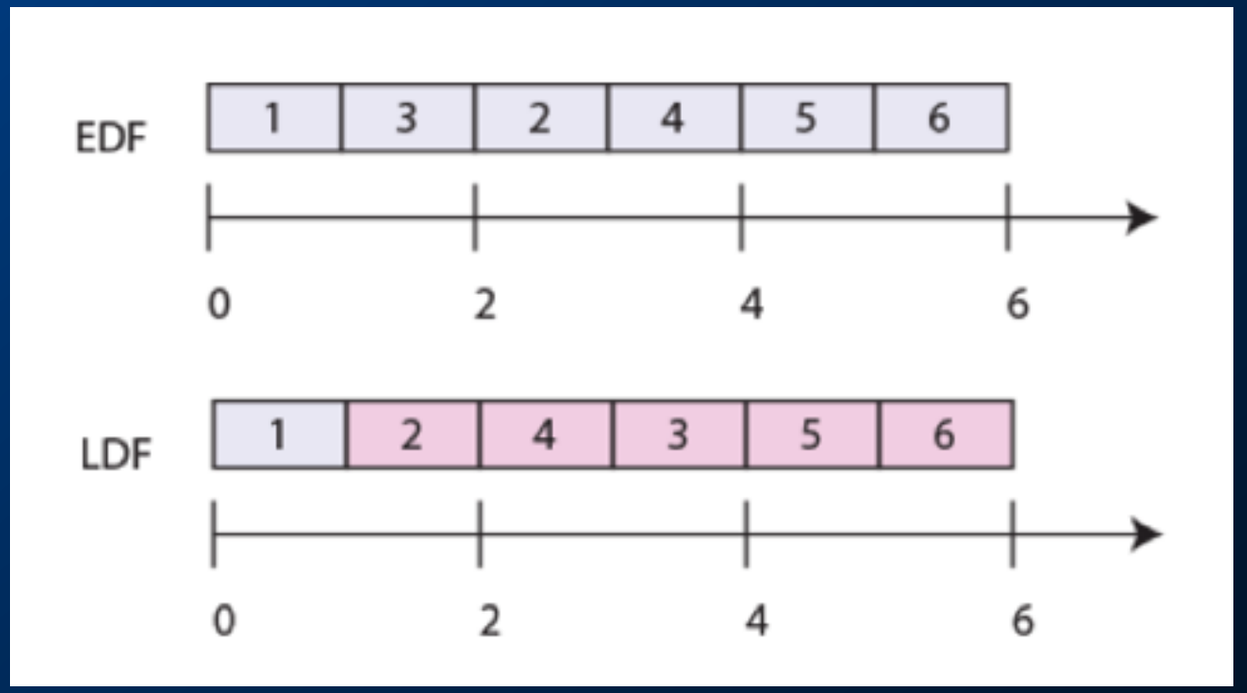




The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

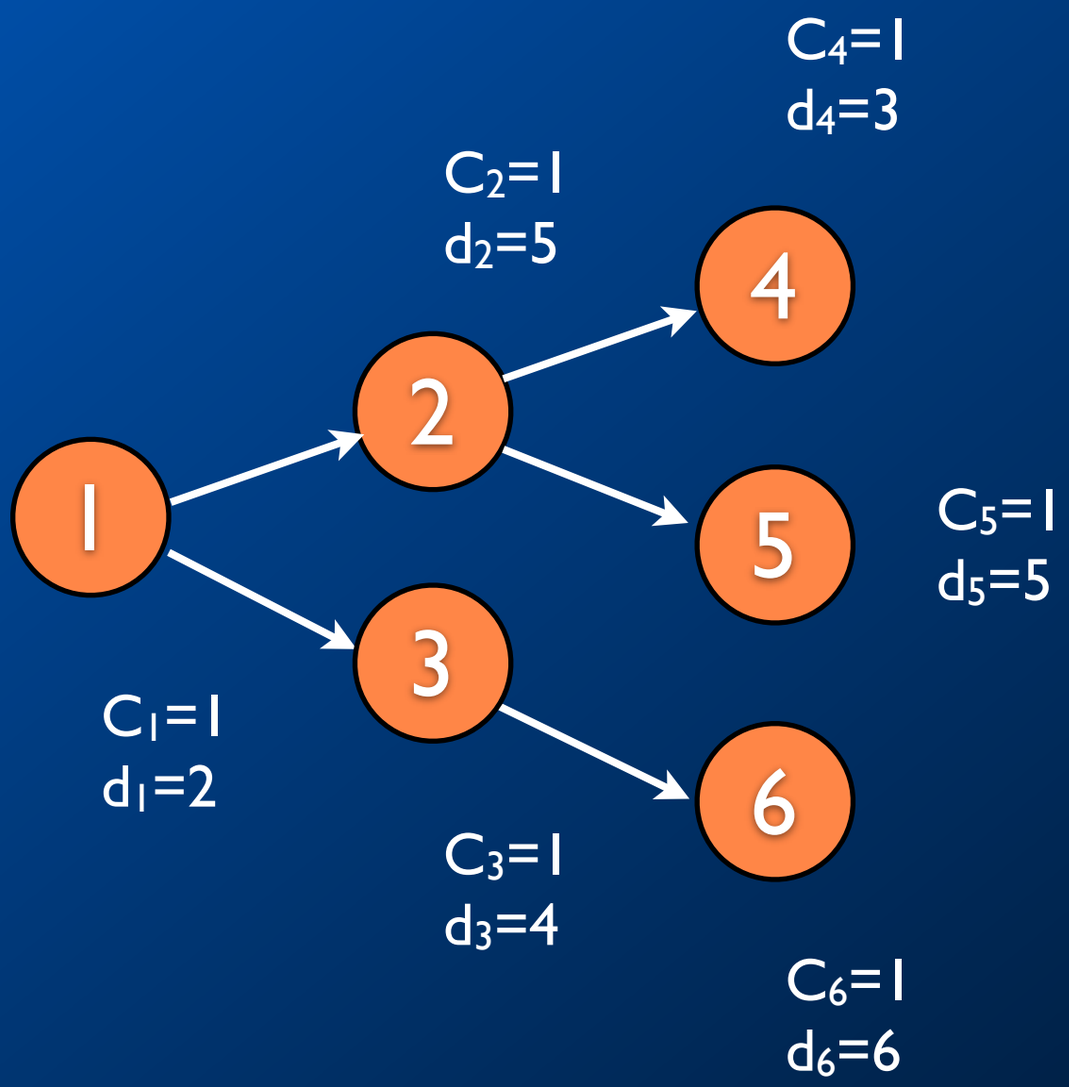
Latest Deadline First (LDF)(Lawler, 1973)

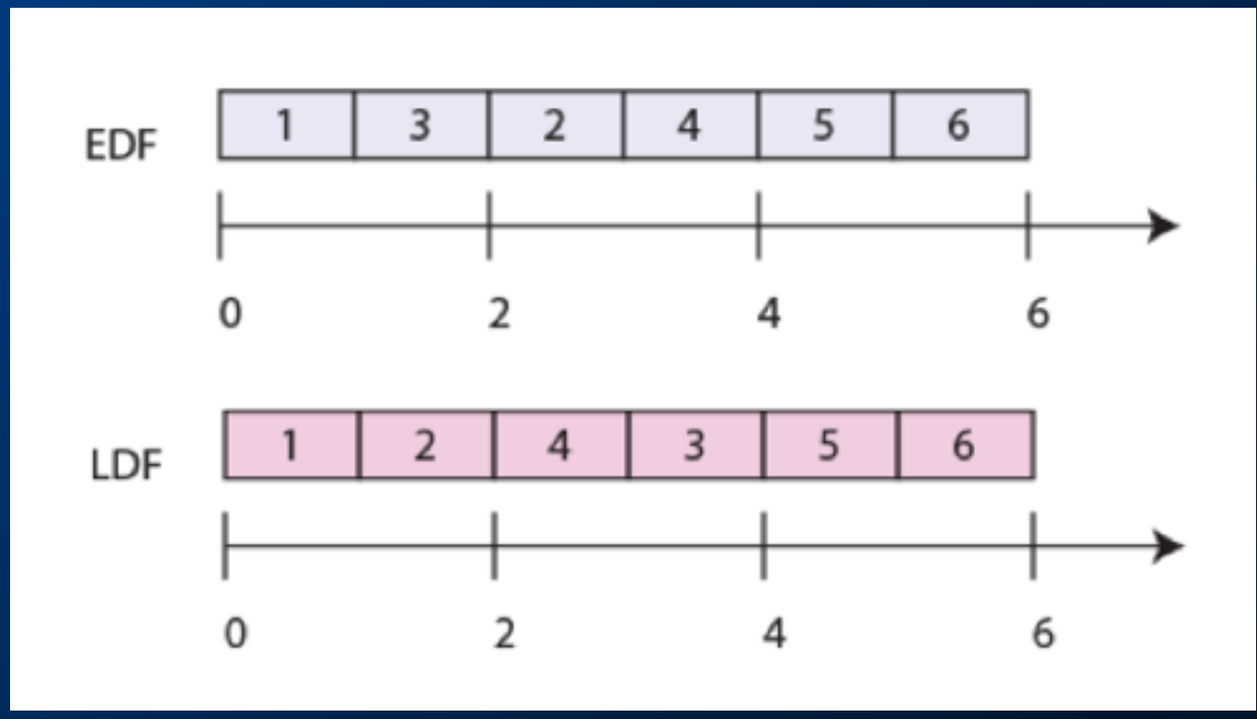




The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)(Lawler, 1973)





The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF) (Lawler, 1973)

- LDF is optimal in the sense that it minimizes the maximum lateness.
- It does not require preemption. (We'll see that EDF does.)
- However, it requires that all tasks be available and their precedences known before any task is executed.

