Which of the following would you most prefer?

A: a puppy, B: a pretty flower from your sweetie, or C: large properly formatted

Embedded Operating Systems

Choose!

ENGN8537

Embedded Systems

Overview

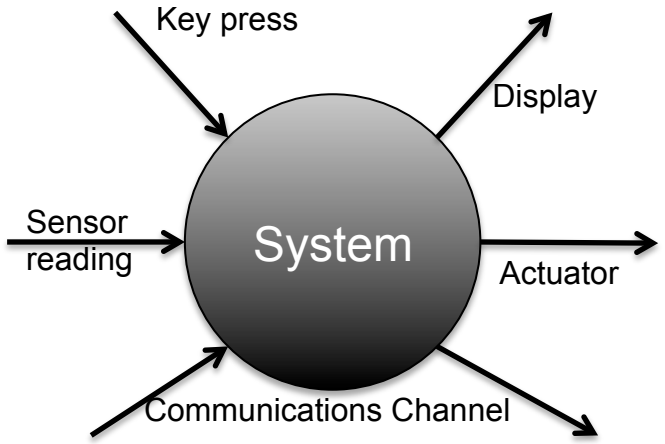
- Operating Environments
- Concurrency in RT Systems
- Scheduling
- Locking and Synchronization

Operating Environments

The most likely situation for embedded software has the system interacting with several unpredictable input/output activities.

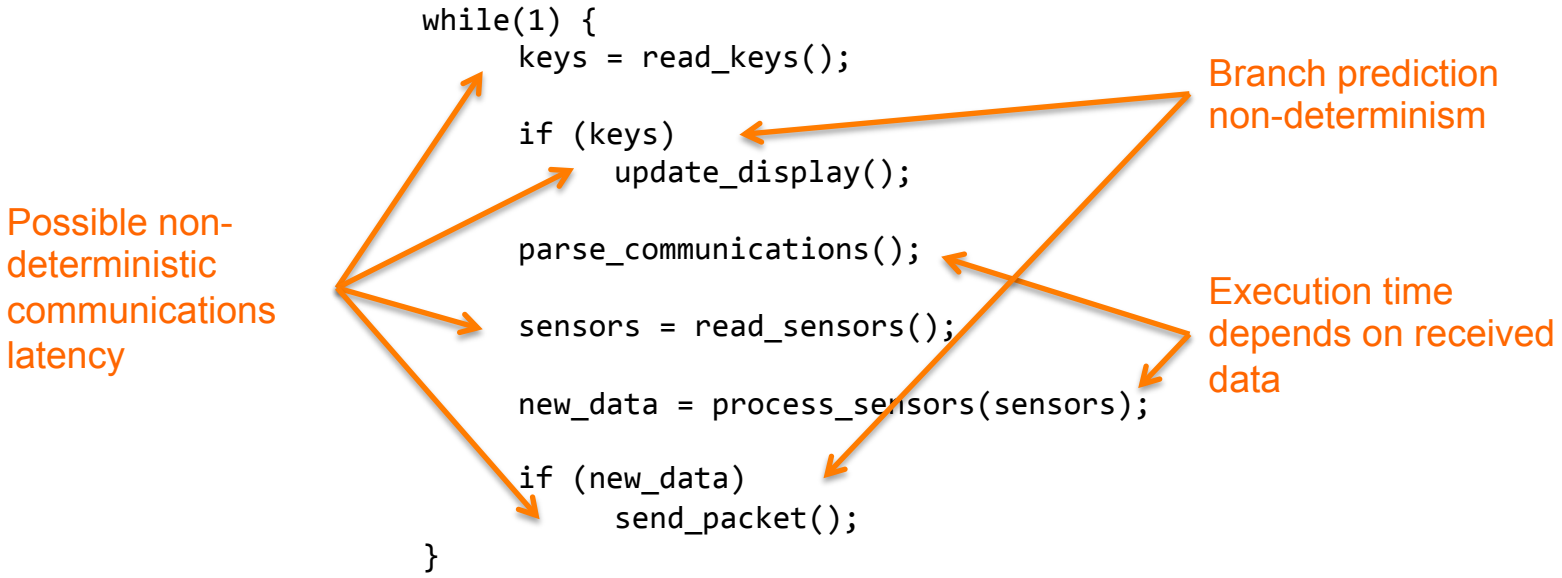
Traditionally a single loop will deal with each activity in turn with the assumption that the loop executes with a time scale much shorter than any of the events of interest

```
while(1) {
      keys = read_keys();

      if (keys)
            update_display();

      parse_communications();

      sensors = read_sensors();

      new_data = process_sensors(sensors);

      if (new_data)
            send_packet();
}
```



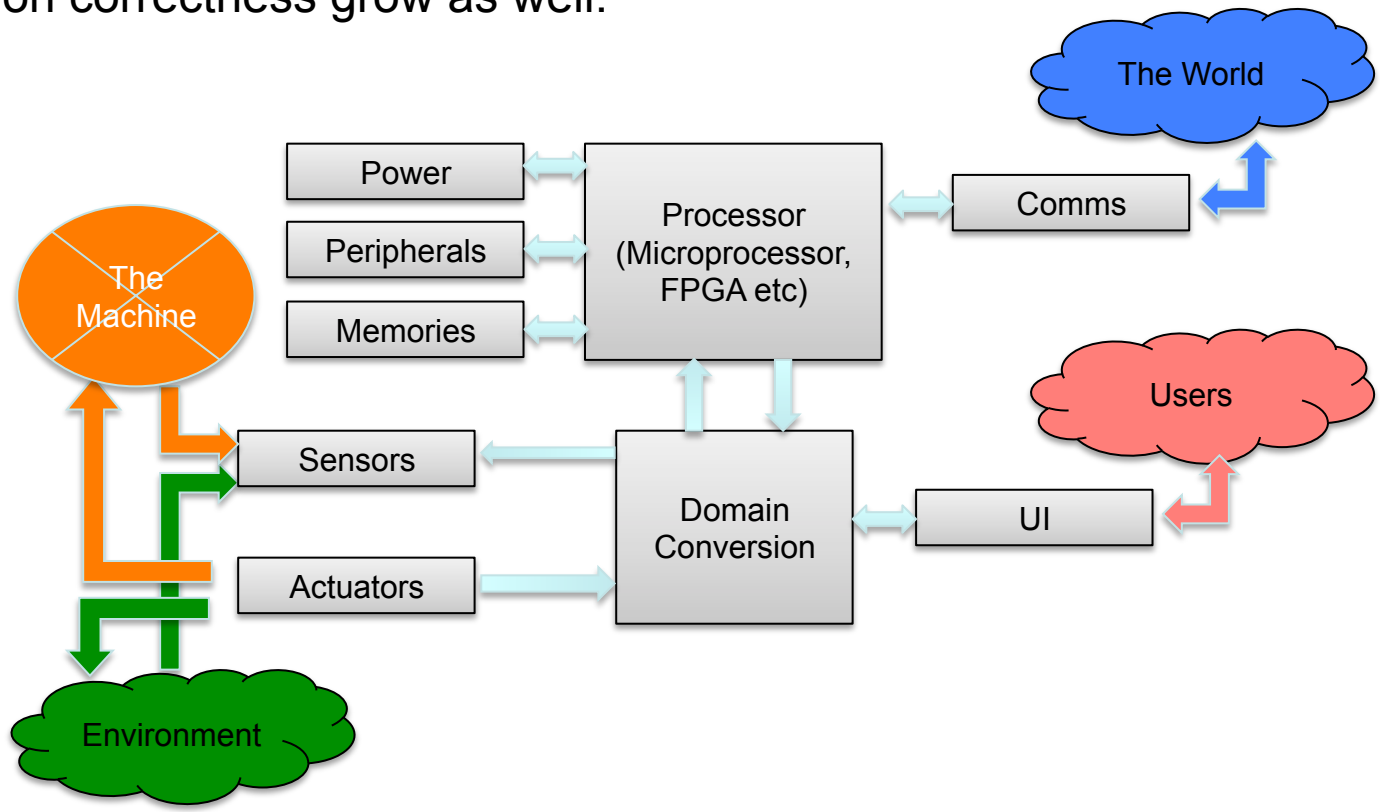
Operating Environments

This is generally fine for simple systems with limited processor capabilities and completely provable (or at least bounded) input sequences. It's also useful as a 'quick hack' to test the functionality of portions of the embedded device. However:



Operating Environments

As the system complexity grows, the difficulties associated with proving execution correctness grow as well.



Operating Environments

There's no central location that can 'force' correctness of execution except the programmer's head, and it has to do it off-line.

A key observation: Not every action has the same requirements regarding correct execution. If there were a central piece of code that knew about execution priorities and could control execution flow, it could ensure at run time that the programmer's wishes were being carried out.

Operating Environments

These central pieces of code are termed ‘Executives’ or ‘Schedulers’ and are part of an Operating Environment. In order to run in this environment, the embedded software should be split up as follows:

- Activity: The base unit of code that performs a given action, or portion of an action. No activity can forcefully be ‘preempted’ by another activity and all activities have direct access to each other’s resources (including data).
- Thread: A container for one or more activities that can be ‘preempted’ by another thread at the request of the scheduler.
- Task: A container for one of more threads where no activity or task in one thread has direct access to the resources of an activity or task in another thread.
- Application: A container for one or more tasks

Operating Environments

By using threads or tasks, the programmer can defer decisions regarding the order in which actions should be completed until run time, when all the variables are known.

Threads and tasks, which can be preempted, also protect against timing overruns; if something is taking longer to run than it should, the executive can take appropriate action asynchronously

- Temporarily give the CPU to another, higher priority task
- Send a signal to the misbehaving task ‘warning’ it of its bad behaviour
- Terminate the misbehaving task but give it a chance to clean itself up
- Kill the misbehaving task immediately

… but what if the misbehaving task has some sort of exclusive use of a resource? Killing it prematurely may leave that resource completely unusable to other activities.

Operating Environments

Multithreaded applications also have another major advantage in modern embedded systems:

They can transparently take advantage of multicore processors!

