Software Concepts for Embedded Systems

Software Development

- Software development is often both expensive and risky.
- The total expenses of software development and the significant risk of failure (surprisingly common in large projects) are not fully appreciated by many managers.
- Software is often impossible to specify completely at the start of a project. Software requirements are typically incomplete and often change as the project progresses.
- Software systems can be arbitrarily complex, and this complexity can easily overwhelm the understanding and capabilities of teams of even expert software designers.
- Turn-over rates are high in the software industry. Expertise is lost as designers leave. New designers must be trained and are (at first) more likely to introduce design errors.

Software Engineering

- "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software”   [IEEE Std-610.12-1990]
- To increase the productivity of the software designers, and improve the reliability of the resulting software systems:
  - Adopt proven software engineering methods.
  - Program in a well-designed *high-level language*. In the embedded systems industry, C is dominant.
  - Use assembly language only when efficiency and/or fast execution speed are essential.
  - *Avoid designing the entire system from scratch*.
  - Re-use and adapt existing known-good software.
  - Build a new software system on top of an existing reliable operating system or kernel (e.g., MicroC/OS).
  - Grow a working simple prototype into the final system.

Divide-and-Conquer Design Strategy

- A standard strategy for managing complex engineering design problems is to *partition the design into loosely interacting, simpler sub-systems or modules*.
- The simpler sub-systems can then be designed and verified separately. The resulting sub-system solutions can then be combined to solve the original design problem.
- In software systems, the sub-systems could include:
  - operating system or kernel (usually from a 3rd party)
  - the file system (often from a 3rd party)
  - messaging system (usually part of the O.S.)
  - exception and/or interrupt handling routines
  - network interface and other device drivers
  - library routines, functions, object classes, etc.
  - application tasks / processes / threads (custom)
  - webserver interface, custom forms, etc.

Aspects of Modular Design

- What makes a design have appropriate modularity?
- There are no hard rules. Here are some guidelines:
  - **Keep modules fairly short** and **easy to understand**: they should contain only one or two pages of code.
  - Modules should have **high cohesion**: each module is concerned with closely related functions and data.
  - Modules should enclose and **hide information details** from other modules that don't need those details.
  - Modules should be **loosely coupled**: there should only be relatively few & simple inter-module dependencies.
  - Only the **essential parameters** should be passed from one module to another.
  - **Global variables should not be used** to pass inform-ation between modules. A global variable, if one is required, should be updated by only one module.

Sources of Real-time Events

- Real-time events originate either externally or internally to the embedded system.
- ***Externally initiated events***, for example:
  - A signal has been received from elsewhere, such as a *user input* or a *communications message.*
  - A *sensor* has detected a change in the environment.
  - An *actuator* has completed a commanded action.
  - An *alarm condition* (e.g., power failure) has occurred.
- ***Internally initiated events***, for example:
  - A *timer-triggered interrupt* has occurred.
  - A *direct memory access data transfer* has finished.
  - A *software-triggered exception* has occurred.

General Kinds of Real-Time Specifications

- An embedded read-time system must act in a way that satisfies all the real-time specifications.
- Typical kinds of real-time specifications:
  - ***Maximum response time*** to different events
  - ***Maximum variability/jitter in the response time***
  - ***Accuracy of the repetition frequency***: e.g., an analog sensor signal may need to be sampled and converted at some specific frequency.
  - ***Maximum variability/jitter in repetition frequency***
  - ***Degradation behaviour*** when the workload exceeds system capacity and/or if the battery charge is nearly exhausted. E.g., can less important activities be safely deferred (possibly by skipping over code or disabling interrupts) so that important events will be handled?

The Important Role of Idle Time in Real-time Systems

- ***Idle time*** is CPU execution time when no real-time work is being performed by the software.
- Idle time can be used to perform low-priority work (e.g., background preemptive testing, measuring the amount of idle time by incrementing an idle instruction counter, memory defragmentation). When idle time is used for low-priority work it is sometimes called ***background time***.
- Sufficient idle time <u>must</u> be present in real-time systems:
  - ❖ For *externally initiated events*, idle time provides the necessary excess CPU capacity so that *worst-case bursts of event-handling workload can be handled* within the maximum response time and determinism (i.e., max. response time variability) specifications.
  - ❖ For *internally initiated events*, idle time provides flexibility so that the effects of *internal sources of timing variability* (e.g., variability in the execution time of compiled software, variability caused by cache memory and virtual memory, variability caused by DMA activity) *can be absorbed and effectively hidden*. The timing for internally initiated events should be determined by H/W timers.

Programming on Bare Metal

- For the simplest embedded systems, it may be desirable to design the software system as a ***standalone program*** that runs directly on the microcomputer hardware without the presence of a kernel or operating system. This is often called programming on "***bare metal***".
- The main alternative to a bare metal software design is one that runs on the microcomputer hardware along with generic kernel or operating system software.
- ***Main advantage of programming on bare metal***: Very efficient operation with the simplest possible software.
- ***Disadvantages***: The software will not leverage the features in available kernels or operating systems. It may be harder to add new functionality later. Leads to a less modular and less flexible S/W design. Harder to do divide-and-conquer.

Software Architectures for Embedded Systems

Single-threaded Architectures *(often bare-metal)*

- Foreground-background systems
- Sleep mode with interrupts
- One non-preemptive loop (no interrupts)
- One non-preemptive loop with interrupts
- Multiple non-preemptive loops with interrupts
- etc.