EDF*

- The problem of scheduling a set of n tasks with precedence constraints (concurrent activation) can be solved in polynomial time complexity if tasks are preemptable.
- The EDF* algorithm determines a feasible schedule in the case of tasks with precedence constraints if there exists one.
- By the modification it is guaranteed that if there exists a valid schedule at all then
  - a task starts execution not earlier than its release time and not earlier than the finishing times of its predecessors (a task cannot preempt any predecessor)
  - all tasks finish their execution within their deadlines

EDF*

- Modification of deadlines:
  - Task must finish the execution time within its deadline
  - Task must not finish the execution later than the maximum start time of its successor
- Solution: task j depends on task i

$$d_i' = \min(d_i, d_j' - C_j)$$

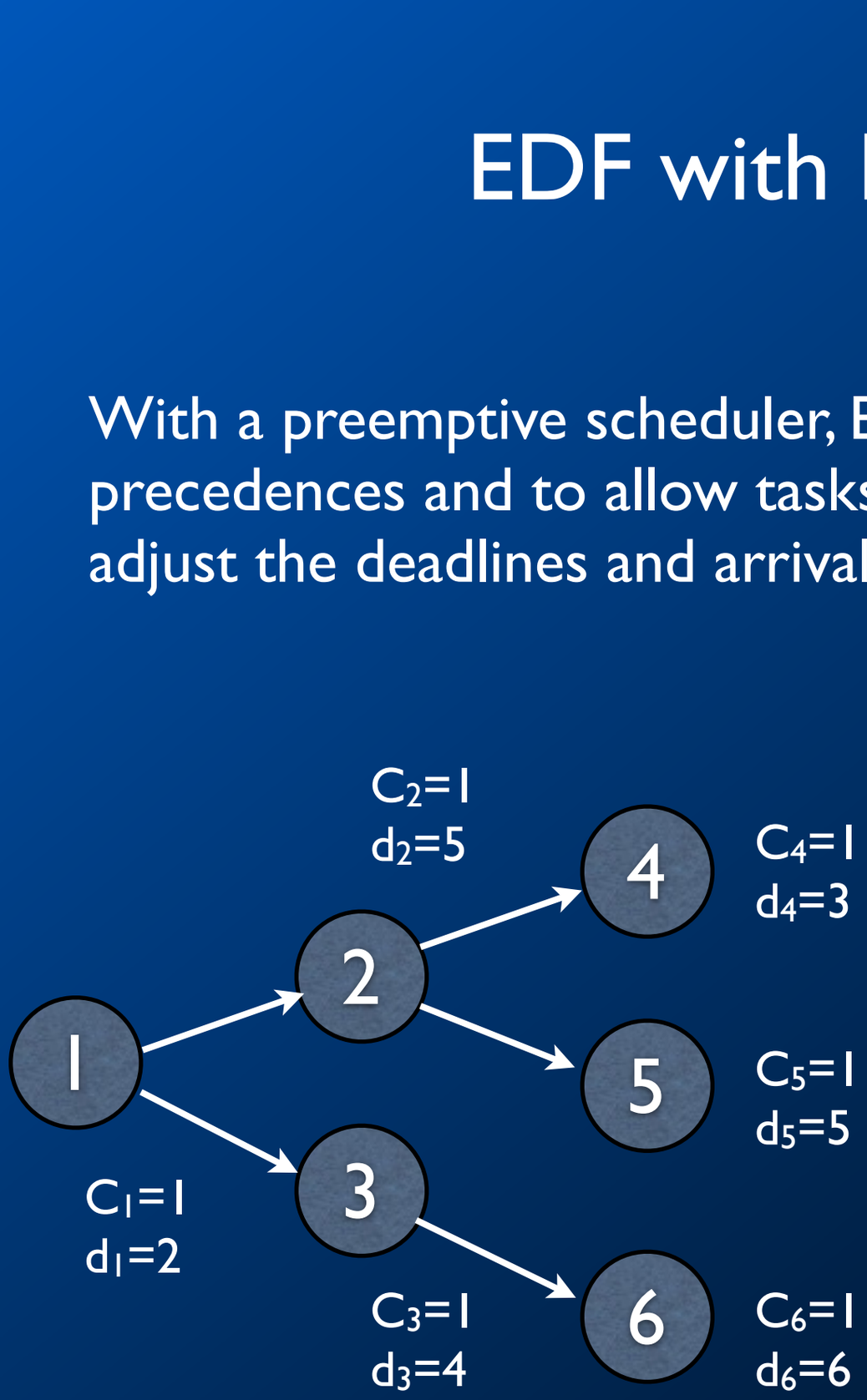
EDF*

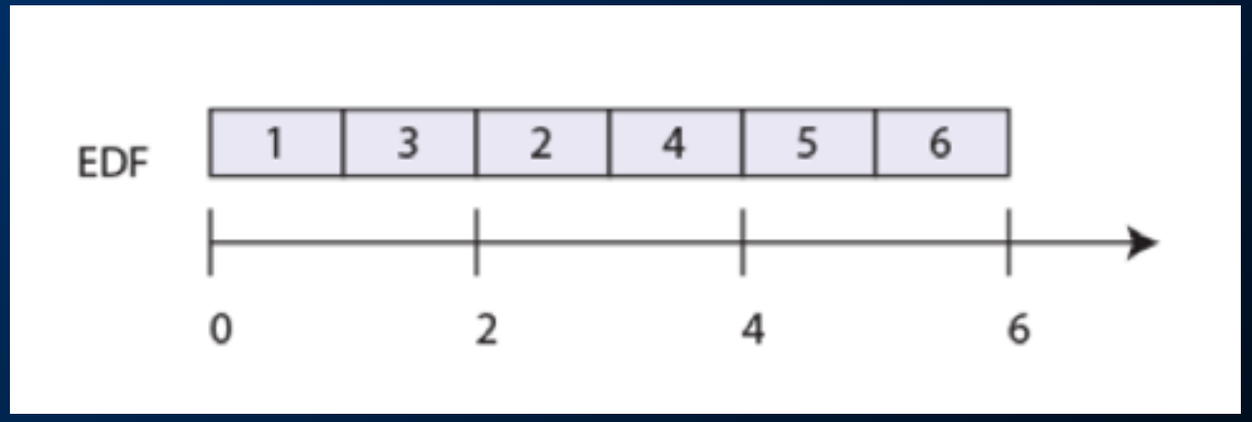
- Modification of release times:
  - Task must start the execution not earlier than its release time.
  - Task must not start the execution earlier than the minimum finishing time of its predecessor.
- Solution: task j depends on task i

$$r_j' = \max(r_j, r_i + C_i)$$

EDF with Precedences

With a preemptive scheduler, EDF can be modified to account for precedences and to allow tasks to arrive at arbitrary times. Simply adjust the deadlines and arrival times according to the precedences.





Recall that for the tasks at the left, EDF yields the schedule above, where task 4 misses its deadline.

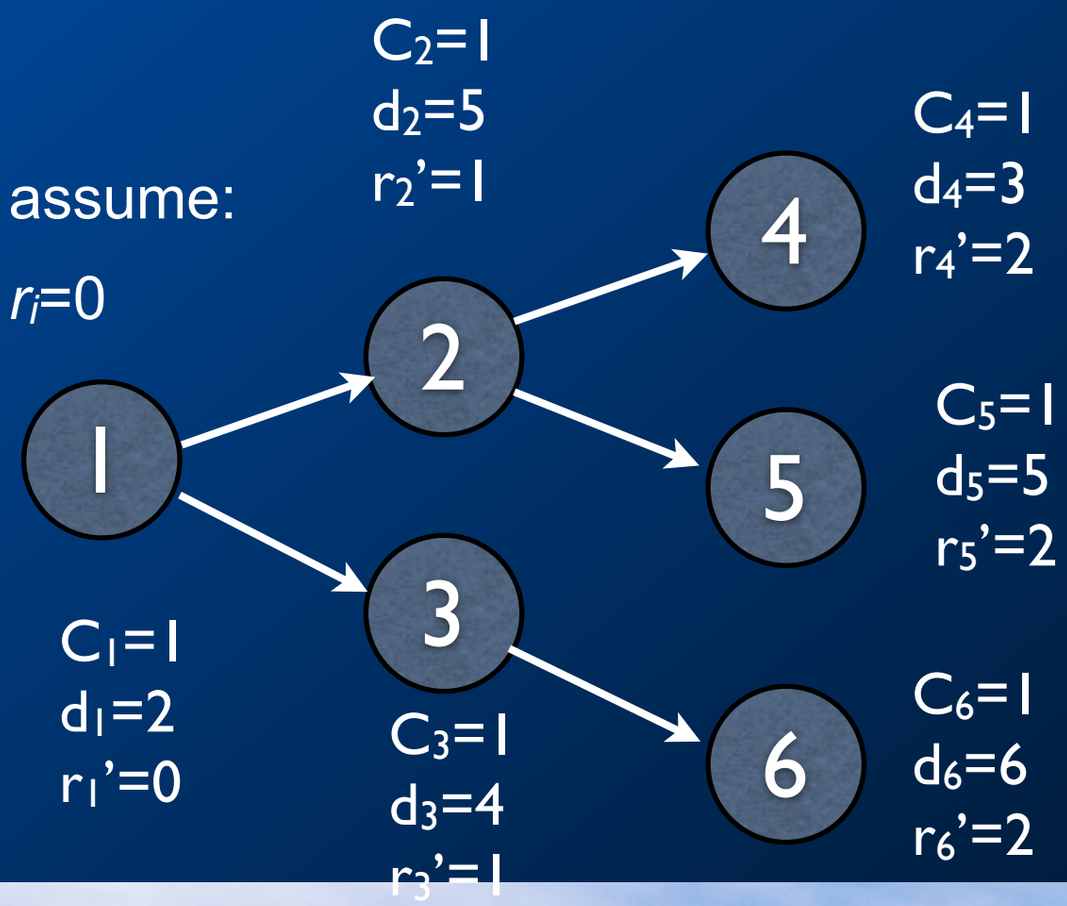
EDF with Precedences
Modifying release times

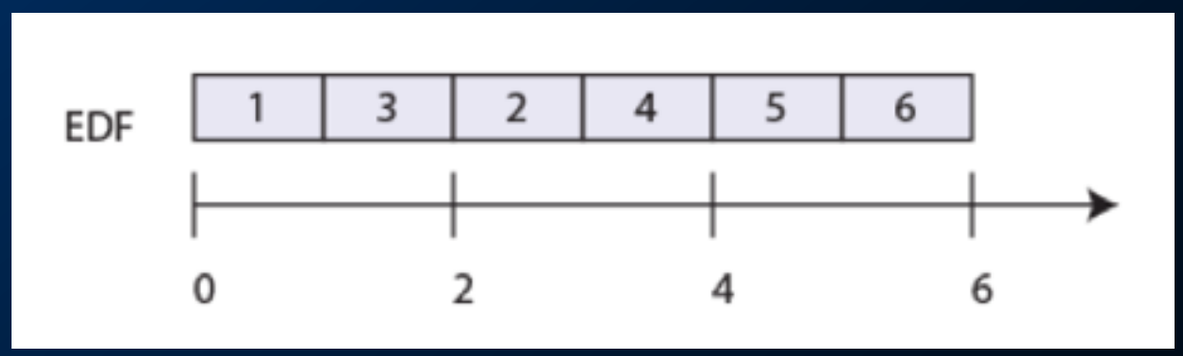
Given *n* tasks with precedences and release times r_i, if task *i* immediately precedes task *j*, then modify the release times as follows:

$$r'_j = \max(r_j, r_i + C_i)$$

assume:

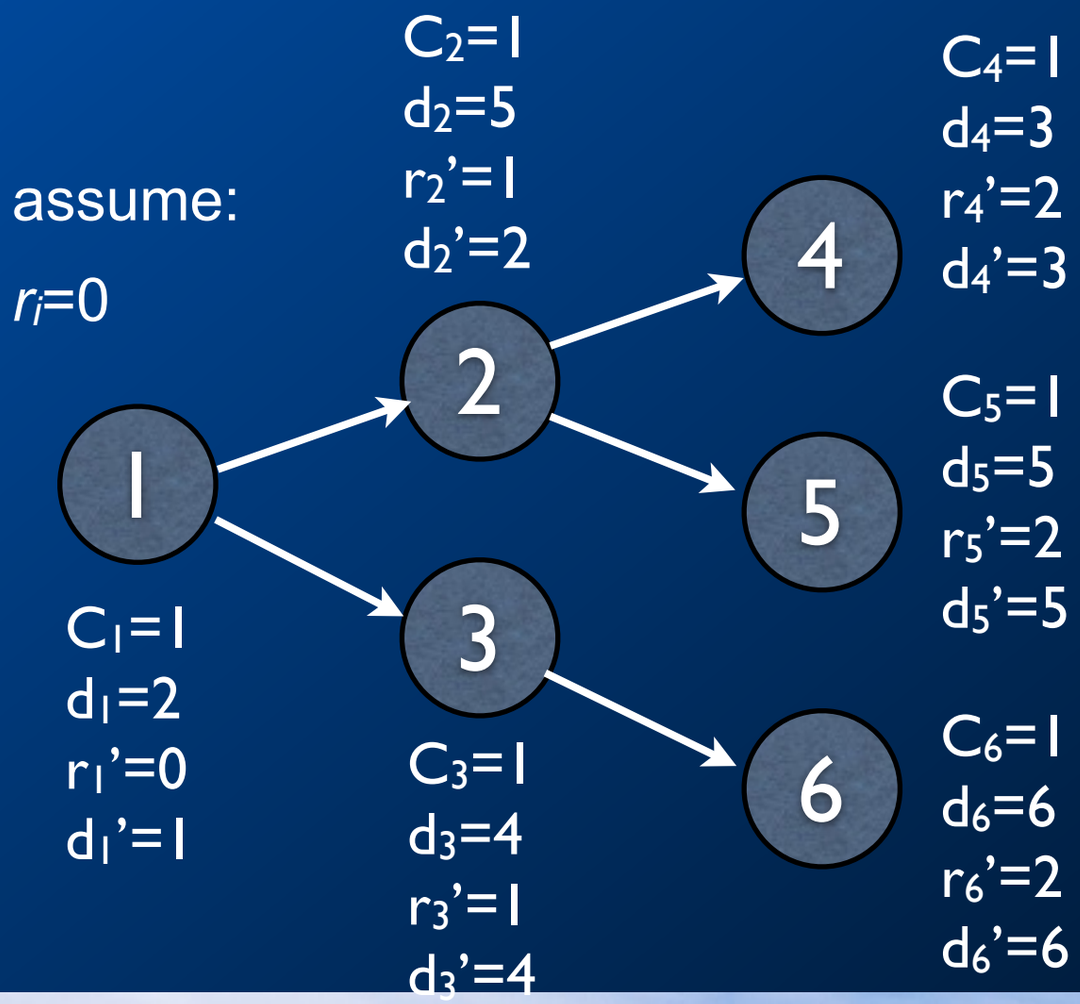
r_i=0



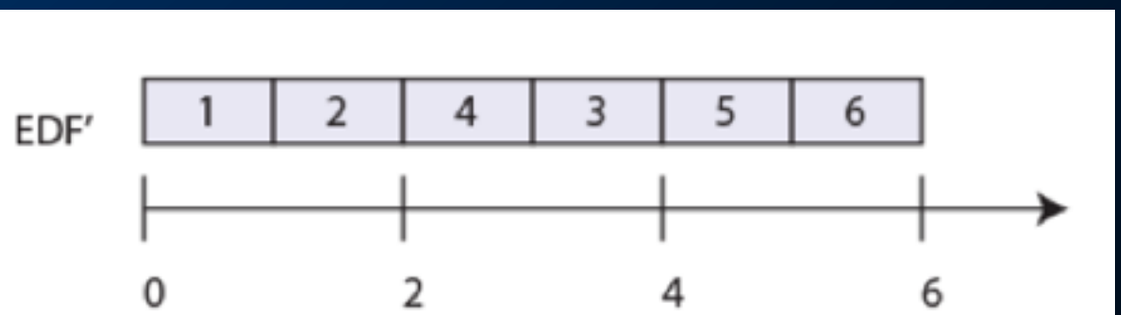


EDF with Precedences
Modifying deadlines

Given *n* tasks with precedences and deadlines d_i, if task i immediately precedes task *j*, then modify the deadlines as follows:



$$d'_i = \min(d_i, d'_j - C_j)$$



Using the revised release times and deadlines, the above EDF schedule is optimal and meets all deadlines.

Optimality

- EDF with precedences is optimal in the sense of minimizing the maximum lateness.

scheduling anomalies

- Mutual exclusion
  - Priority inversion
  - Priority inheritance
  - Priority ceiling
- Multiprocessor scheduling
  - Richard's anomalies

Accounting for Mutual Exclusion

- When threads access shared resources, they need to use mutexes to ensure data integrity.
- Mutexes can also complicate scheduling.