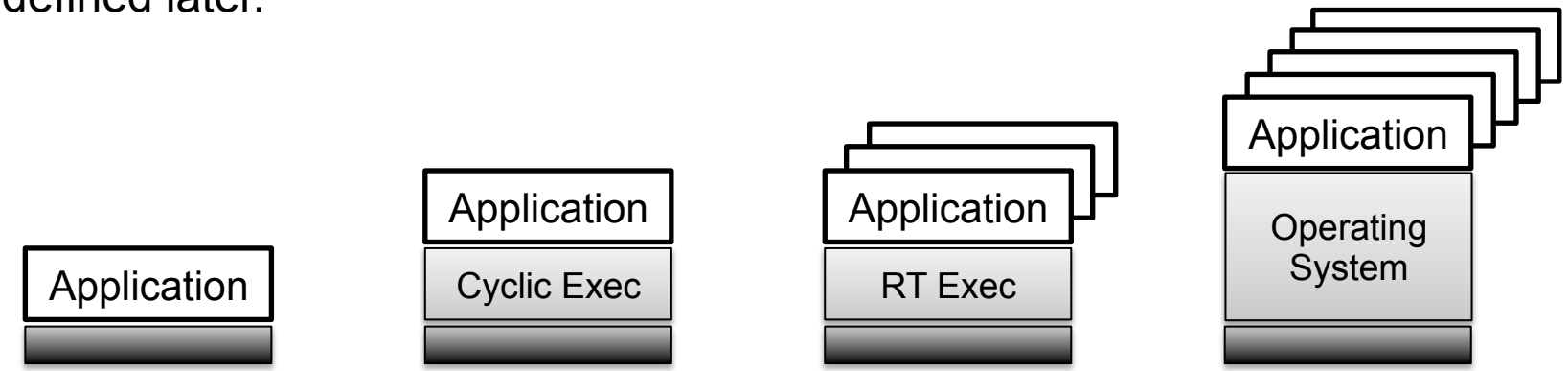
In traditional multithreading, the idea of concurrency is present but isn't really a reality in that there was only one CPU therefore at any instant of time to an external observer, only one thread was running. An external observer might be a peripheral or memory, not just a conscious entity!

The introduction of multicore CPUs have uncovered a whole new range of bugs and false assumptions as this is no longer the case.

Operating Environments

An Operating Environment is made up of the executive, primitives to control access to resources, devices drivers and other 'general purpose' utilities. The central part of the operating environment of an embedded system is the executive though, so from here we will use the terms 'operating environment', 'scheduler' and 'executive' interchangeably.

The term 'operating system', distinct from operating environment, will be defined later.



Bare Metal

Using a Cyclic Executive

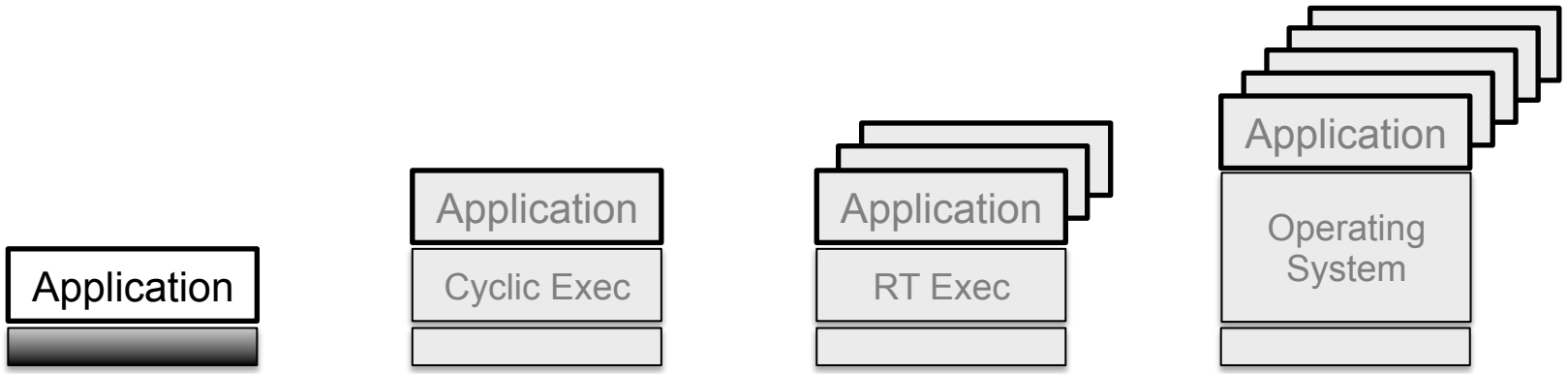
Using a Real Time Executive

Multi-tasking Operating System

Operating Environments

Bare Metal systems are those discussed earlier in the lecture. They may have library support, device drivers and other utilities, however there is no central executive making decisions about how and when activities should be performed.

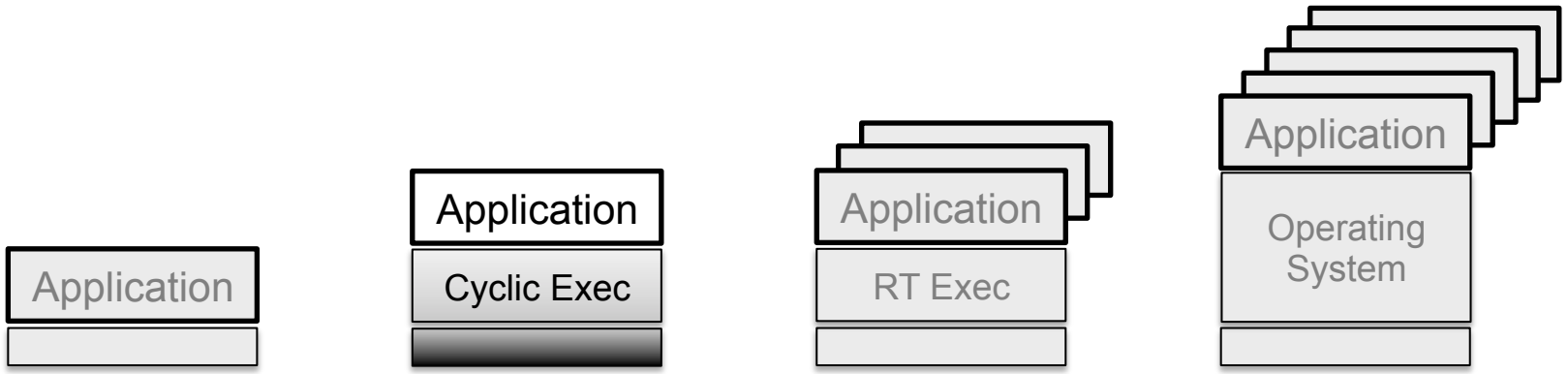
The correct operation of these systems must be completely proven (and provable!) off-line by the programmer.



Operating Environments

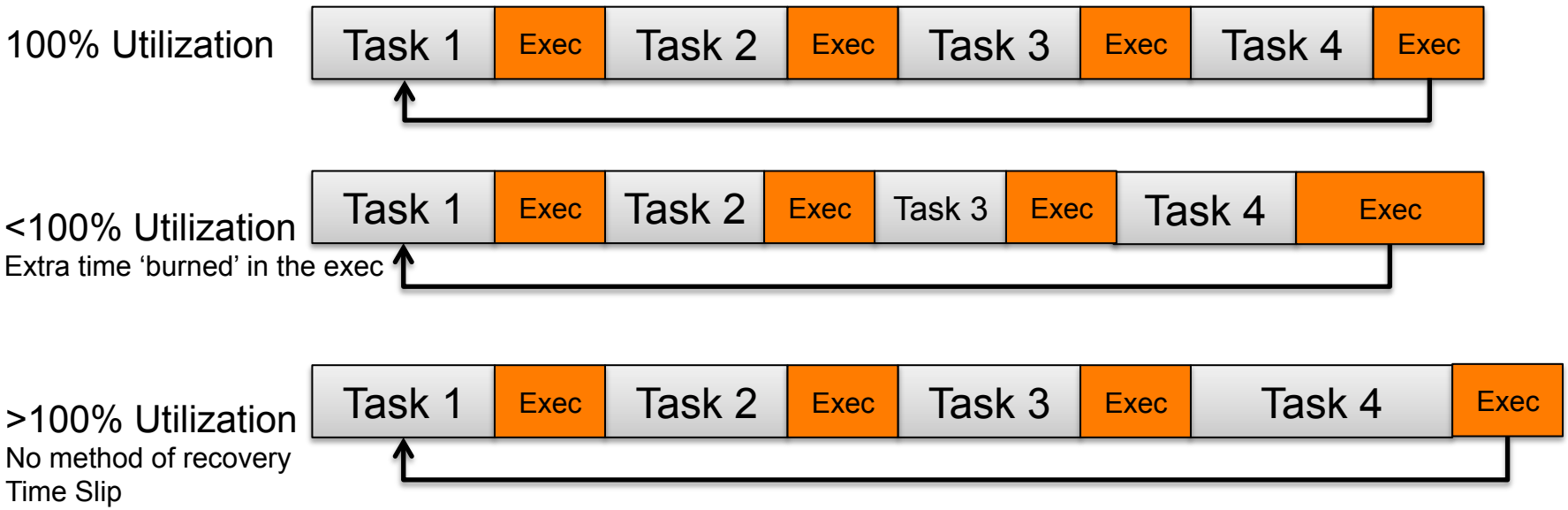
A Cyclic Executive can control only the execution of ‘activities’; that is, it doesn’t have the power to preempt the execution of any piece of code, only choose what happens once the current piece of code has finished running.

This type of scheduling is termed ‘cooperative’ and has the advantage that the executive need not be able to save state halfway through an execution. Everything is sequential and provable so long as the longest possible activity execution time is shorter than the shortest possible gap between critical events.



Operating Environments

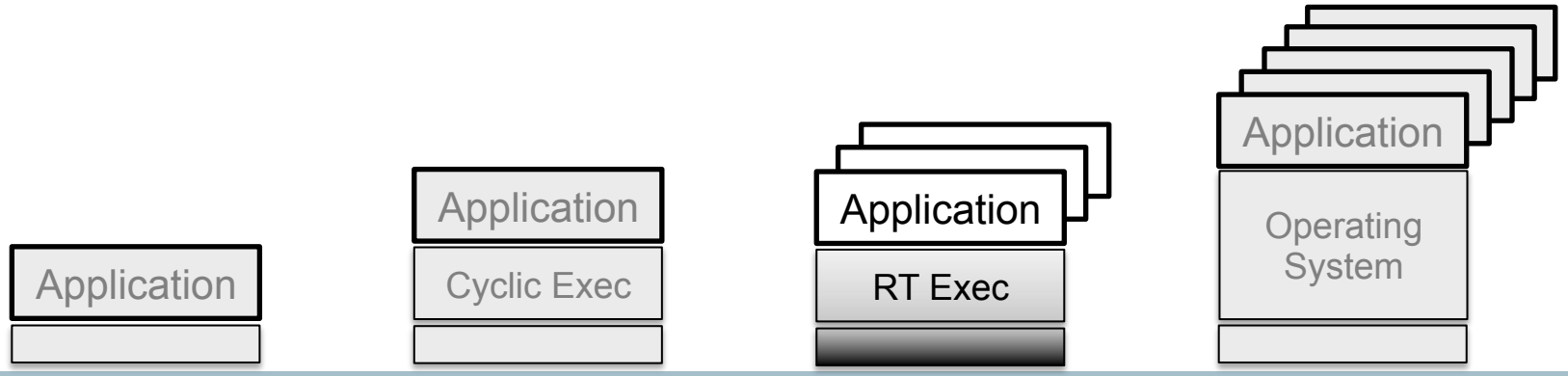
A Cyclic Executive may be implemented in a few different ways. The most simple way forward is to have a simple ring of actions that the executive calls in turn.