Multitasking Architectures *(using a kernel or operating system)*

- Round-robin time slicing
- Periodically scheduled state-driven code
- Cooperative multitasking
- Preemptive multitasking
- etc.

Foreground-Background Systems

- The simplest microcomputer-based systems might be programmed using one S/W thread executing in a single nonpreemptive loop, omitting a kernel or operating system.
  - The main loop can meet less challenging timing constraints through normal instruction execution.
    => ***background processing***

    Drawback: A long main loop might cause overly slow or overly variable response time to input events. You will not likely get hard real-time performance.
  - Interrupt service routines provide faster response for critical events (assuming interrupt hardware is present)
    => ***foreground processing***
- Foreground-Background systems are widely used in less demanding applications (e.g., appliances, simple games)

Sleep Mode with Interrupts

- It may be possible and desirable to keep the microcomputer in a ***low-power sleep state*** most of the time. Stretching the lifetime of a battery-based power supply is typically a key priority. Hard real-time performance is not required.
- The microcomputer could be ***woken up by external inter-rupt signals*** that exceed a specified interrupt priority level.
  - All of the "work" could be done by the interrupt service routines. There would be no background processing.
  - The MCF54415 provides a privileged STOP instruction that allows the software to load the SR with a new interrupt priority mask before putting the controller into one of three possible stop modes: *wait*, *doze* & *stop*.
  - Only sufficiently high interrupts will wake up the CPU.

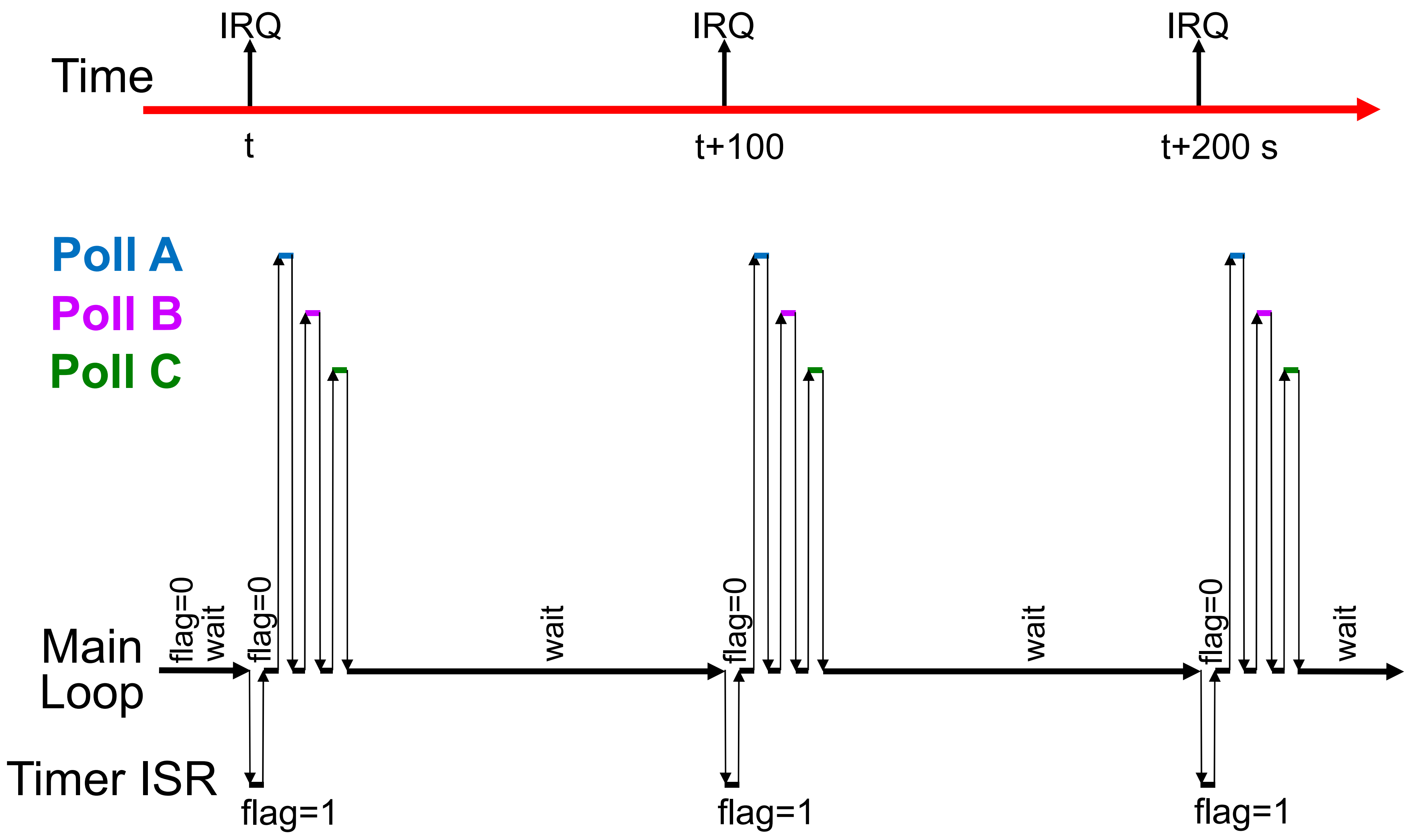
One Nonpreemptive Loop (1)

- Also called “***static nonpreemptive scheduling***”.
- A number of input sources or devices need to be polled. The required polling frequencies may be different.
- A ***single processing loop*** polls all of the inputs & devices in some fixed order. The loop iterates at a frequency that is at least as fast as the fastest required polling frequency.
- **The worst-case (longest possible) execution time of the software in the loop must be less than the loop period.**
- The looping frequency is best determined by a ***hardware timer***. Don’t rely on the instruction execution times to determine the looping frequency.
- A waiting sub-loop at the end of the main loop safely uses up excess time at the end of each iteration of the main loop.