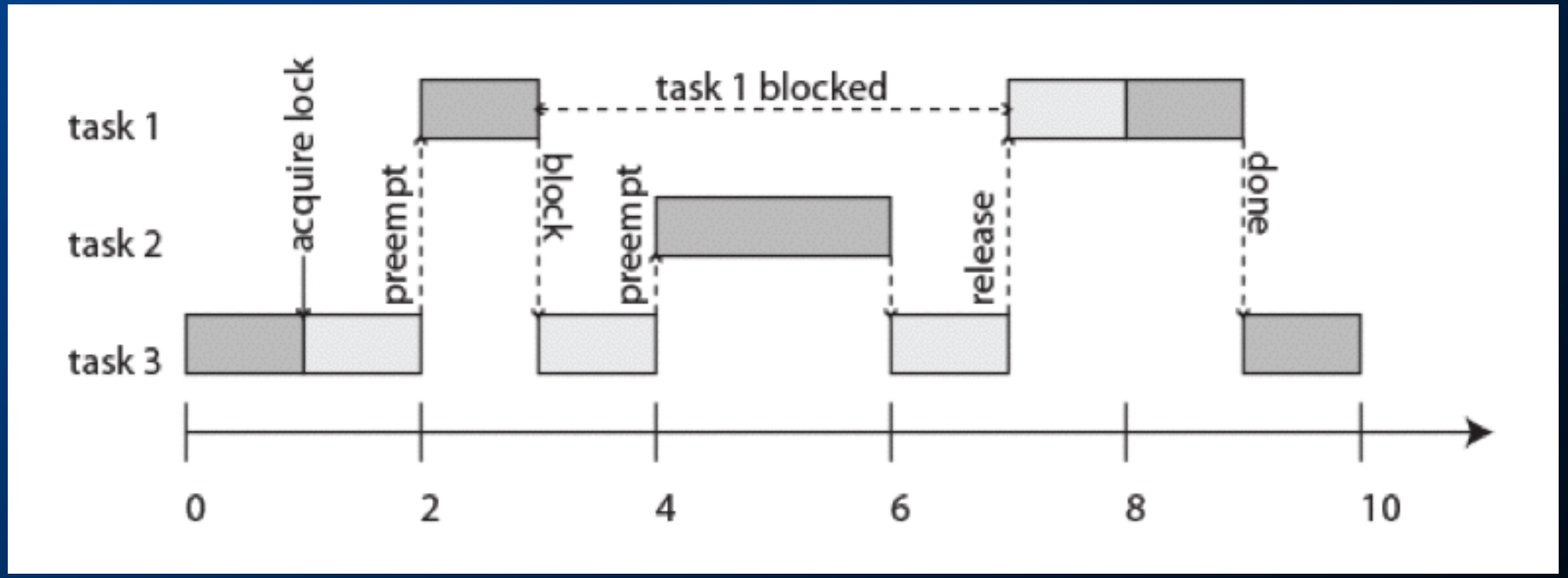
Recall mutual exclusion mechanism in pthreads

```
#include <pthread.h>
...
pthread_mutex_t lock;
void* addListener(notify listener) {
        pthread_mutex_lock(&lock);
        ...
        pthread_mutex_unlock(&lock);
}
void* update(int newValue) {
        pthread_mutex_lock(&lock);
        value = newValue;
        elementType* element = head;
        while (element != 0) {
            (*(element->listener))(newValue);
            element = element->next;
        }
        pthread_mutex_unlock(&lock);
   }
int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

Whenever a data structure is shared across threads, access to the data structure must usually be atomic. This is enforced using mutexes, or mutual exclusion locks. The code executed while holding a lock is called a *critical section.*

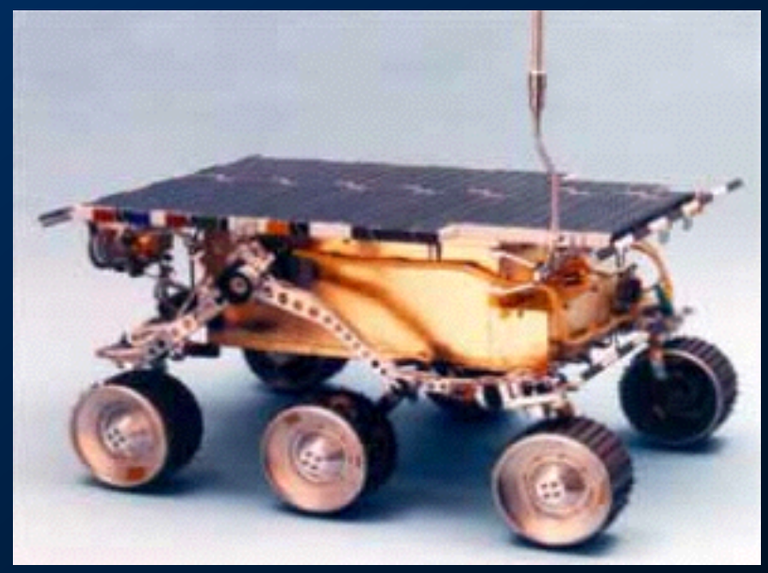
Priority Inversion: A Hazard with Mutexes

- Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.



The MARS Pathfinder problem (1)

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data.



The MARS Pathfinder problem (2)

- “VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”
- “Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).”

The MARS Pathfinder problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread, ... When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority."

High priority: retrieval of data from shared memory
Medium priority: communications task
Low priority: thread collecting meteorological data

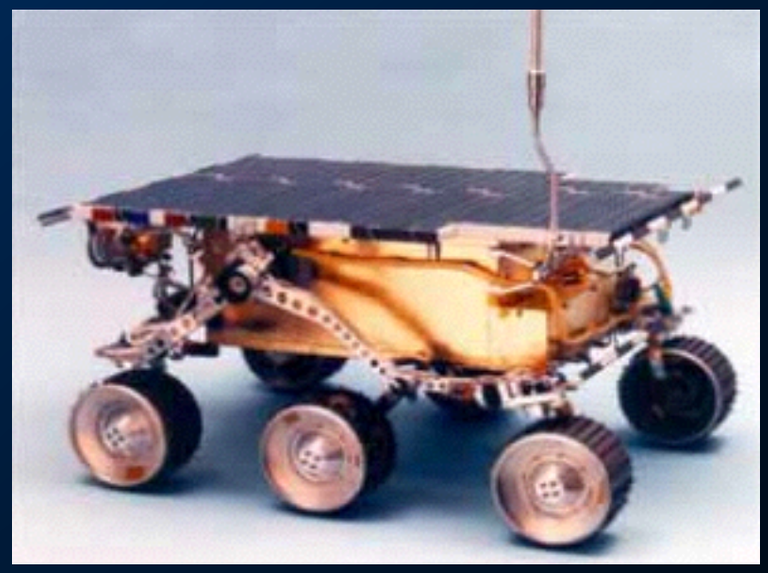
The MARS Pathfinder problem (4)

"Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion."

Priority inversion on Mars

- Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to "on". When the software was shipped, it was set to "off".

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to "on", while the Pathfinder was already on the Mars [Jones, 1997].



Priority Inheritance Protocol (PIP) (Sha, Rajkumar, Lehoczky, 1990)

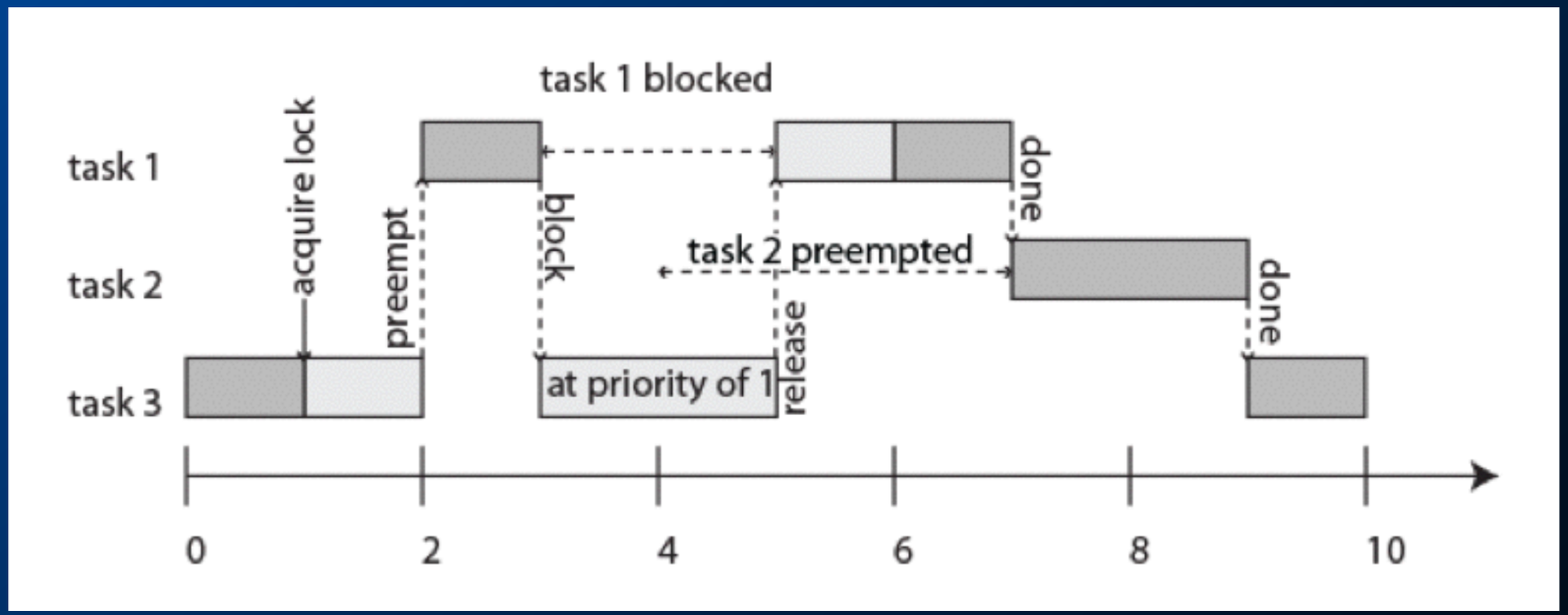
- Assumptions:
  - *n* tasks which cooperate through *m* shared resources; fixed priorities, all critical sections on a resource begin with a wait(*Si*) and end with a signal(*Si*) operation.
- Basic idea:
  - When a task *Ji* blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.
- Terms:
  - We distinguish a fixed nominal priority *Pi* and an active priority *pi* larger or equal to *Pi*. Jobs J_1, ...J_n are ordered with respect to nominal priority where J_1 has highest priority. Jobs do not suspend themselves.

Priority Inheritance Protocol (PIP)

- Algorithm:
  - Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a FCFS discipline.
  - When a job J_i tries to enter a critical section and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
  - When a job J_i is blocked, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it inherits the priority of the highest priority of the jobs blocked by it).
  - When J_k exits a critical section, it unlocks the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
  - Priority inheritance is transitive, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

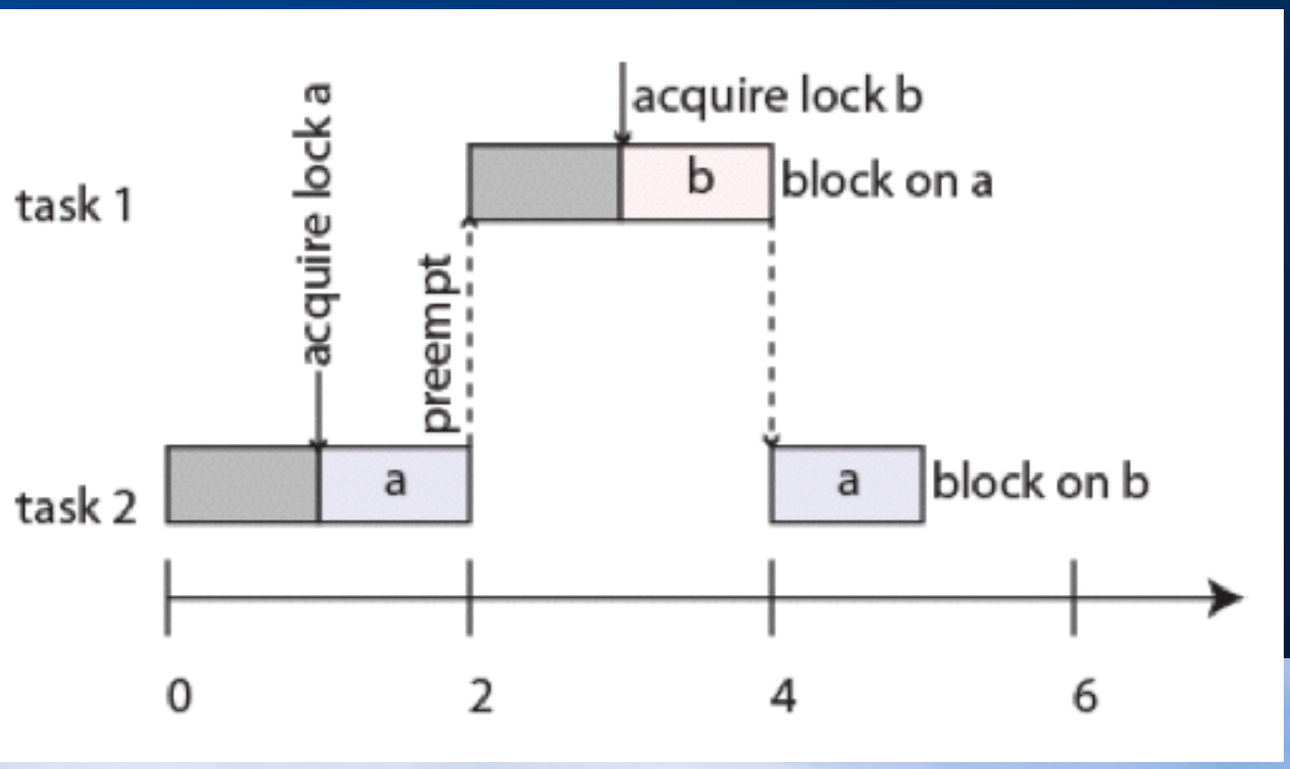
Priority Inheritance Protocol (PIP) - example

- Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 3 inherits the priority of task 1, preventing preemption by task 2.



Deadlock

The lower priority task starts first and acquires lock a, then gets preempted by the higher priority task, which acquires lock b and then blocks trying to acquire lock a. The lower priority task then blocks trying to acquire lock b, and no further progress is possible.



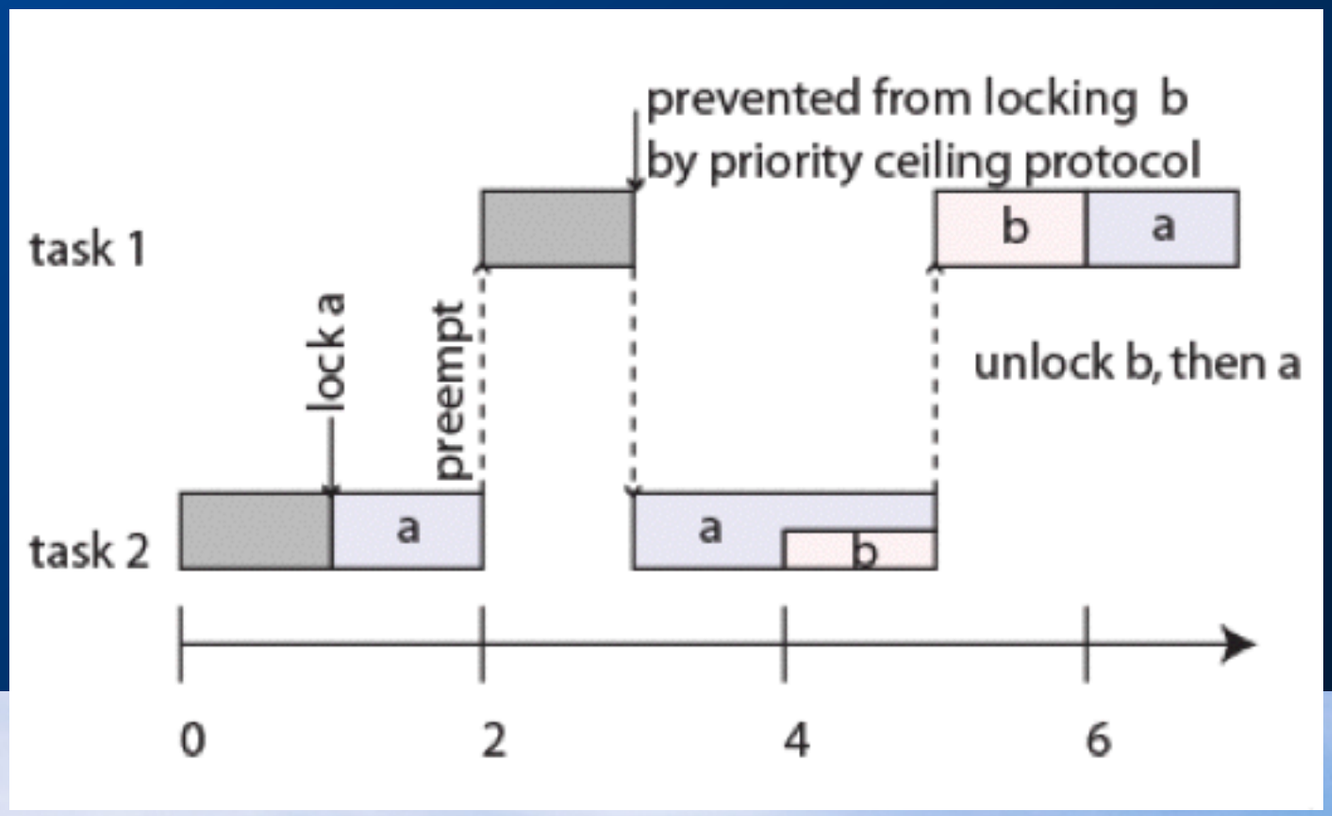