Operating Environments

Real Time executives are the first of the schedulers we've seen that may offer preemption.

Preemption is the ability of the scheduler to evict the current thread from the CPU and hand the CPU over to another thread instead. In doing so, it must know how to save the state of the preempted thread so that when it is later given the CPU back, it can pick up where it left off with no gaps in its world knowledge. To any given thread, preemption just seems like one of its instructions took an unusually long time to execute (with some exceptions we'll see later).



Operating Environments

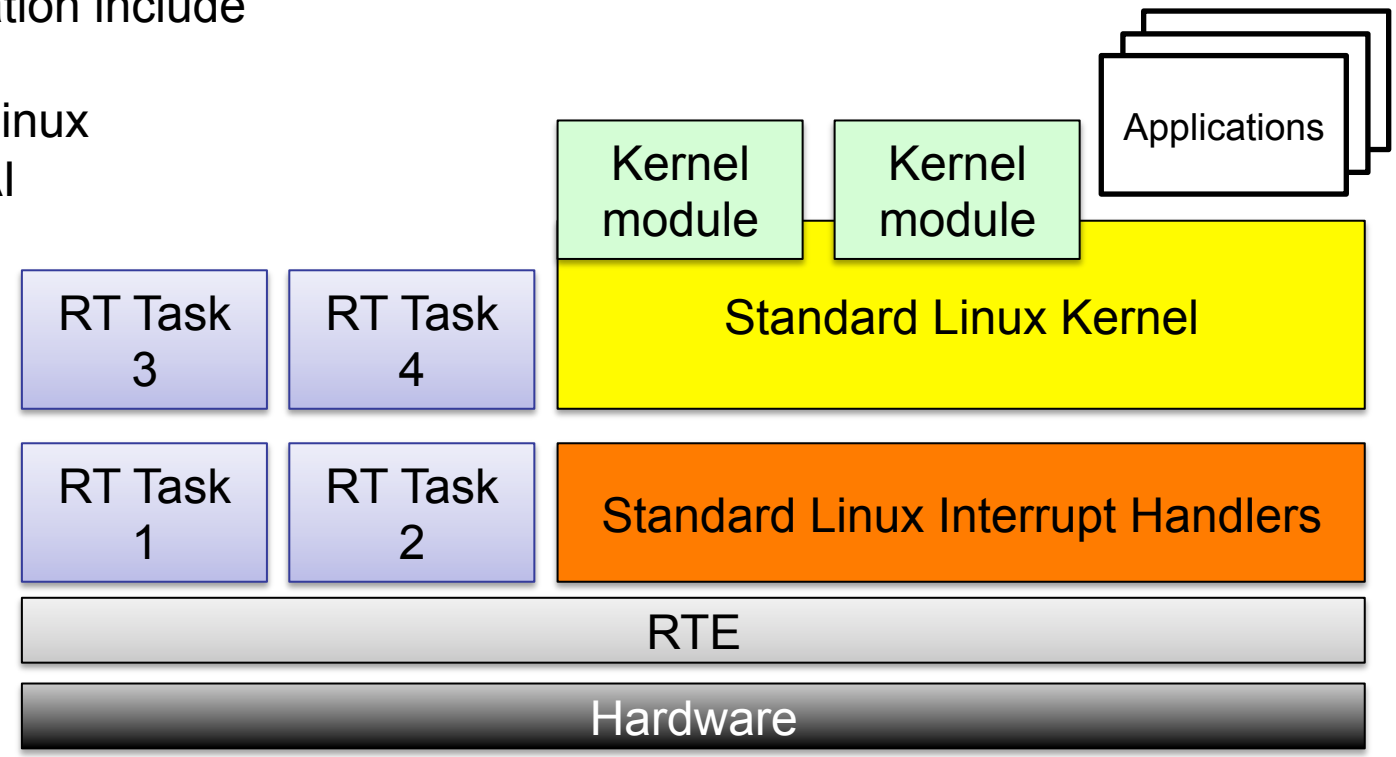
Real Time Executives form the core of many embedded operating environment used in the world today.

- FreeRTOS
- VxWorks
- VRTX
- QNX
- LynxOS
- MicroC/OS
- and many others

Operating Environments

In addition, it's not uncommon to run a full operating system as a thread inside the RTE! This method can be used to improve the response time of desktop operating systems such as Linux. Suitable RTEs for this application include

- RTLinux
- RTAI



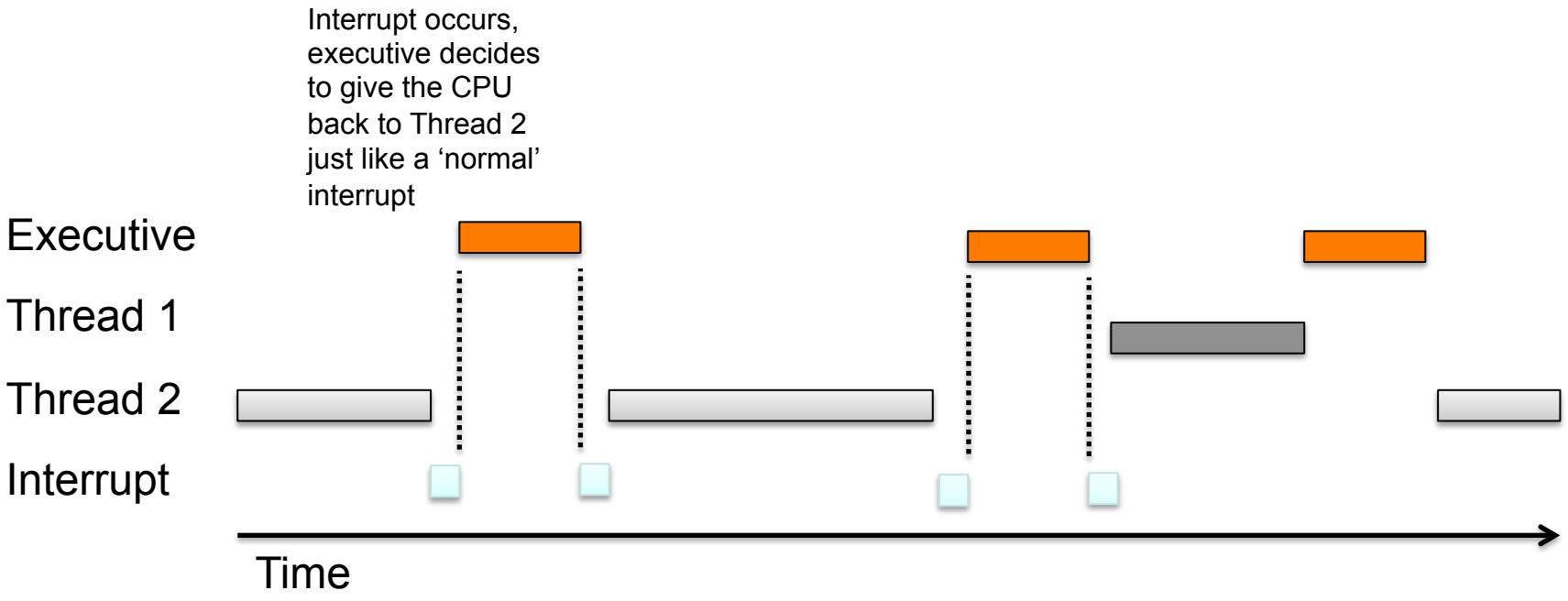
Operating Environments

In order for the executive to make the decision to preempt the running task, it requires the CPU itself. The executive must preempt the running task to determine whether it should preempt the running task?!

Operating Environments

In order for the executive to make the decision to preempt the running task, it requires the CPU itself. The executive must preempt the running task to determine whether it should preempt the running task?!

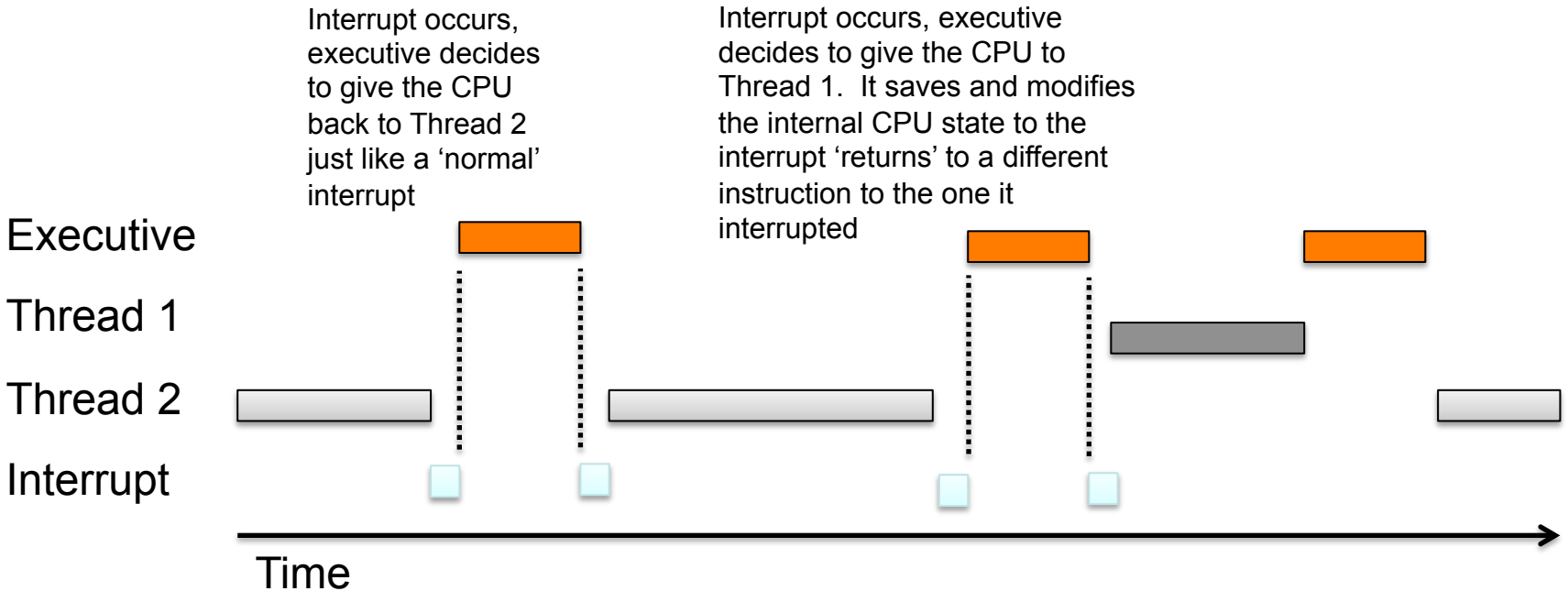
The answer is to use a special interrupt, generally triggered from a timer.