One Nonpreemptive Loop (2)

```
initialization_steps();
static int flag = 0;
while (1) {
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}
    flag = 0;   /* cleared for next iteration */
    /* Start executing the nonpreemptive loop */
    poll_A();
    poll_B();
    poll_C();
}  /* end while */
```

One Nonpreemptive Loop (3)



One Nonpreemptive Loop (4)

- ***Advantages***:
  - Simplicity
  - Guaranteed minimum polling frequency
  - Guaranteed maximum response time
- ***Disadvantages***:
  - Relatively poor determinism and possibly slow response when handing external events.
  - Some devices may be polled faster than what is necessary and/or preferred.
  - The single loop has relatively poor cohesion. This can be counteracted by using modular polling subroutines for each polled input and device.

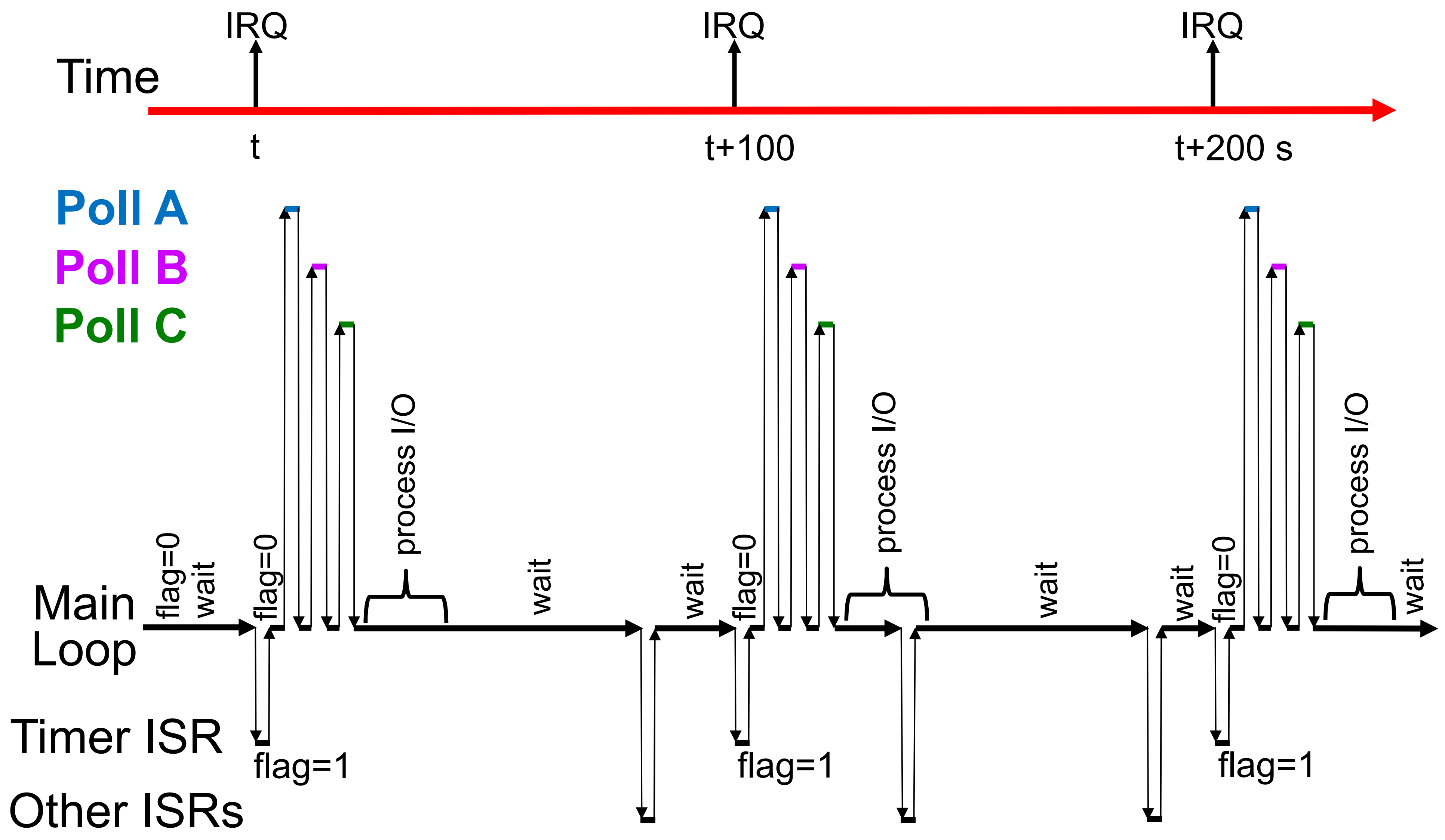
One Nonpreemptive Loop with Interrupts (1)

- Enhance a single nonpreemptive loop with a few interrupts to ensure that hardware-triggered events are serviced promptly and deterministically.
- Each ***interrupt service routine*** (ISR) should be ***short***. For example, an ISR might access a few hardware registers and transfer data to/from a buffer in memory. ***Detailed data processing should be done using software in the main loop, not the ISR.*** The ISR can enable data processing by signalling to a flag or semaphore that allows software outside the ISR to proceed later (within a brief delay).
- Keep the processor lightly loaded so that the worst-case execution of ISRs does not compromise the real-time performance of the main processing loop.