```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;
void* thread_1_function(void* arg) {
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
}
void* thread_2_function(void* arg) {
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
}
```

Priority Ceiling Protocol (PCP) (Sha, Rajkumar, Lehoczky, 1990)

- Every lock or semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it.
  - Can one automatically compute the priority ceiling?
- A task T can acquire a lock only if the task's priority is strictly higher than the priority ceilings of all locks currently held by other tasks
  - Locks that are not held by any task don't affect the task
- This prevents deadlocks
- There are extensions supporting dynamic priorities and dynamic creations of locks (stack resource policy)

Priority Ceiling Protocol

In this version, locks a and b have priority ceilings equal to the priority of task 1. At time 3, task 1 attempts to lock b, but it can't because task 2 currently holds lock a, which has priority ceiling equal to the priority of task 1.

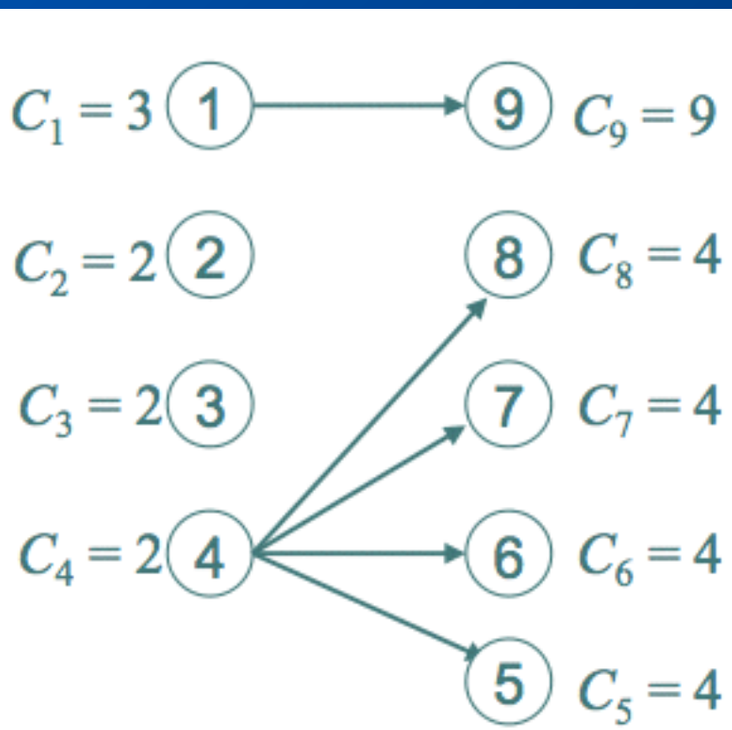