Operating Environments

In order for the executive to make the decision to preempt the running task, it requires the CPU itself. The executive must preempt the running task to determine whether it should preempt the running task?!

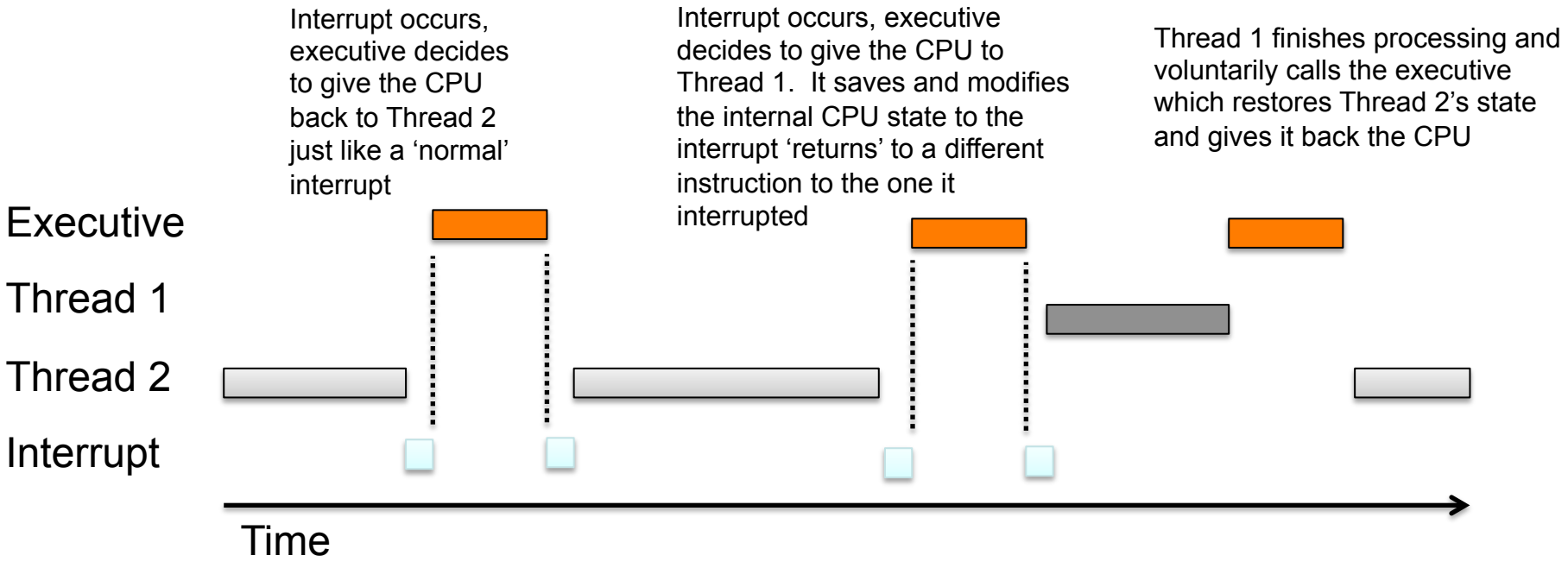
The answer is to use a special interrupt, generally triggered from a timer.



Operating Environments

In order for the executive to make the decision to preempt the running task, it requires the CPU itself. The executive must preempt the running task to determine whether it should preempt the running task?!

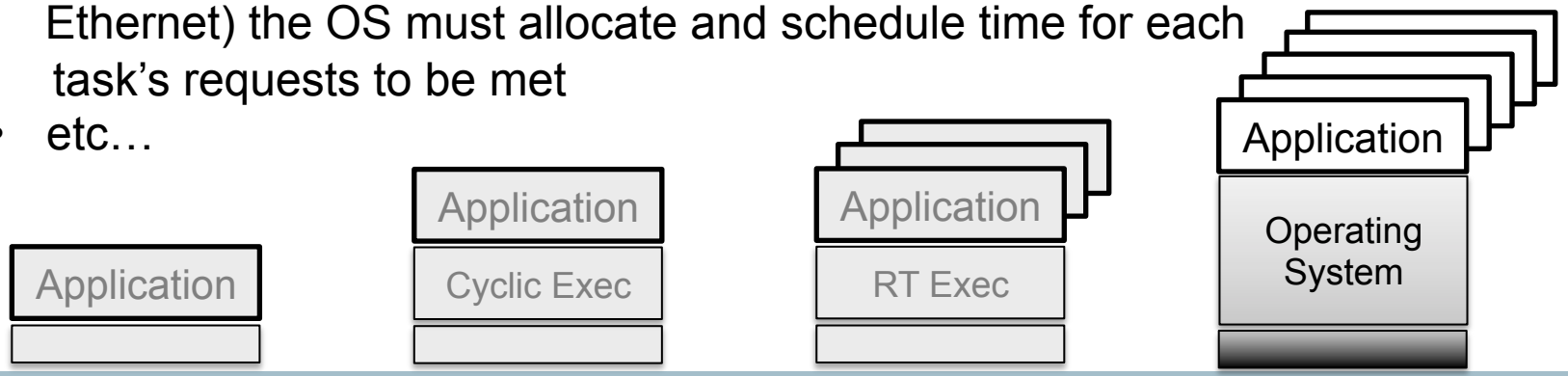
The answer is to use a special interrupt, generally triggered from a timer.



Operating Environments

Real Time Executives can generally only work with threads, not tasks. A full multi-tasking environment generally requires an operating system. In addition to simple scheduling, tasks require complete isolation of resources which in turn requires:

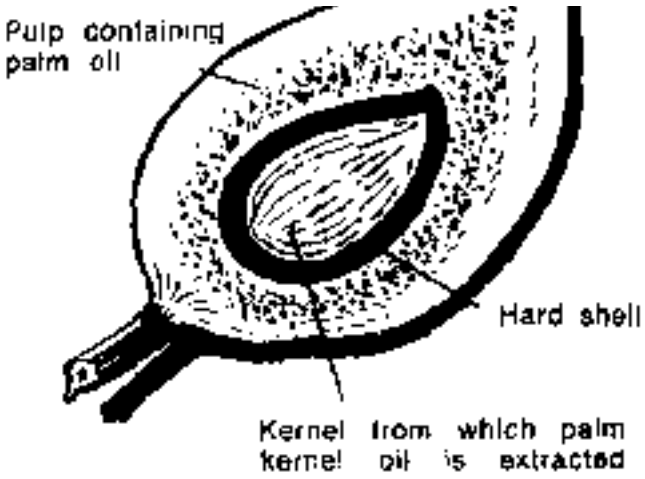
- Virtual Memory: One task's memory location 0x10000000 will be different from another task's memory location 0x10000000
- Storage Arbitration: The operating system must manage file systems on underlying storage devices and allocate and schedule time for each task's requests to be met
- Comms Arbitration: For comms links supporting multiple end points (eg Ethernet) the OS must allocate and schedule time for each task's requests to be met
- etc…



Operating Environments

The operating system was originally divided in to exactly two parts, the Kernel and Shell. The Kernel was the portion of code that provided core functionality and the shell gave the user interface. The user interface is now extended beyond the shell, however the concept of the kernel is still very much in tact.

For example, Linux is not an operating system, it's a kernel. The operating system is formally GNU/Linux where GNU is the name of the collection of tools that exist outside the kernel.



Operating Environments

The non-kernel components of a modern operating system are referred to as the userspace. This will include

- Windowing system
- Network manager
- Service manager
- Application stack
- Internationalization tools (I18N)
- Power policy tools

A commonly used phrase is that the kernel provides mechanism, userspace provides policy.

Concurrency in RT Systems

We have asserted that the existence of a preemptive executive can improve the determinism of the system by considering all possible options for allocation of CPU time in order to choose the best order in which to run threads.

In addition to ordering access to the CPU, the executive can order access to other shared resources (such as the network) to the same end.

It can also provide a central location at which to calculate the worst case behaviour of the system.

This prediction can be used at compile time to verify that the overall temporal requirements of the application can be met, or at run time to permit or reject tasks as they apply for resources.

Concurrency in RT Systems

We will examine two different scheduling algorithms commonly used in real time executives:

Earliest Deadline First (EDF)

- Dynamic priority
- Preemptive

Rate Monotonic Priority (RMP)

- Static priority
- Preemptive

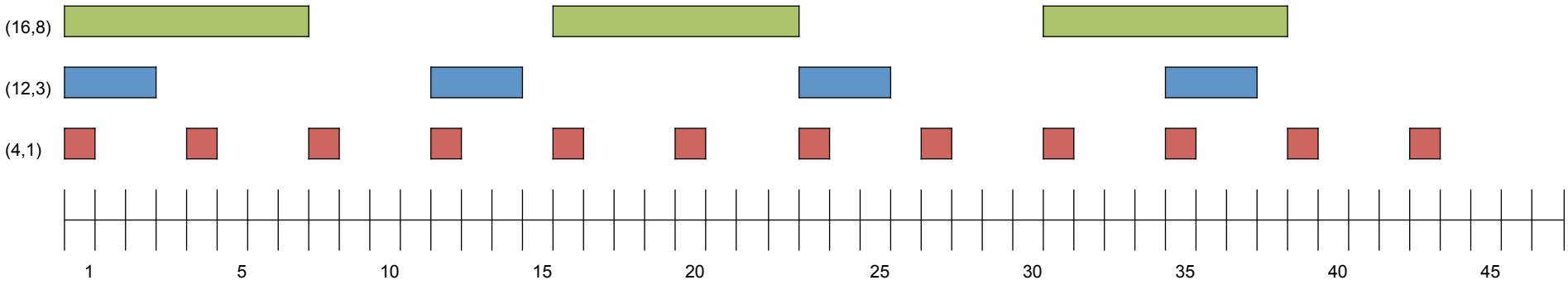
Concurrency in RT Systems

For our analysis we will assume the following:

- All threads are periodic with known period
- Threads are independent
- Overhead of the executive is negligible
- All threads have deadlines equal to their cycle period (i.e. the last iteration of a thread has to have completed before the next iteration starts)

Concurrency in RT Systems

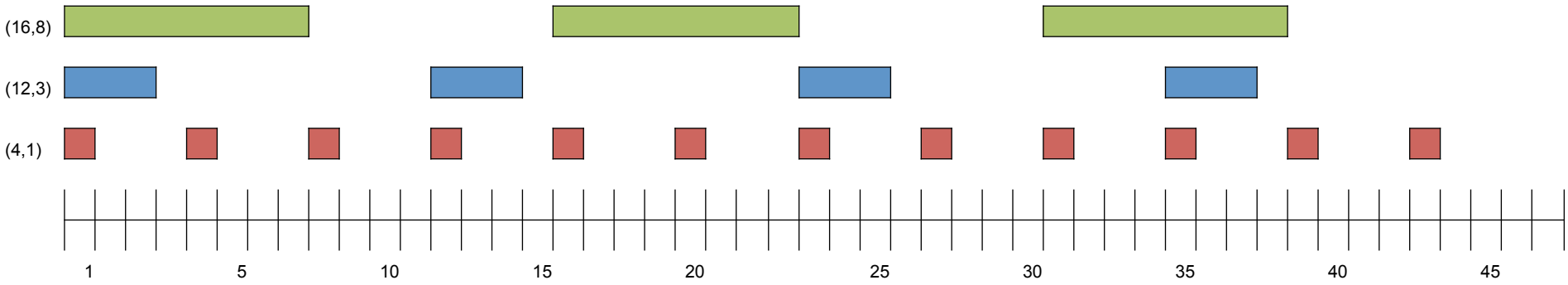
Example Requested Times



Is this schedulable under any algorithm?