One Nonpreemptive Loop with Interrupts (2)

```
initialization_steps();
static int flag = 0;
while (1) {
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}
    flag = 0;
    /* Start executing the nonpreemptive loop */
    poll_A();
    poll_B();
    poll_C();
    process_data_inputs_from_other_ISRs();
    transfer_data_outputs_to_other_ISRs();
}  /* end while */
```

One Nonpreemptive Loop with Interrupts (3)



One Nonpreemptive Loop with Interrupts (4)

- ***Advantages***:
  - Simplicity
  - Guaranteed minimum polling frequency
  - Predictable maximum response time for both polled devices and external events.
- ***Disadvantages***:
  - Some devices may be polled faster than necessary, which is wasteful of the processor's time.
  - The modularity of the single processing loop is reduced if the other IRQs generate work in the loop.
  - Frequent execution of multiple ISRs can reduce the determinism provided by each ISR to external events.

Multiple Nonpreemptive Loops with Interrupts (1)

- Use ***two or more nonpreemptive loops*** that iterate at different, harmonic periods. Note: the multiple loops must be implemented using *one software thread*. Use interrupts to provide fast and deterministic response to external events.
- Executions of the multiple loops should be enforced using a ***single hardware timer*** to ensure timing accuracy.
- ***Keep the ISRs short***. Most data processing must be done outside the ISRs, in the one main software thread.
- ***Keep the processor lightly loaded*** so that even the worst-case execution of ISRs does not compromise the real-time performance of the multiple processing loops.
- Each polled input or device is assigned to the one loop that iterates just fast enough to meet its real-time constraints (frequency and/or maximum response time).

Harmonic Loop Periods

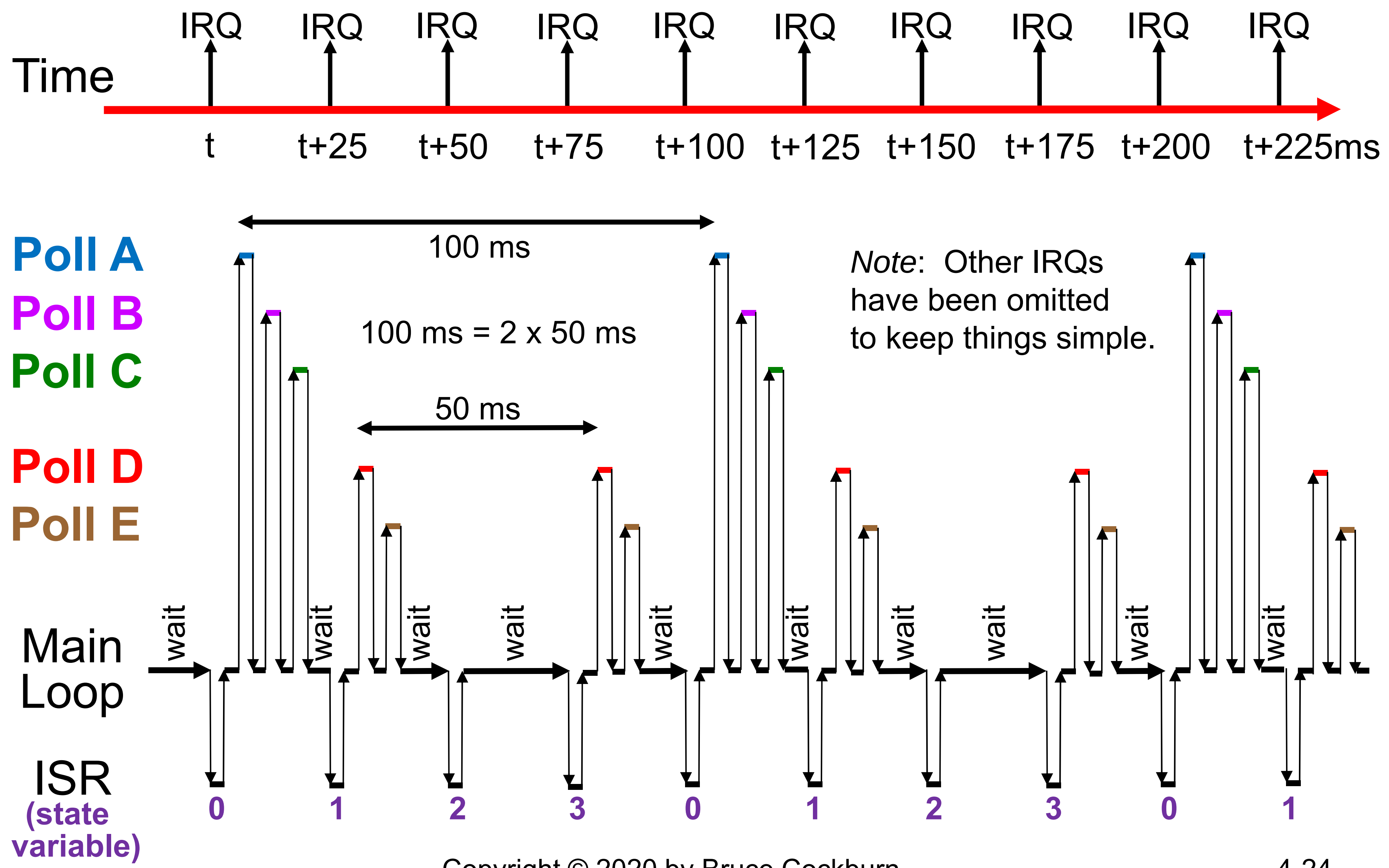
- When multiple nonpreemptive loops are executing in a real-time system, it becomes very difficult to predict the execution time behaviour and to guarantee that all the real-time response time constraints will be met.
- These problems are greatly reduced if the loop periods are ***harmonic***, that is, each loop period is a whole multiple of ***all*** shorter loop periods.
- The loop periods 10 ms, 50 ms and 200 ms are harmonic. Note that 50 = 5 x 10, 200 = 20 x 10, and 200 = 4 x 50.
- The loop periods 10 ms, 25 ms and 133 ms are not harmonic. (Note: 10 ms, 20 ms and 120 ms are harmonic).
- A single hardware timer should be responsible for producing the timing of all of the harmonic loop periods.