```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;
void* thread_1_function(void* arg) {
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
}
void* thread_2_function(void* arg) {
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
}
```

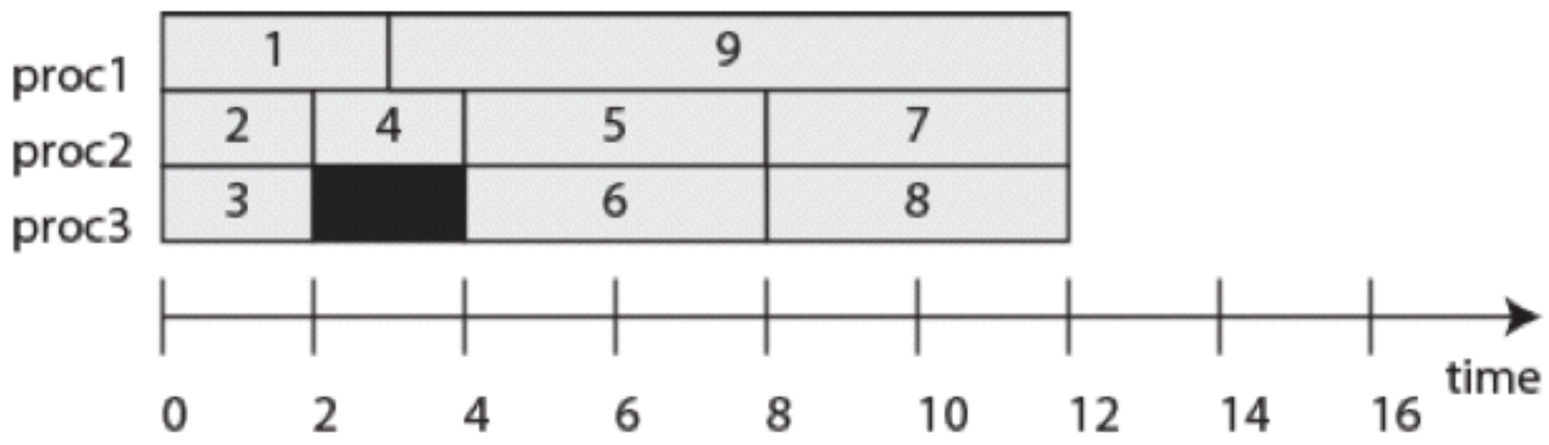
Brittleness

- In general, all thread scheduling algorithms are brittle: Small changes can have big, unexpected consequences.
- I will illustrate this with multiprocessor (or multicore) schedules.
- *Theorem (Richard Graham, 1976): If a task set with fixed priorities, execution times, and precedence constraints is scheduled according to priorities on a fixed number of processors, then increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length.*

Richard's Anomalies

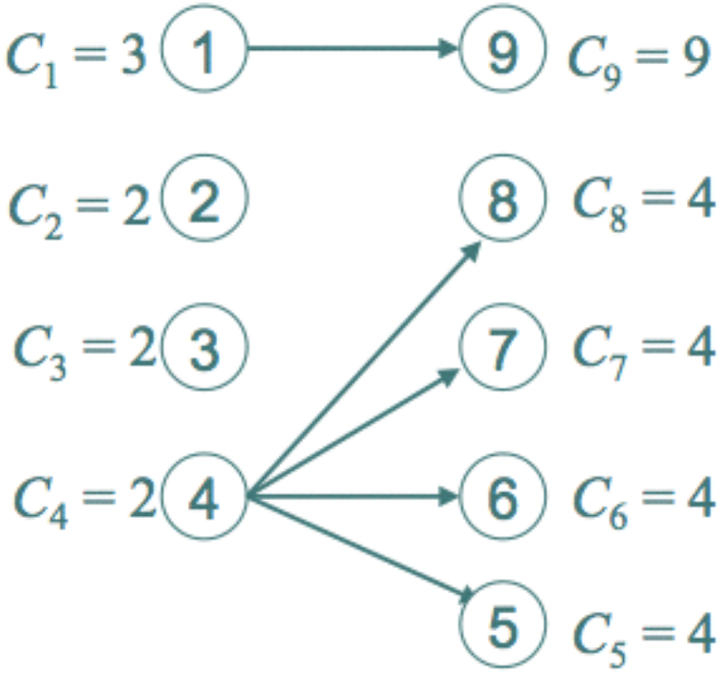


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

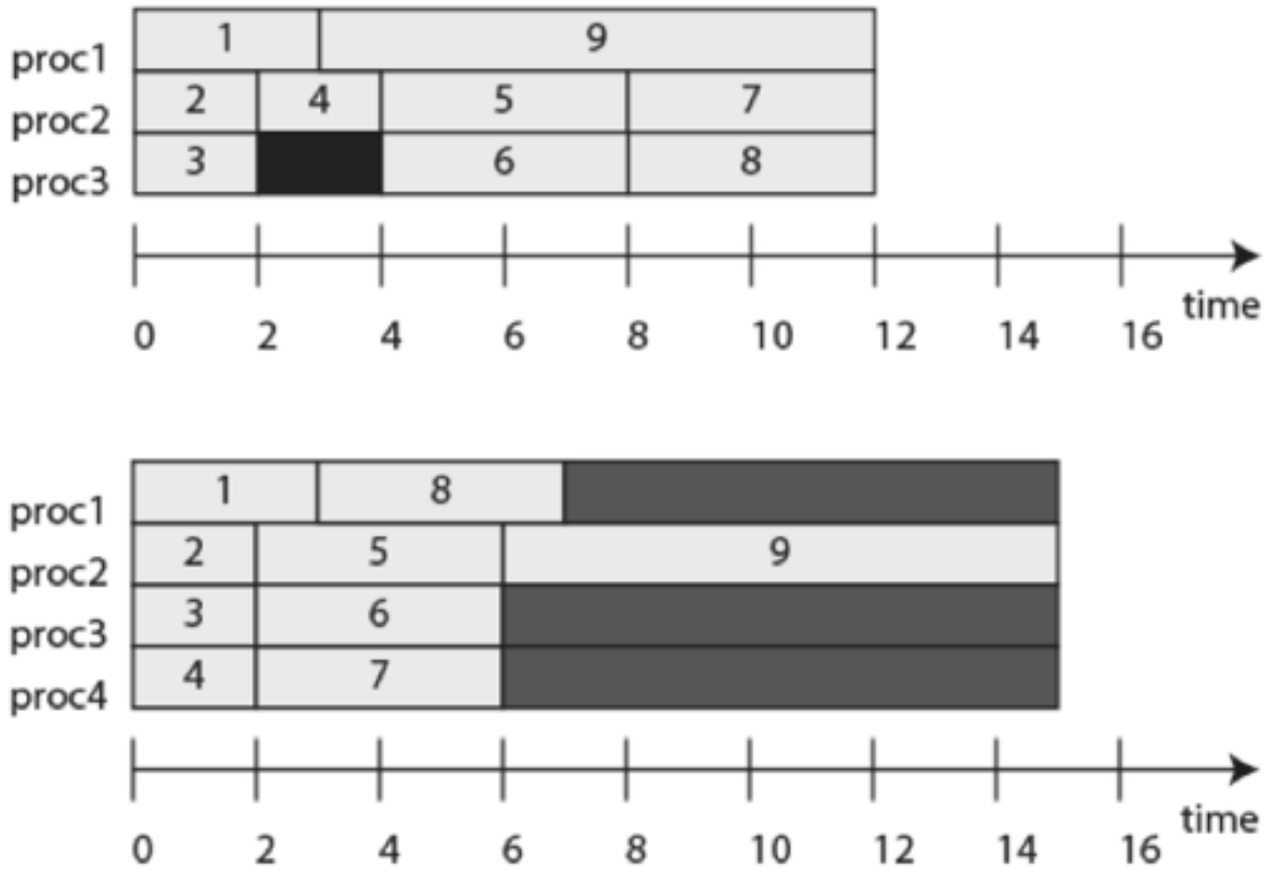


What happens if you increase the number of processors to four?

Richard's Anomalies: Increasing the number of processors

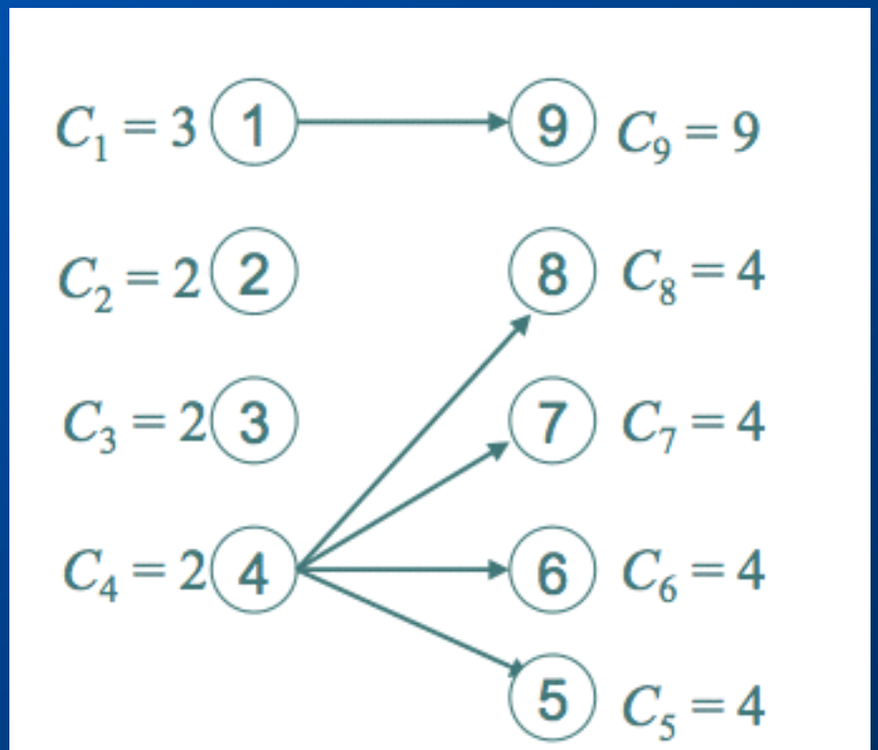


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

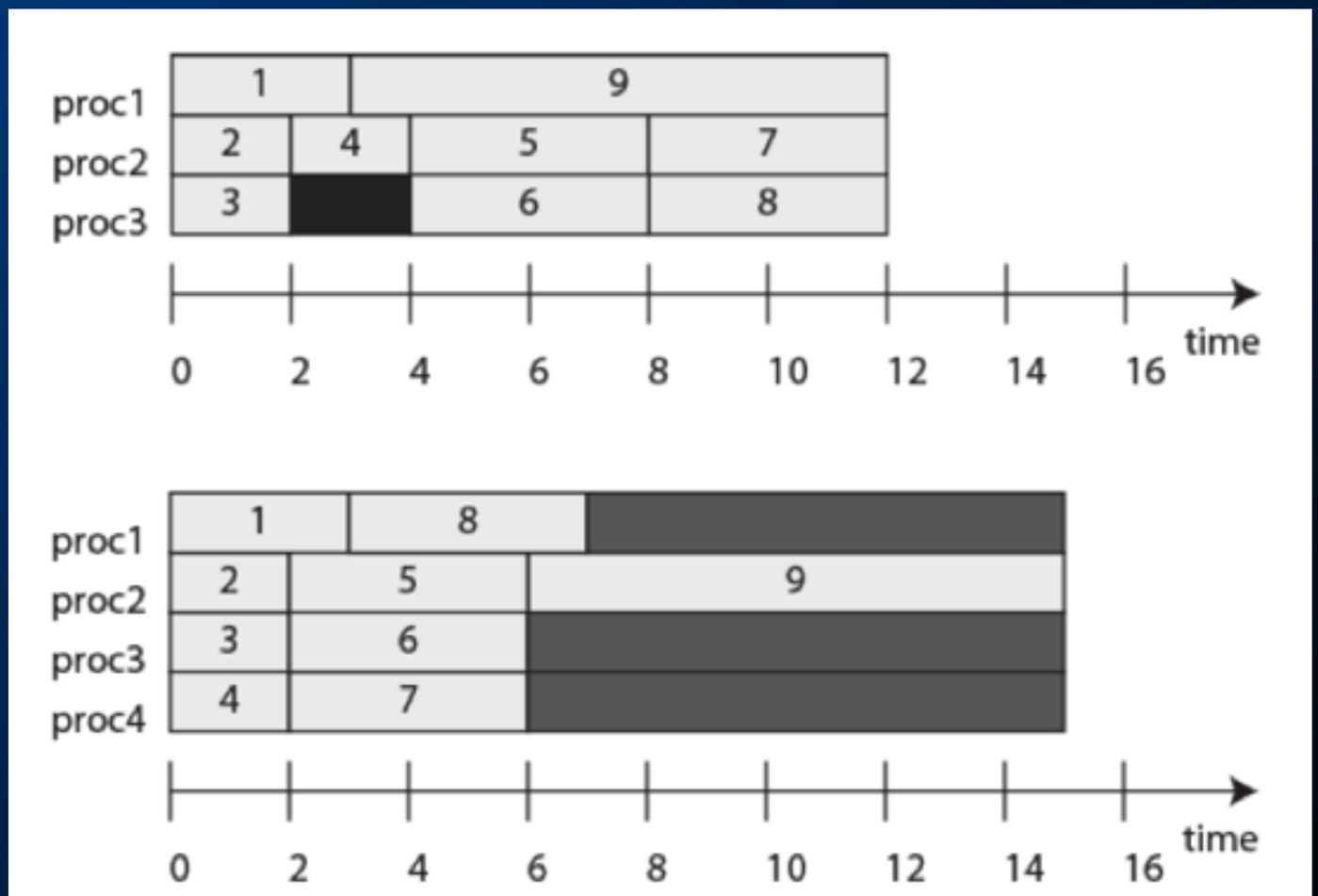


The priority-based schedule with four processors has a longer execution time.