Concurrency in RT Systems

Example Requested Times

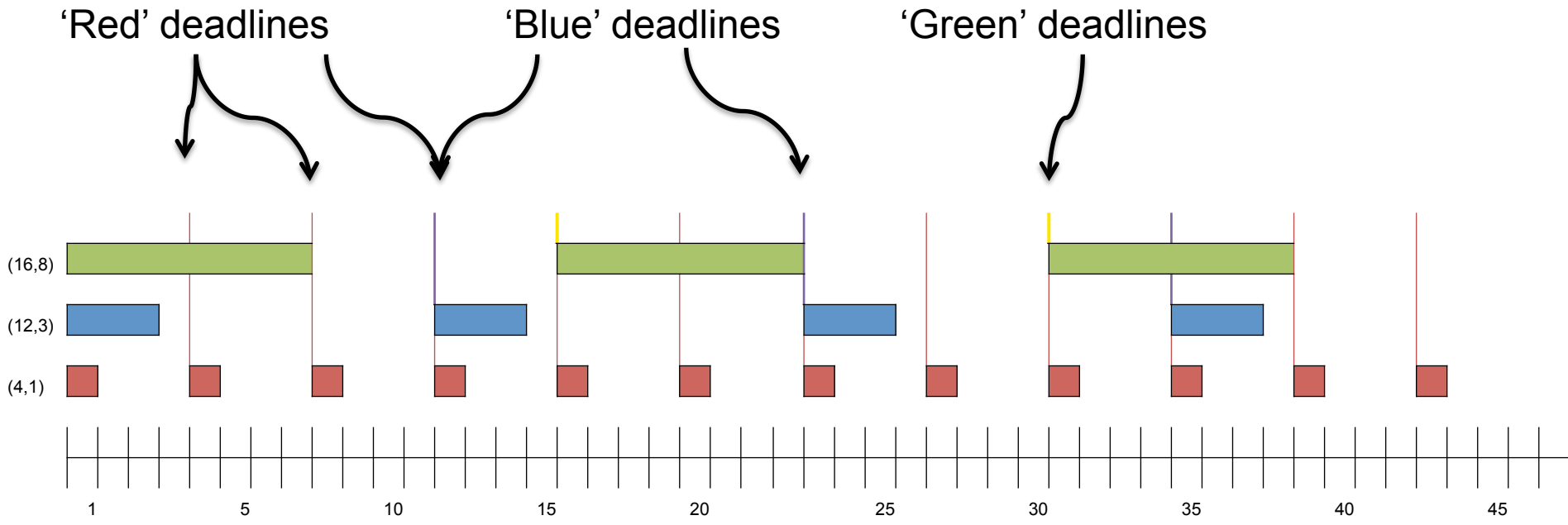


Is this schedulable under any algorithm?

$$\frac{1}{4} + \frac{3}{12} + \frac{8}{16} = 1$$ ✔

Concurrency in RT Systems

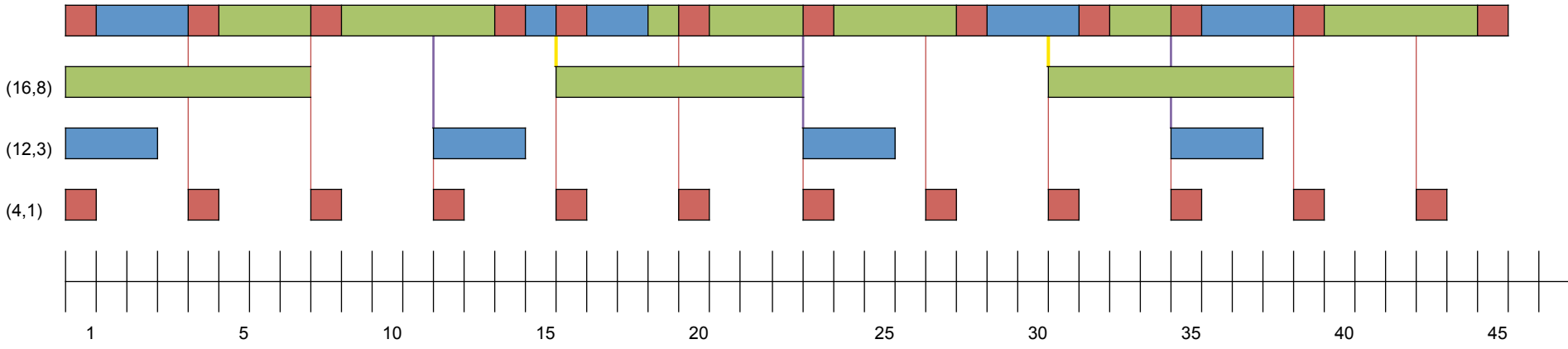
Example Earliest Deadline First



Concurrency in RT Systems

Example Earliest Deadline First

Execute the current thread until
1. It finishes or
2. Another thread is found with an earlier deadline



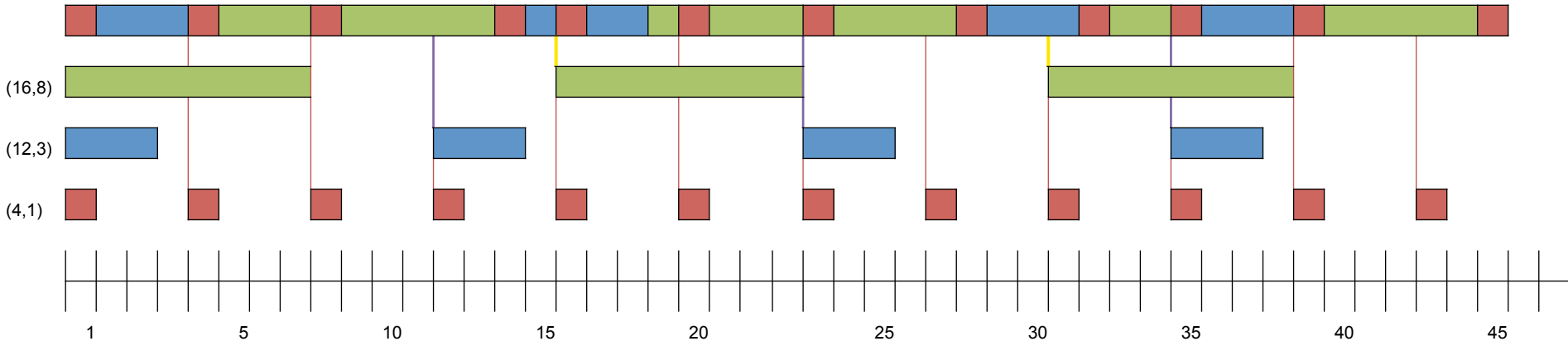
Avoid context switches in the case of equal deadlines

Concurrency in RT Systems

Example Earliest Deadline First

EDF can provide maximal utilization; that is, it it's safe up to and including 100% processor usage

➔ So long as each thread conforms exactly to its execution time limit

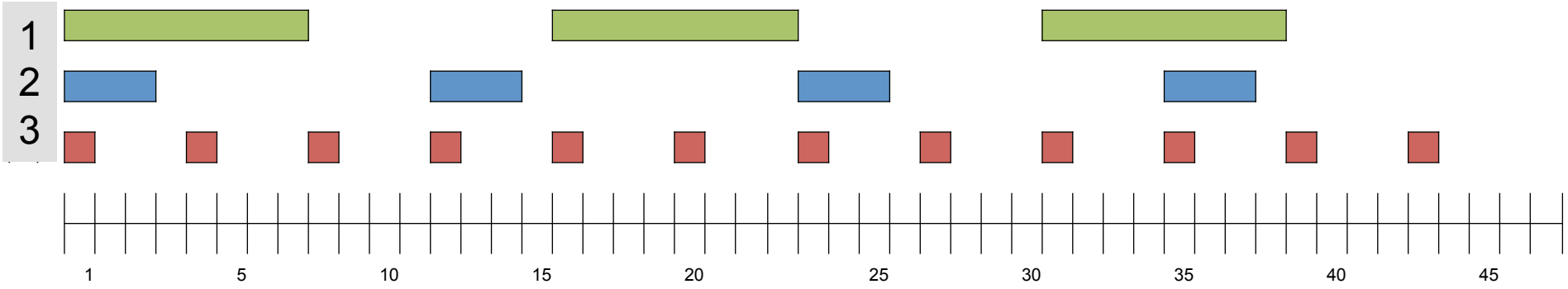


Concurrency in RT Systems

Example Fixed Priority Rate Monotonic

Assign each thread a priority based on its cycle time such that the threads that execute often get high priority.