Example with Two Nonpreemptive Loops (1)

```
void function do_100_ms_loop() { poll_A(); poll_B(); poll_C(); }
void function do_50_ms_loop() { poll_D(); poll_E(); }

initialization_steps();
enum state_type { ST0=0, ST1, ST2, ST3 } state;
static int state = ST0;   /* 100 ms list executes first */
static int flag = 0;
while (1) {
    /* wait for the timer ISR to set flag to 1 */
    while ( !flag ) {}
    switch ( state ) {
        case ST0: do_100ms_loop();    state = ST1;  break;
        case ST1: do_50ms_loop();     state = ST2;  break;
        case ST2: /* no loop here */  state = ST3;  break;
        case ST3: do_50ms_loop();     state = ST0;  break;
    }  /* end switch */
    flag = 0;
} /* end while */
```

Example with Two Nonpreemptive Loops (2)



Meeting Real-time Constraints

Multiple Nonpreemptive Loops with Interrupts (cont'd)

- ***Advantages***:
  - Guaranteed minimum polling frequency for each input or device
  - Guaranteed maximum response time for polled inputs and devices
  - External events are handled promptly and deterministically using hardware-triggered interrupts.
- ***Disadvantages***:
  - Some devices may be polled faster than necessary, which is wasteful of the processor's time.
  - The one software thread has less cohesion because it now handles more than one loop. A state variable keeps track of the timer ISR calls and the active loop.

Tasks

- A ***task*** is an executing program together with the current context of the task at the present program instruction.

- The ***context*** of a task has several components:
  - the contents of all *CPU registers*
  - the contents of the task's *stack* in RAM
  - the contents of any other *allocated RAM locations*
  - the state of any other *allocated system resources*

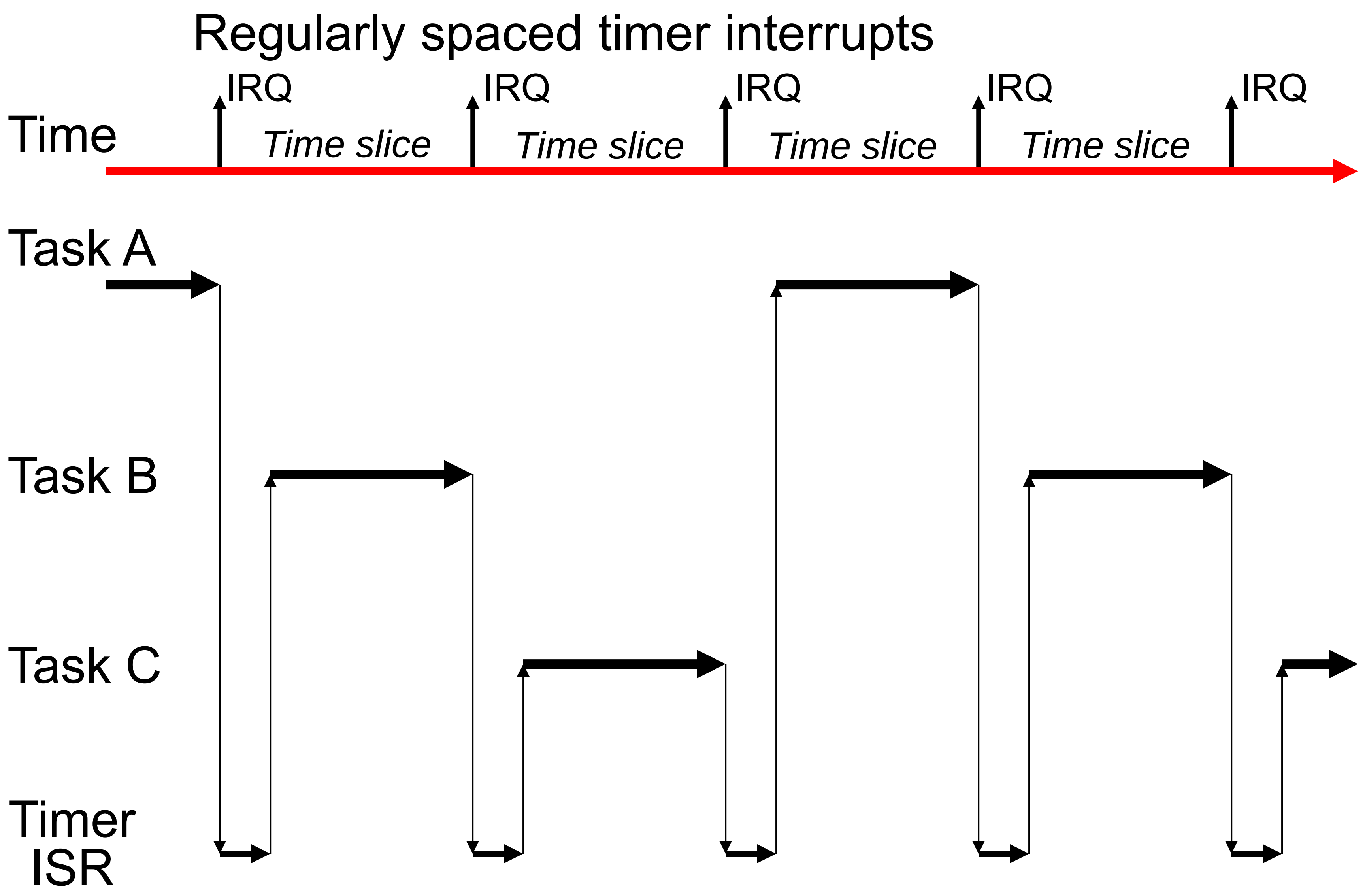
- A task can be stopped at one time and restarted later if the task context is saved in a special data structure, which is often called a ***task control block*** (TCB).