Greedy Scheduling

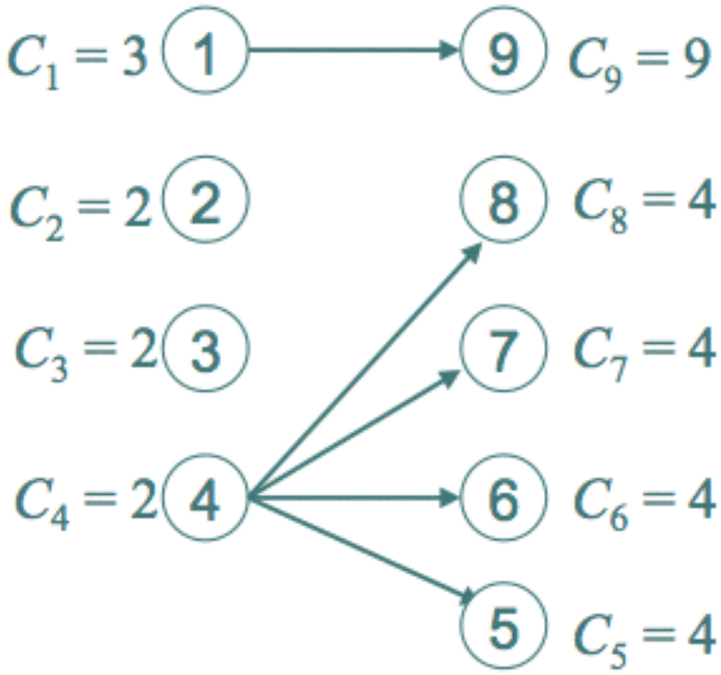


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

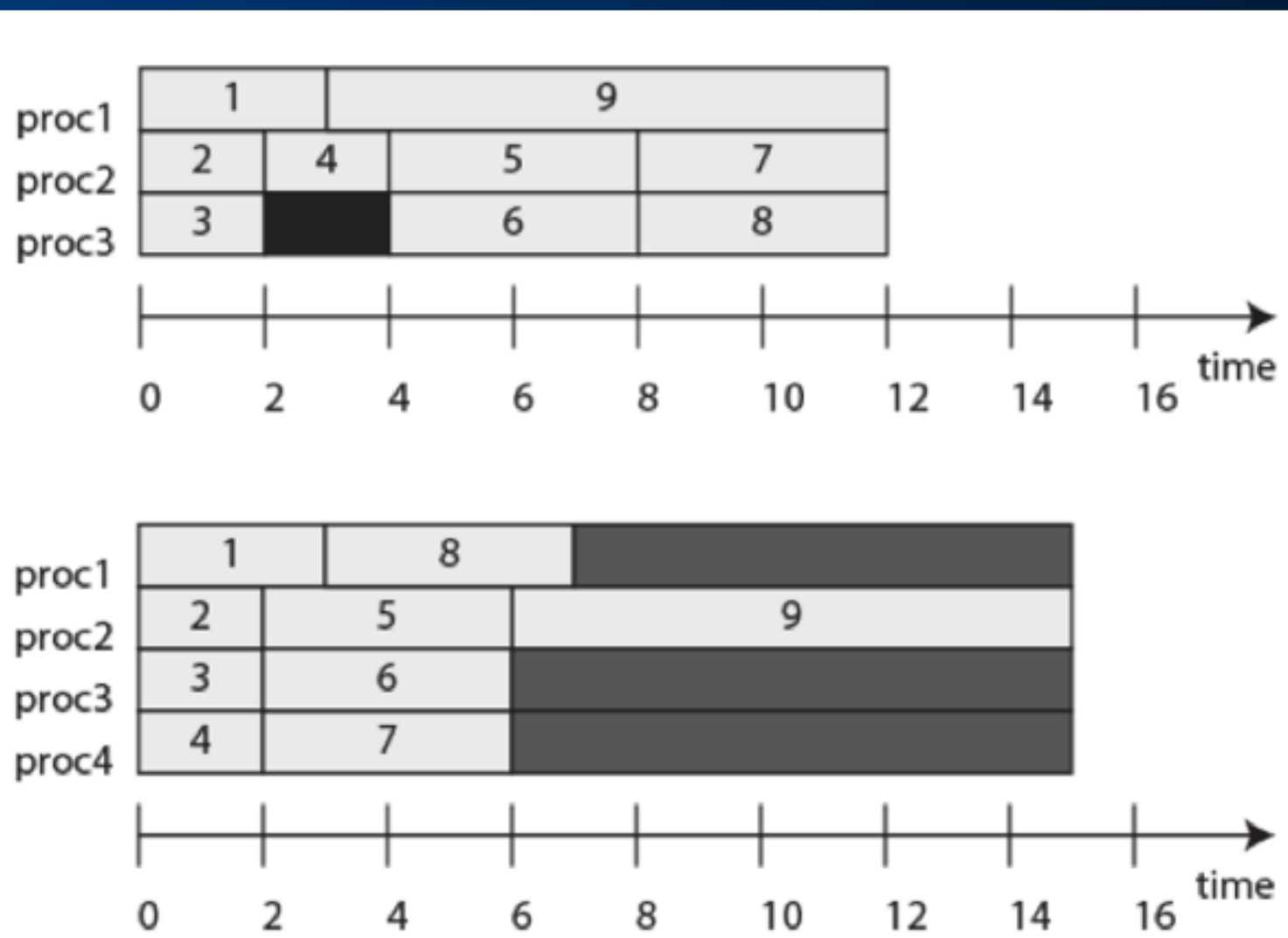


Priority-based scheduling is "greedy." A smarter scheduler for this example could hold off scheduling 5, 6, or 7, leaving a processor idle for one time unit.

Greedy scheduling may be the only practical option.

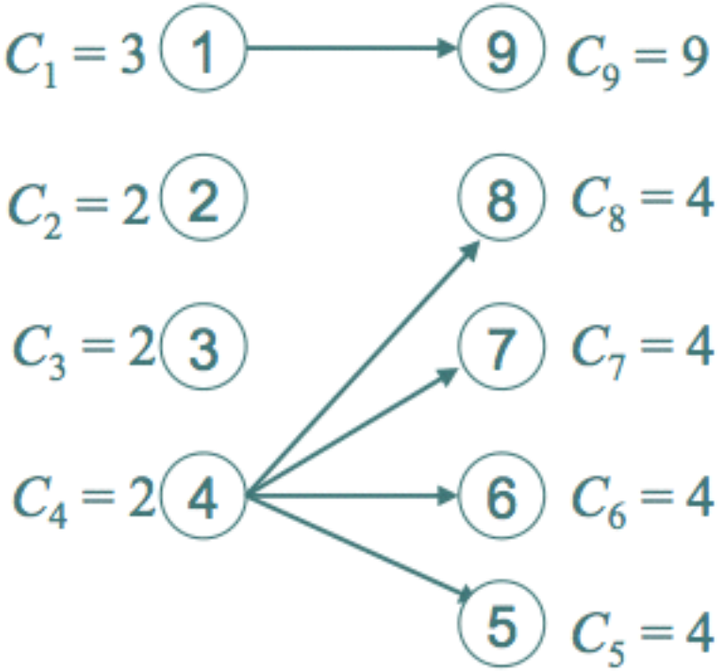


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

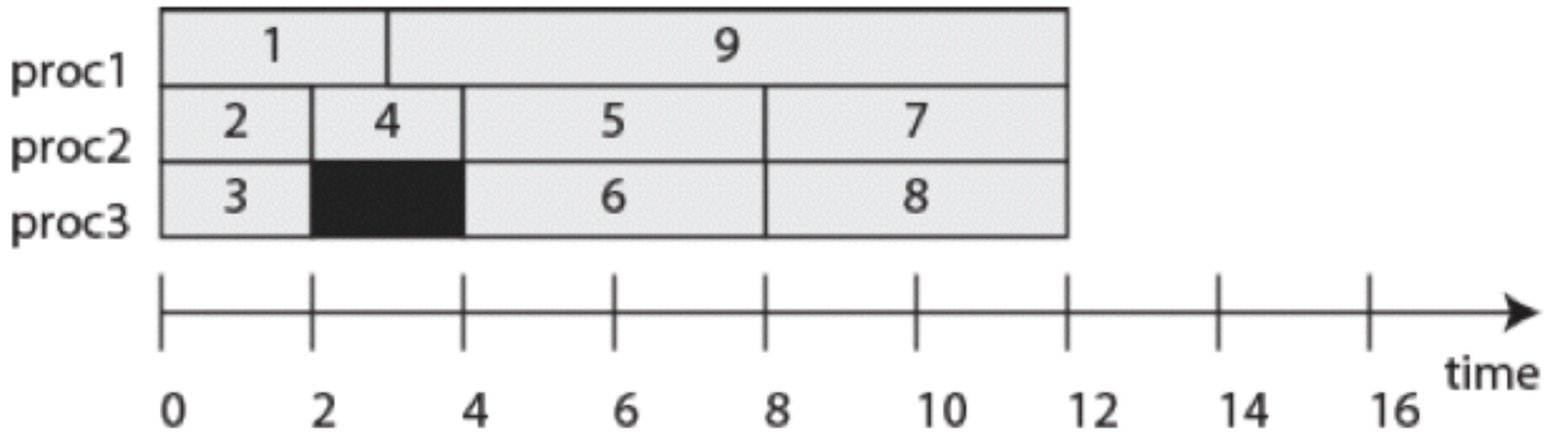


If tasks “arrive” (become known to the scheduler) only after their predecessor completes, then greedy scheduling may be the only practical option.

Richard's Anomalies

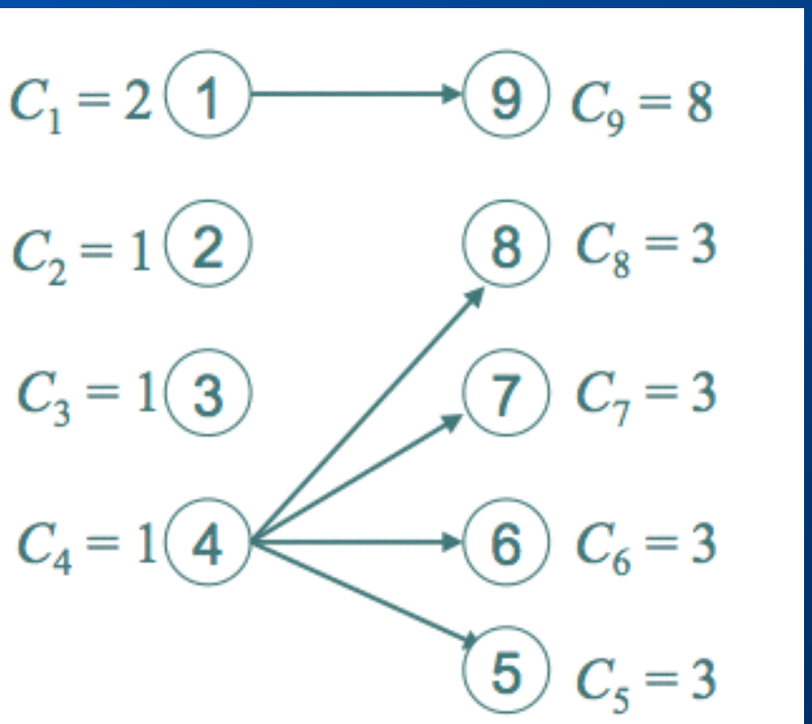


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

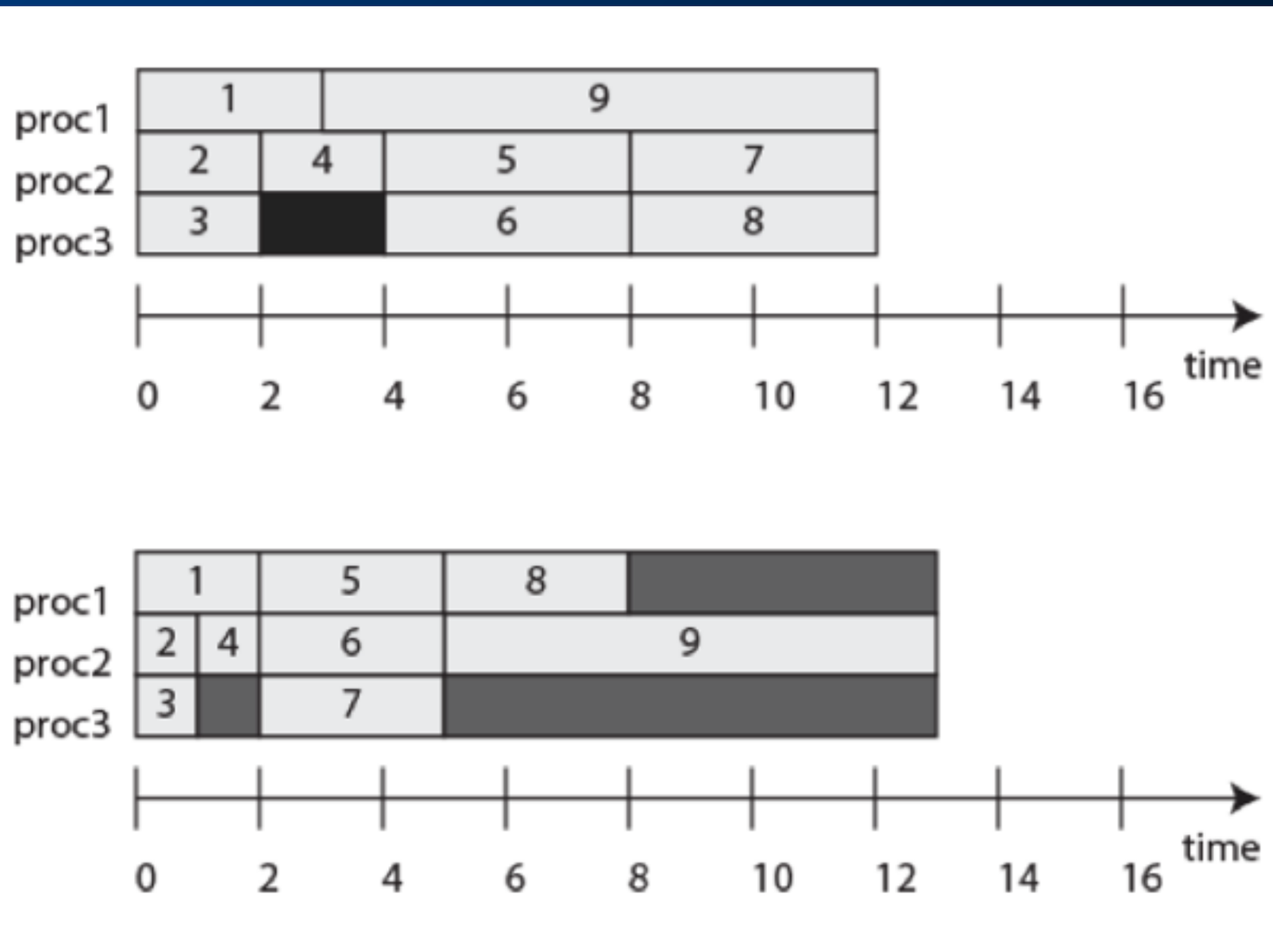


What happens if you reduce all computation times by 1?

Richard's Anomalies: Reducing computation times

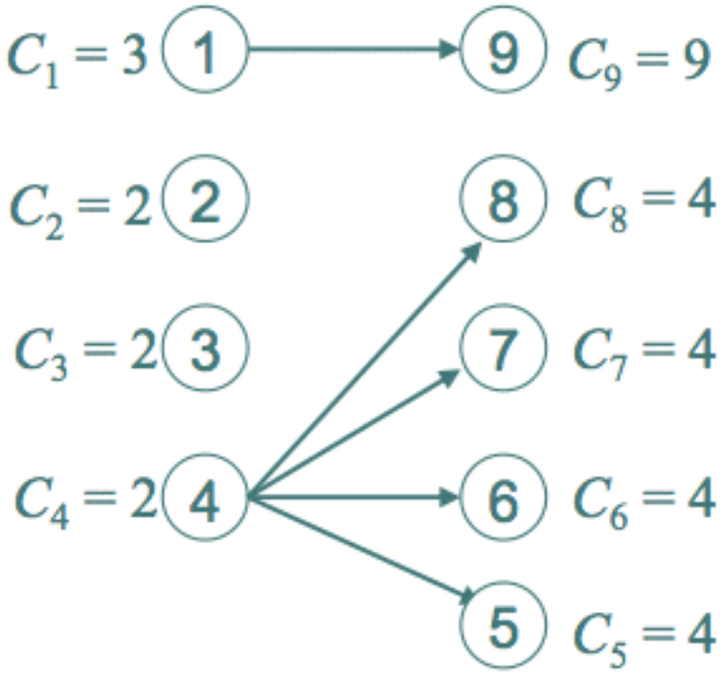


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

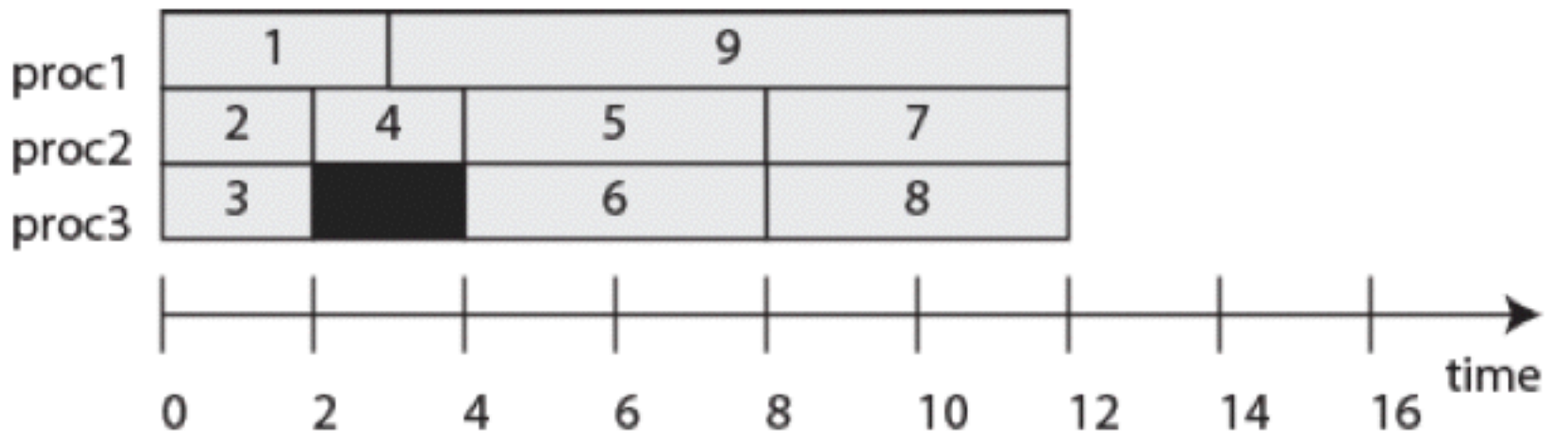


Reducing the computation times by 1 also results in a longer execution time.

Richard's Anomalies