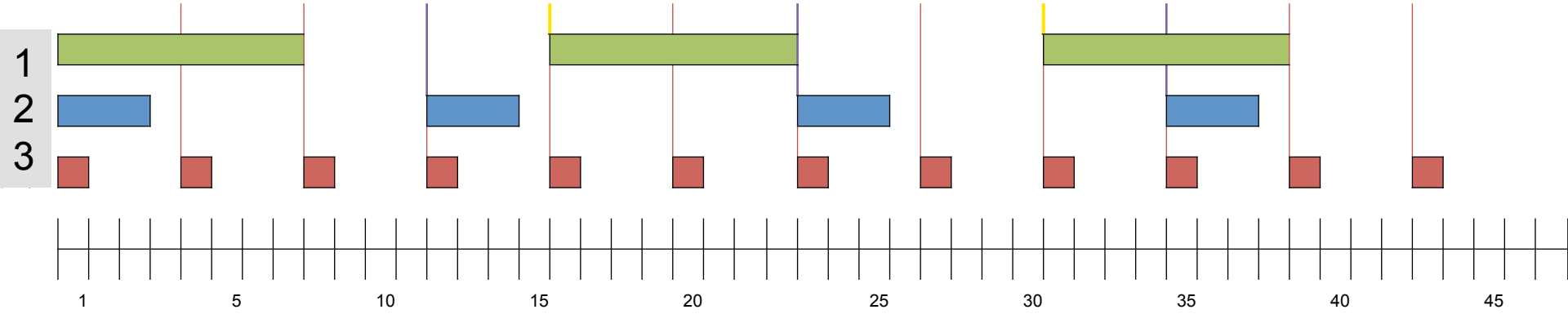
Rate monotonic ordering is optimal within the framework of fixed priority algorithms in that if a set of tasks can be scheduled with any fixed priority algorithm, they can be scheduled with rate monotonic ordering



Concurrency in RT Systems

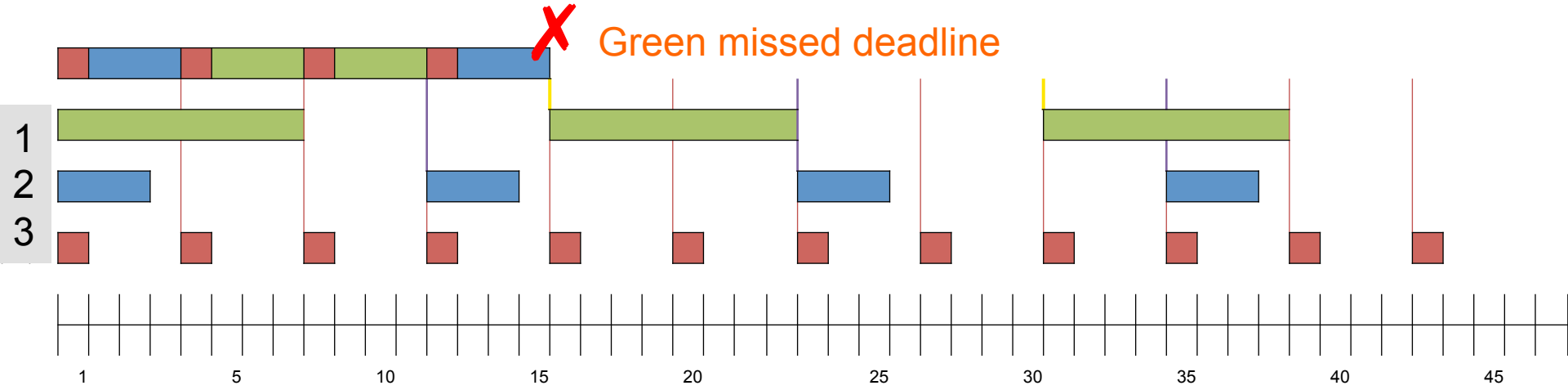
Example Fixed Priority Rate Monotonic

Same deadlines and scheduling as the previous case, but this time we have fixed priorities assigned to each of the three threads.



Concurrency in RT Systems

Example Fixed Priority Rate Monotonic

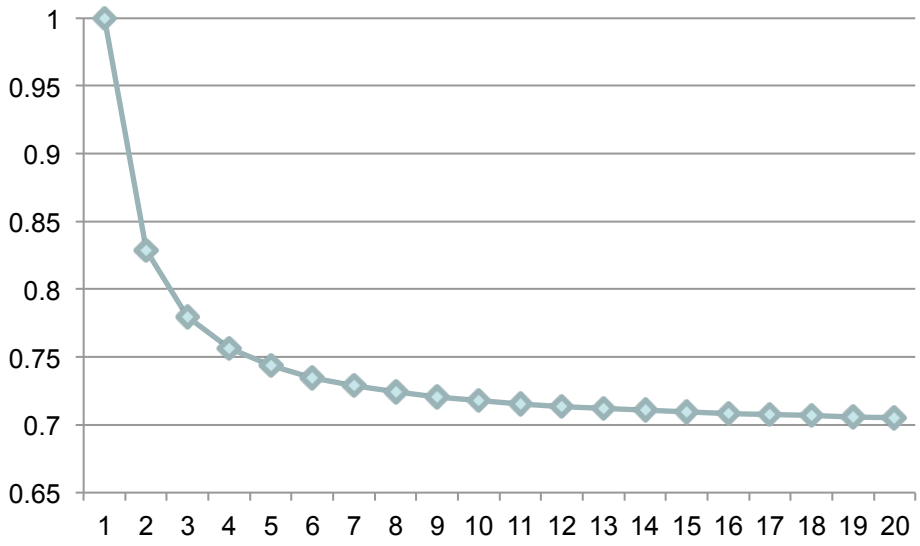


Concurrency in RT Systems

Example Fixed Priority Rate Monotonic

Fixed Priority scheduling cannot schedule to 100% utilization for an arbitrary number of threads! The following condition is a sufficient condition to determine maximum utilization, but not a necessary condition.

$$U = N\left(2^{\frac{1}{N}} - 1\right)$$



Concurrency in RT Systems

Earliest Deadline First vs. Fixed Priority Rate Monotonic

- EDF can handle higher utilization (full) than FPS
- FPS is easier to implement and has less run-time overhead

- Graceful degradation features (when resource is overbooked)
  - FPS: Threads with lower priority will always miss their deadlines first
  - EDF: Any thread may miss their deadlines and can cause a cascade of missed deadlines

Concurrency in RT Systems

We have only considered scheduling where all threads are hard real time.

Scheduling with soft real time or non-real time threads won't be considered theoretically in this course, however we'll see one practical implementation when we examine the Linux Kernel's scheduler in the next lecture.

Concurrency in RT Systems

The strength of tasks (as opposed to threads) is that a task has a completely isolated view of the machine. It has its own memory segment and no data is shared with other tasks.

Threads on the other hand do have access to each other's data areas.

First we'll look at the mechanisms available for inter-task communication.

Concurrency in RT Systems

Tasks will need to communicate with each other, either locally or remotely.

Remote communications is almost always over a network, however local communication can be done in a number of different ways.

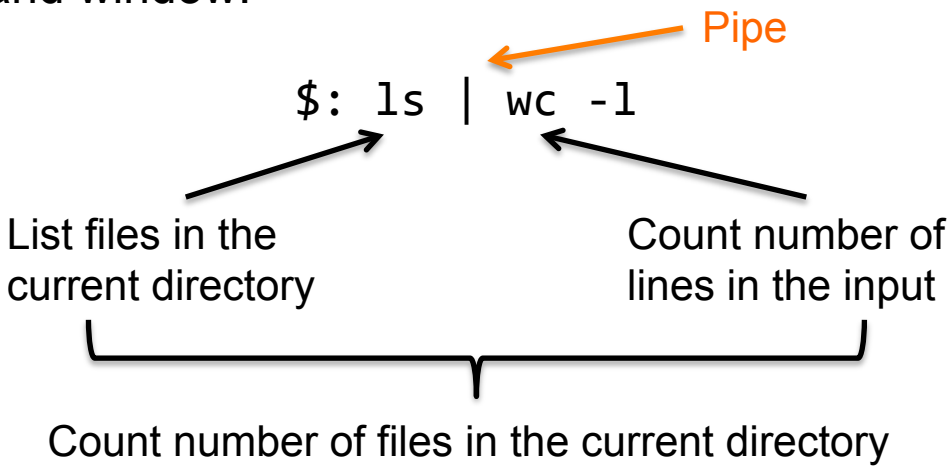
- Pipes
- Sockets
- Signals
- Shared Memory

Concurrency in RT Systems

Pipes

Pipes are FIFO channels between tasks. They usually handled in a similar way to traditional files, with each end of the pipe opening it for 'reading' or 'writing', then closing upon completion.

If you've used Linux (or any UNIX-style shell) you've probably used pipes in the command window:



Concurrency in RT Systems

Pipes

Pipes are unidirectional in nature and can only exist between tasks that share a file descriptor namespace. This is important, as it means the pipe has to be created by the task, or a parent of the tasks, that will use it.

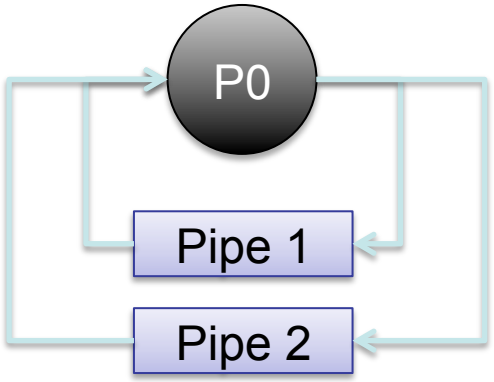
In the shell example, this works because the shell itself is the parent of the two applications that require use of the pipe.

A bidirectional pipe can be created by using a pair of pipes and agreeing beforehand which pipe will run which direction.

Concurrency in RT Systems

Pipes

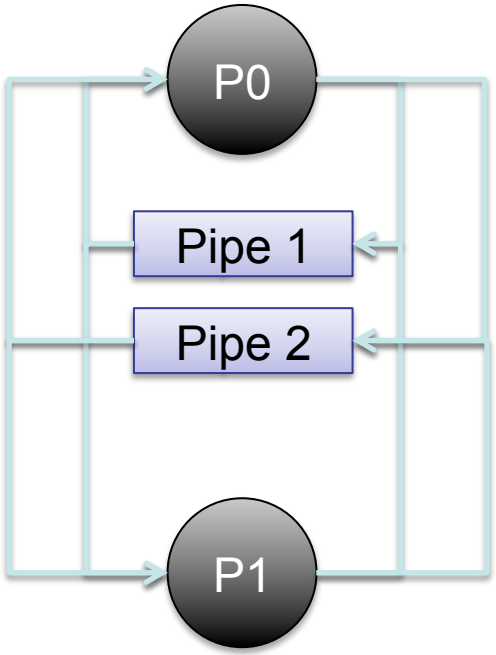
1. First process creates two pipes



Concurrency in RT Systems

Pipes

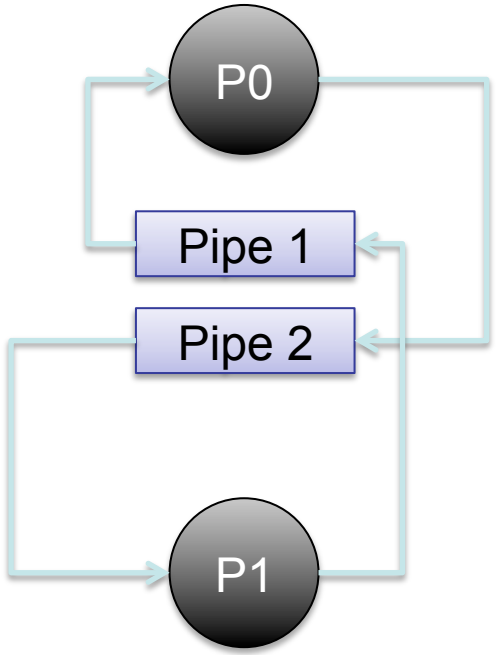
1. First process creates two pipes
2. First process creates the second
   this shares the same pipes



Concurrency in RT Systems

Pipes

1. First process creates two pipes
2. First process creates the second
   this shares the same pipes
3. Child and parent close different halves of each pipe, leaving a bidirectional link



Concurrency in RT Systems

Sockets

Sockets are the mechanism used for network communications in POSIX-like environments (including Linux/UNIX etc.). When a socket is created, you can specify the type such as “TCP”, “UDP” or “UNIX” sockets, where UNIX sockets stay on the local computer only.