- The task control block will only have space to contain the CPU registers; the other parts of the context are left alone (but other tasks must be prevented from changing them).

Partitioning Software into Multiple Tasks

- An effective way of developing a software system is often to design it as a collection of interacting tasks.
- Each task is written as if it has exclusive access to the CPU.
- Each task provides *one coherent service* in the system, and should therefore have a *relatively simple structure*.
- The tasks interact with each other using synchronization primitives (e.g., semaphores), messages, shared data structures, etc. to produce the desired system behaviour.
- System software is provided for stopping and restarting tasks. Usually only one task can run at a time on one CPU. (*Note*: processors are now widely available that provide hardware-controlled “multithreading” or “hyperthreading” that allows multiple tasks to appear to run on one CPU core.)

Strictly Round-Robin Time Slicing



Context Switching

- In a round-robin multitasking system, the timer ISR is part of a body of system software called the ***kernel***.
- The timer Interrupt Service Routine must first decide if the currently executing task should be suspended.
- If not, then the ISR executes a "***return from exception***" to allow the same task to keep on running with another slice.
- If yes, then the ISR causes a "***context switch***" as follows:
  1. the contents of all CPU registers are loaded into the task's Task Control Block (TCB).
  2. A new task is selected from a queue of ready-to-run tasks (actually, a queue of TCBs).
  3. The CPU registers are loaded from the register values that were saved previously in the new task's TCB.
  4. The ISR executes a "return from exception" instruction, and the new task resumes executing where it left off.

Is Time Slicing a Real-time Kernel?

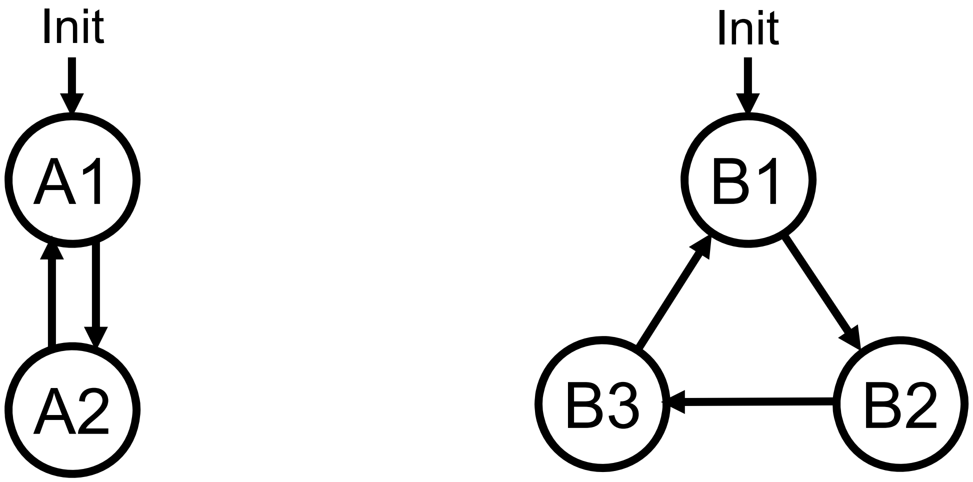
- The time slicing multitasking architecture effectively creates multiple virtual CPUs, with one such CPU for each task.
- Time slicing is a simple way of *sharing one CPU* for the execution of *multiple non-hard-real-time software tasks*.
- The different tasks can be allocated different proportions of the available time on the one actual CPU.   For example, some tasks can be allocated a greater fraction of the available equal-duration time slices per second.
- To provide some timing flexibility (at the cost of extra kernel complexity), tasks might have the ability to give up the CPU early before the end of their allocated slice.
- Real-time events (internal and external) can be handled by interrupt service routines in conjunction with tasks.