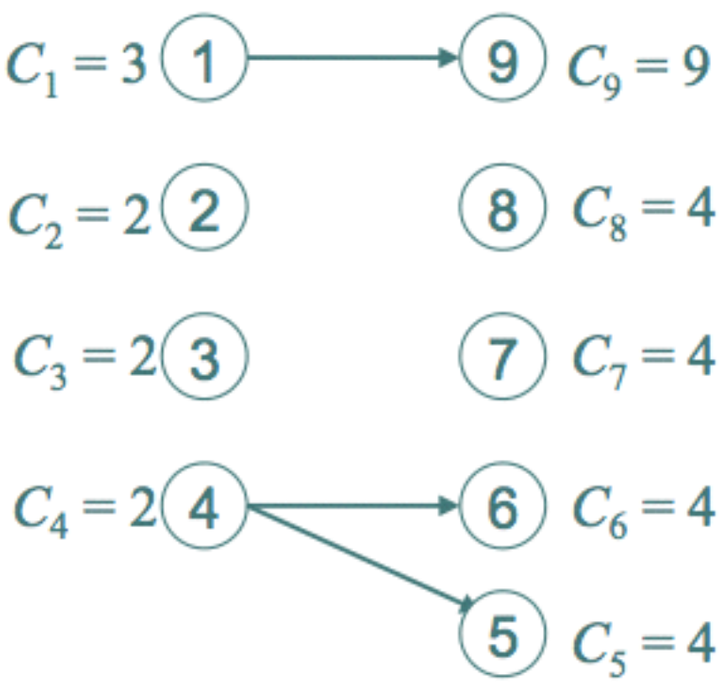


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

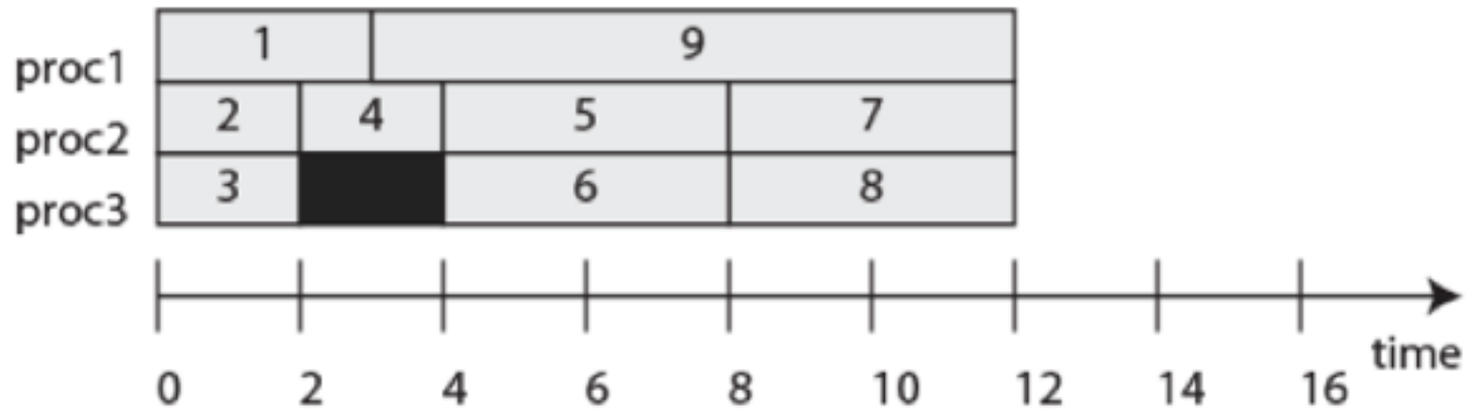


What happens if you remove the precedence constraints (4,8) and (4,7)?

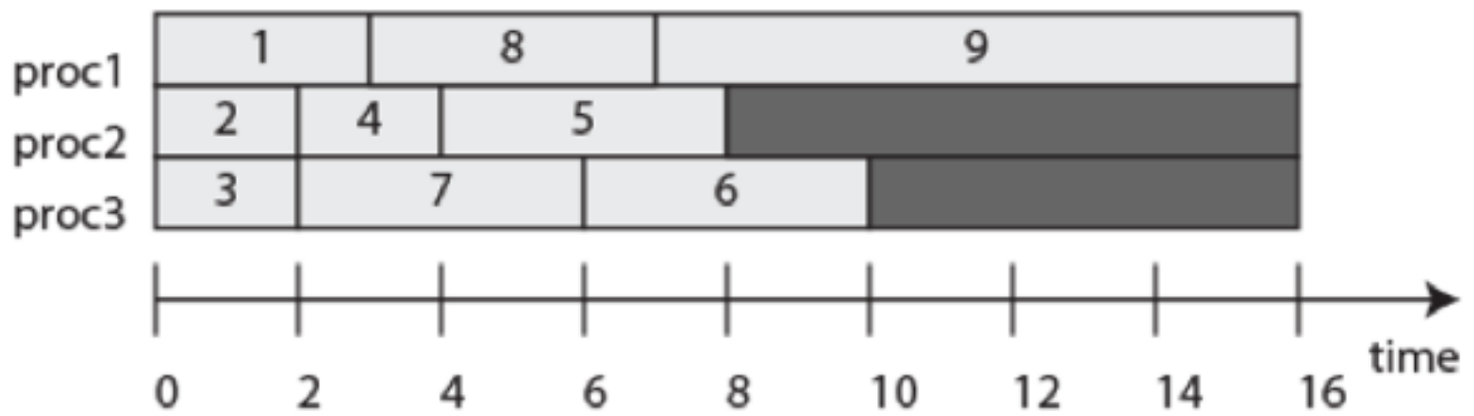
Richard's Anomalies: Weakening the precedence constraints



9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

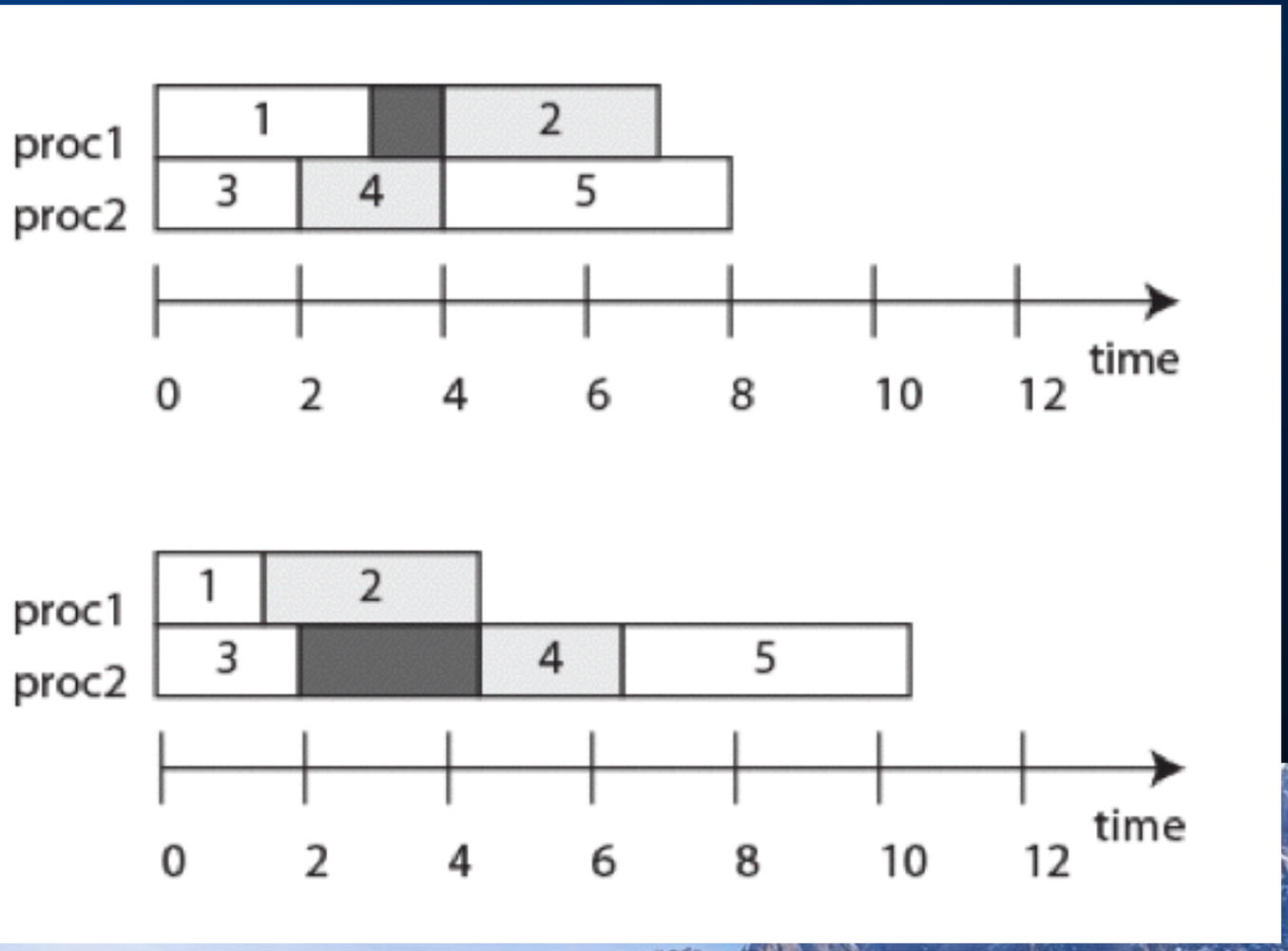


Weakening precedence constraints can also result in a longer schedule.



Richard's Anomalies with Mutexes: Reducing Execution Time

- Assume tasks 2 and 4 share the same resource in exclusive mode, and tasks are statically allocated to processors. Then if the execution time of task 1 is reduced, the schedule length increases:



Conclusion

- Timing behavior under all known task scheduling strategies is brittle. Small changes can have big (and unexpected) consequences.
- Unfortunately, since execution times are so hard to predict, such brittleness can result in unexpected system failures.

References

- Edward Ashford Lee, Sanjit Arunkumar Seshia. Introduction to Embedded Systems: A Cyber-Physical Systems Approach, chapter 11. Lulu.com. (嵌入式系统导论：CPS方法)
- Embedded Software Architecture. (tss)

Thank you!