Sockets act in a similar way to pipes but

- Are bidirectional
- Tasks need not have a common parent

Concurrency in RT Systems

Sockets

Depending on socket type, may be able to:

- Guarantee delivery
- Detect delivery failure
- Provide broadcast mechanisms
- Provide client/server architectures
- etc…

There are more efficient ways to communicate between tasks on the same machine, but the flexibility and power of sockets means they continue to find use in this application.

Concurrency in RT Systems

Sockets

Sockets are so important for local communications, Google has recently (August 2012) submitted patches to the Linux Kernel implementing a feature called ‘TCP Friends’.

The core of the TCP Friends patch is simple: If both ends of the TCP socket are found to be on the same machine then the data is immediately queued for reading at the other end, bypassing the network stack entirely.

Of course there’s plenty of ‘faking’ that the operating system has to do, pretending to be a network stack, but the performance improvements seem significant.

Concurrency in RT Systems

Signals

Signals are a way for tasks to communicate events, rather than data. If a task knows the identifier of another task, it can send an event to that task.

Signals are chosen from a relatively small set of predefined values with specific meanings, a programmer is not usually free to choose their own signal meaning (though they can 'repurpose' existing signals if both ends of the communication agree).

Signals are delivered asynchronously to the target task which means they can be used to (temporarily or permanently) control the execution flow of the other program.

For example, the `dd` program can be used to copy large amounts of data on Linux systems. If another task sends `dd` the '`SIGUSR1`' signal, `dd` will pause the transfer and print out progress information.

Concurrency in RT Systems

Shared Memory

By default, tasks don't share memory. They can, however, elect to share some portion of memory explicitly if it suits them. This is a very simple method of communication but brings with it issues of synchronization:

```
int i;

i++;        |   i = 0;
```

What's the value of i? This is called a 'race' condition, as both threads 'race' to see who can change the value of i first.

But it's more subtle again: Depending on the architecture and instruction set, there's a chance that the increment or set operations will take multiple asm instructions to complete. The scheduler could jump in half-way through the operation and leave i in a state that's neither 0 nor the next number in sequence!

Concurrency in RT Systems

Race Conditions

Race conditions like this may occur when ever the following conditions are met:

- There's some degree of concurrency in the system
- Resources are shared between concurrent code paths
- The accesses to the resource are not atomic
- The accesses to the resource are not synchronized

Note that points 1 and 3 are related – if there's no concurrency, all actions are atomic!

Concurrency in RT Systems

Race Conditions

Almost all embedded systems have some degree of concurrency, either through preemptive scheduling or simply by interrupts.

Almost all embedded systems use shared memory implicitly as there's no concept of tasks or memory isolation.

So the question then becomes: Are the actions atomic? What can we do to synchronize?

Concurrency in RT Systems

Race Conditions

Atomic actions cannot be broken in to smaller pieces; that is, they either complete successfully or not at all.

At the most basic level, a single RISC instruction is generally atomic on a single processor.

But what about CISC (and hybrids)?

What about multiprocessor systems?

Concurrency in RT Systems

Race Conditions

In those cases it's almost impossible to assume atomicity, you have to build it. This effectively means disabling concurrency for the duration of the atomic action

```
save_and_disable_interrupts();
do_critical_work();
restore_interrupts();
```

Concurrency in RT Systems

Race Conditions

This approach can ruin response times as interrupts may be disabled for quite a long period, keeping the executive away from the CPU for that time and unable to make decisions.

It also doesn't work on multiprocessor systems as then there's a source of concurrency other than just interrupts. Disabling whole other processors is even more heavy handed. This is hard to solve without hardware support.

Concurrency in RT Systems

Race Conditions

On x86 multiprocessor machines, it isn't enough just to disable interrupts, you have to also lock the memory bus. Other CPUs can do what they need to, so long as they don't require a memory access. Once again, not very fine grained control of the problem.

More modern CPUs like ARM will provide a Memory Reservation Engine. A load instruction can take a 'reservation' on a piece of memory and the matched 'store' instruction at the other end can detect whether there's been any access to the shared piece of memory since that initial load. If there is, the store fails and the processor is instructed to retry the whole atomic action.

This seems wasteful, however race conditions are almost always very rare so optimizing for the case where there's no problem is generally sensible.

Concurrency in RT Systems

Synchronization by Semaphore

Semaphores or ‘spinlocks’ are a very simple solution to this problem. They may be implemented by the processor architecture as single asm instructions, or they may be implemented by a library using atomic asm instructions. Either way, they have the following semantics:

WAIT(lock)	Pauses execution of the calling thread until ‘lock’ is set to ‘free’
SIGNAL(lock)	Frees the lock

We then get a safe version of the code

```
                int i;
 wait(i_lock);      | wait(i_lock);
        i++;        | i = 0;
signal(i_lock);     | signal(i_lock);
```

Concurrency in RT Systems

Synchronization by Semaphore

Semaphores are simple but have two primary criticisms:

1. They are scattered around the code base. You have to be very careful that every possible time a particular variable is accessed, the correct lock is taken
2. Depending on implementation, they may be unfair.
    1. The waiting processes are placed on a queue and the 'front' one is woken up at each signal – Complex but fair
    2. All waiting processes are woken up at each signal and they race to try and claim the lock – Simple but can lead to starvation

Concurrency in RT Systems

Synchronization by Monitor

Another option for synchronization is by a special piece of code called a monitor.

```
                        int i;
locked_increment(i);  |   locked_set(i,0);
```

The data and locks are collected in one place and access is only by function calls. The functions may use semaphores internally, but their usage is centralised and easy to verify.

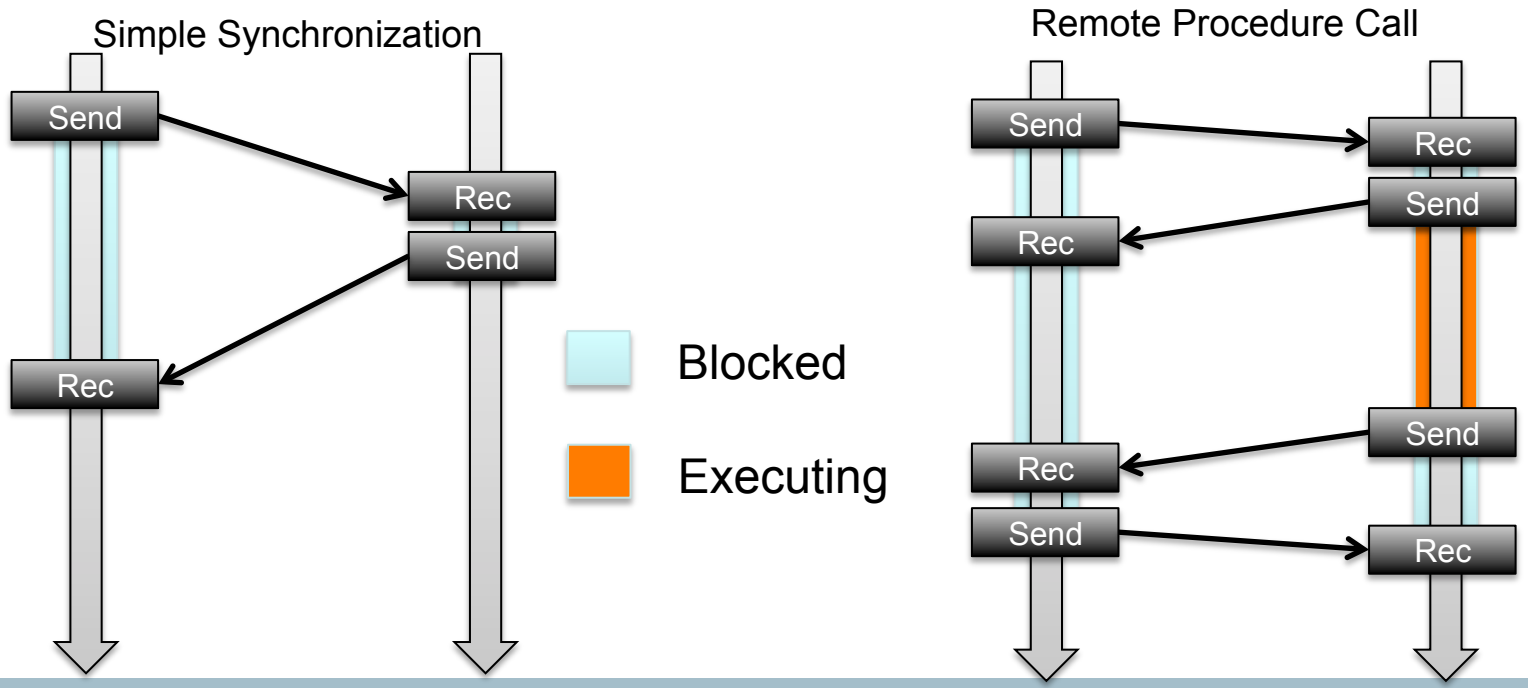
This centralisation also has another feature, it becomes possible to build things like locked lists that block the caller until a condition is met, such as there being space in the list.

```
locked_append(my_list, i);
```

Concurrency in RT Systems

Synchronization by Messages

A different approach to synchronization may be built out of message passing algorithms. These can be used to protect shared data, but also coordinate the processing of jobs, the calling of functions or other such functions



Concurrency in RT Systems

When ever there's locking involved, bad coding may lead to a deadlock.

There are many different types of deadlock, but the simplest to understand is the ABBA deadlock.

```
wait(lock_a);
wait(lock_b);
a = a + b;
                        wait(lock_b);

signal(lock_b);
signal(lock_a);
                        wait(lock_a);
                        a = b = 0;
                        signal(lock_a);
                        signal(lock_b);
```

Concurrency in RT Systems

When ever there's locking involved, bad coding may lead to a deadlock.

There are many different types of deadlock, but the simplest to understand is the ABBA deadlock.

```
wait(lock_a);
wait(lock_b);
a = a + b;

                    wait(lock_b);

signal(lock_b);
signal(lock_a);

                    wait(lock_a);
                    a = b = 0;
                    signal(lock_a);
                    signal(lock_b);
```

Some preemption points will work

Concurrency in RT Systems

When ever there's locking involved, bad coding may lead to a deadlock.