Periodically-Scheduled State-Driven Code

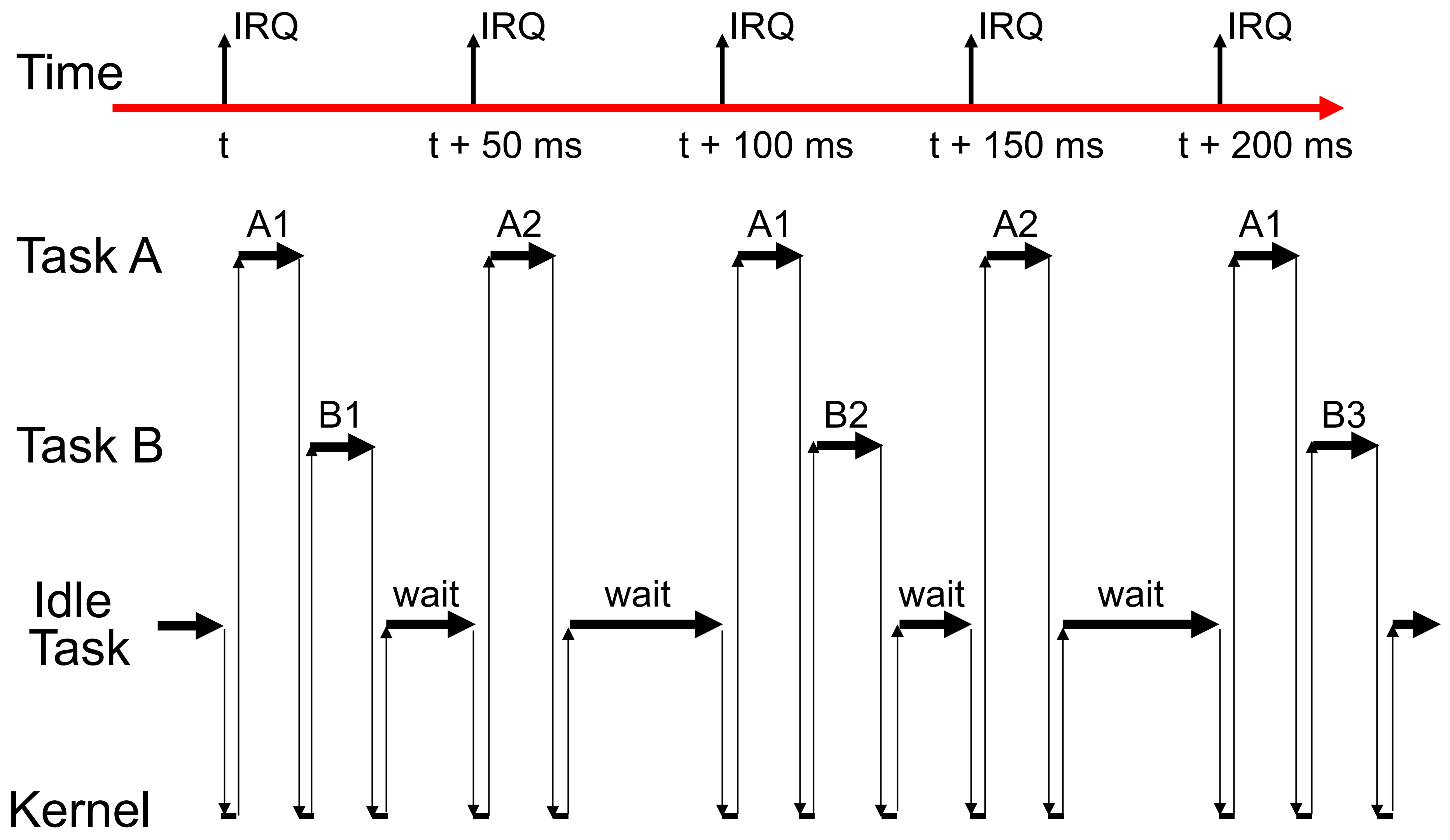
- In some applications, it may be possible to partition each task into a collection of interconnected *states*.
- Each state corresponds to a code segment that needs to be executed in that state (e.g., control and input/output actions).
- After a state code segment has been executed, the task gives up the CPU to the kernel. The delay until the execution of next state of a task can be fixed or changeable.
- Different tasks advance from state to state at predictable rates (e.g., some tasks at 10 ms intervals, other tasks at 100 ms intervals, others at 500 ms intervals, etc.). The periods are harmonic to make execution behaviour more predictable.
- When no task is running, non-state-driven tasks and/or a low priority ***idle task*** safely use up the remaining CPU time.

Example of State-Driven Code (1)

- Consider a system partitioned into two state-driven tasks.
- Task A is to be scheduled to run every 50 milliseconds.
- Task B is to be scheduled to run every 100 milliseconds.
- The idle task is to consist of a simple "busy wait" loop that executes “no operation” (NOP) instructions.
- The state transition graphs for Tasks A & B are as follows:



Example of State-Driven Code (2)



Cooperative Multitasking (1)

- In **cooperative multitasking**, once a task starts to run on the CPU, it can continue to execute on the CPU until it decides to allow another task to execute.
- The tasks must interact cooperatively to ensure that the system behaves correctly.
- Task priorities are not used. However, prioritized hardware interrupts may still be present.
- TinyOS is an example of a successful cooperative multitasking operating system.
- How to introduce some idle time? One could introduce an idle task that busy waits (executes “no operation” or NOP instructions in a loop) until a hardware timer signals the start of the next looping interval.