There are many different types of deadlock, but the simplest to understand is the ABBA deadlock.

Some preemption points lead to deadlock!

```
wait(lock_a);
```

```
wait(lock_b);
```

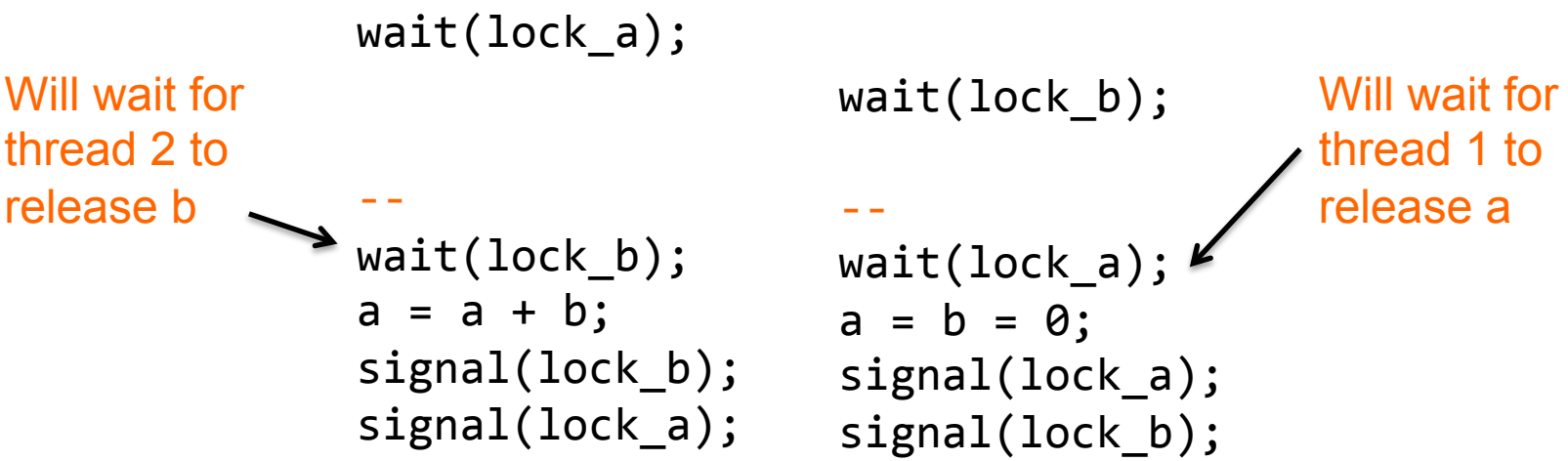
```
--
wait(lock_b);
a = a + b;
signal(lock_b);
signal(lock_a);
```

```
--
wait(lock_a);
a = b = 0;
signal(lock_a);
signal(lock_b);
```

Concurrency in RT Systems

When ever there's locking involved, bad coding may lead to a deadlock.

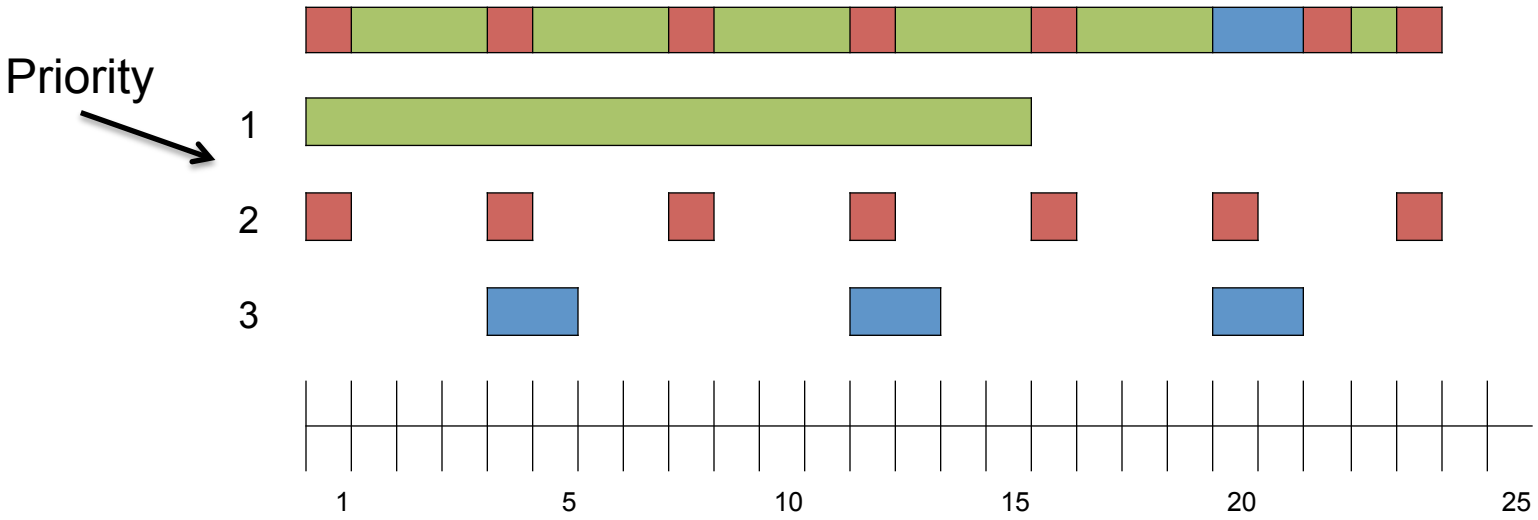
There are many different types of deadlock, but the simplest to understand is the ABBA deadlock.



Concurrency in RT Systems

Another unintended side effect of locking in a multithreaded environment is priority inversion.

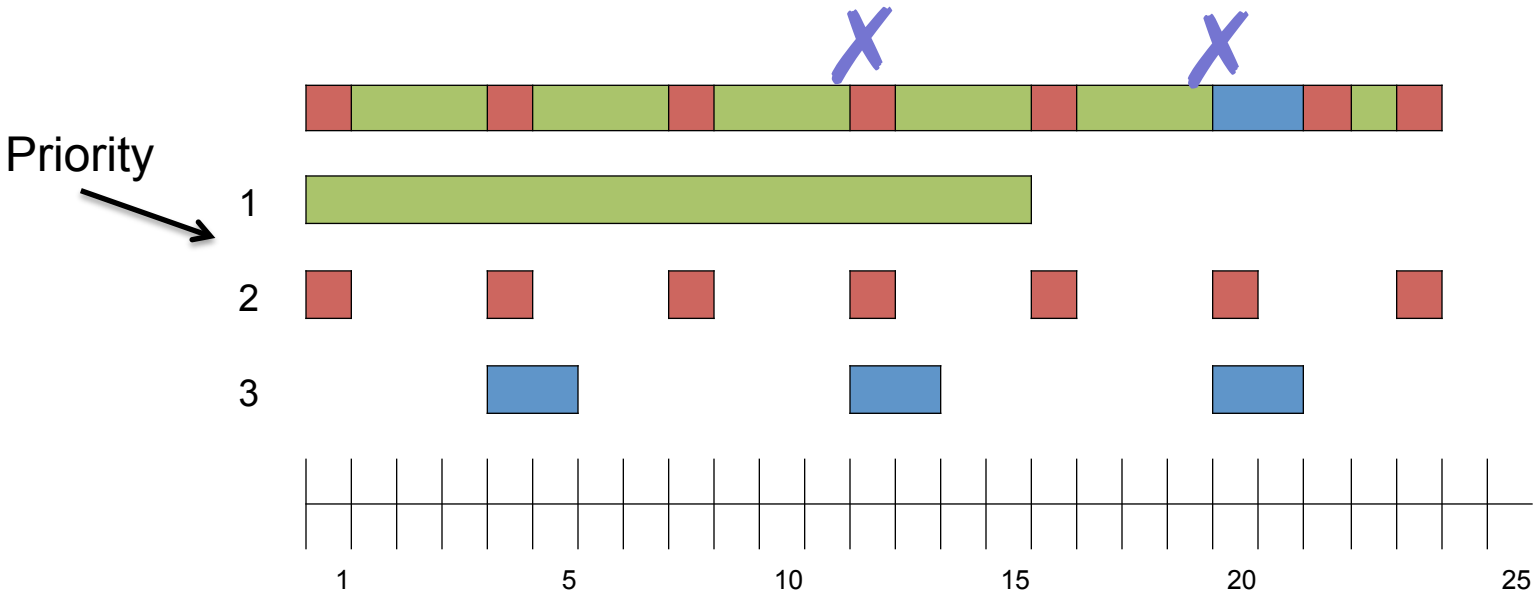
Assume that ‘green’ and ‘blue’ threads both wish to take the same lock.



Concurrency in RT Systems

Another unintended side effect of locking in a multithreaded environment is priority inversion.

Blue thread doesn't get to use the CPU, despite the fact that the lower priority Red task is getting plenty of CPU access. Red is blocking Blue; their priority is inverted.



Tutorial Question

Consider the following C code, written for an AVR microcontroller on the bare metal. The _delay_ms() macro works by simply executing a number of "no operation" instructions to kill the right amount of time. This number is computed at compile-time based on the processor's clock speed.

1. Will enabling interrupts have any effect on how long the delay will actually take to complete? Is the delay likely to be longer or shorter than intended?
2. Say the processor is connected to a motor and that an interrupt is triggered each time the motor completes a revolution. The motor has a top speed of 1000RPM and the interrupt, when triggered, takes 62uS to complete. There is no coupling between the waveform and the motor speed. What is the worst-case error in timing, as a percentage, of the output waveform?
3. Assume the motor is equally likely to be at any speed in its range at any time, what's the average delay actually going to be? What's the standard deviation going to be? What's the likelihood, in percent, that any given delay is going to be off by more than 50%?

Tutorial Question

```
while(1) {
    pin_high();
    _delay_ms(1);
    pin_low();
    _delay_ms(1);
}
```

Tutorial Question

Consider the following two threads. Is access to all variables safe? Can either thread deadlock? If so, give one example set of preemtion points that would trigger it.

```
Semaphore lock_chick, lock_fox, lock_egg;
Integer Fox, Chicken, Egg, Snake;
```

Thread 1

```
wait(lock_chick);
wait(lock_fox);
Fox += Chicken;
wait(lock_egg);
signal(lock_chick);
Egg -= Snake + Fox;
signal(lock_egg);
signal(lock_fox);
```

Thread 2

```
wait(lock_egg);
wait(lock_fox);
Fox = Egg;
signal(lock_fox);
wait(lock_chick);
Egg = Chicken;
signal(lock_chick);
signal(lock_egg);
```