Cooperative Multitasking (2)

Advantages:

- Potentially very efficient. The number of context switches is minimized. One task can "hog" the CPU for as long as it needs.
- The operating system software is very simple.

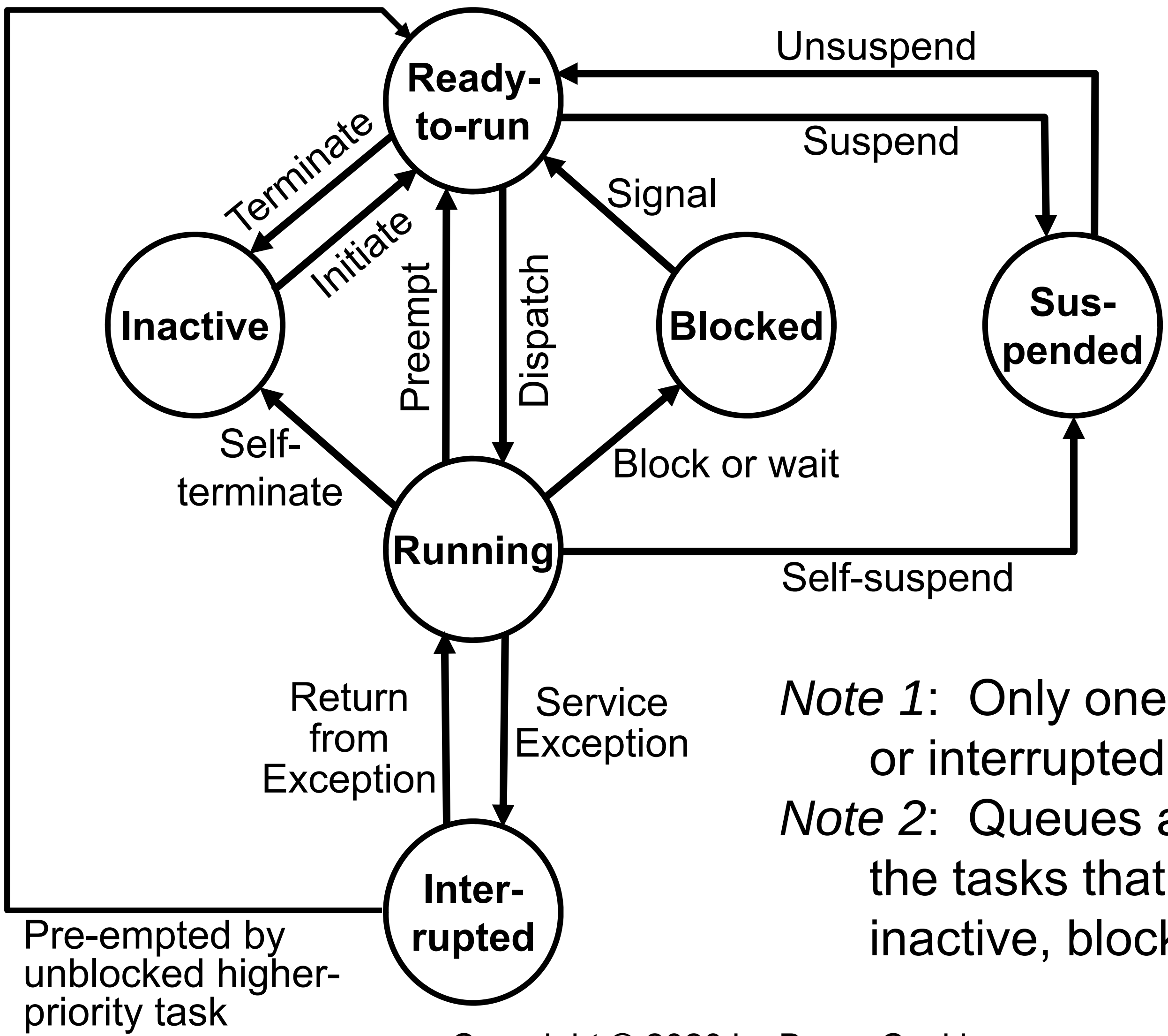
Disadvantages:

- The tasks have relatively poor modularity since the tasks must be designed to cooperate efficiently. One task can "hog" the CPU, preventing other tasks from running.
- Very hard to control or give guarantees on real-time response. ISRs can respond promptly, but the timing of subsequent task software is hard to predict.

Preemptive Multitasking (1)

- In ***preemptive multitasking***, each task is assigned a ***priority*** that determines which among the one or more competing, ready-to-run tasks should be allowed to run on the CPU.
- **The highest-priority ready-to-run task always gets the CPU.** The operating system enforces this rule.
- The priority of each task is assigned to it when the task is first created. Usually, task priorities are fixed. Often task priorities are unique: each priority has at most one task.
- In some cases the priority of a task might be possible to change, but this leads to a situation where it becomes much harder to predict the real-time behaviour of the system.
- Idle time (e.g., $\geq$ 30%) must be present to provide timing flexibility. The idle time is safely consumed using by a lowest-priority idle task that is always ready-to-run.

Process States in a Typical Preemptive Multitasking Environment



Note 1: Only one task is running or interrupted at any one time.

Note 2: Queues are used to record the tasks that are ready-to-run, inactive, blocked, and suspended.

Task States (cont'd)

Running: The one process that is actively executing instructions on the CPU (unless interrupted).

Interrupted: The one running process has been stopped temporarily to allow an interrupt service routine to run.

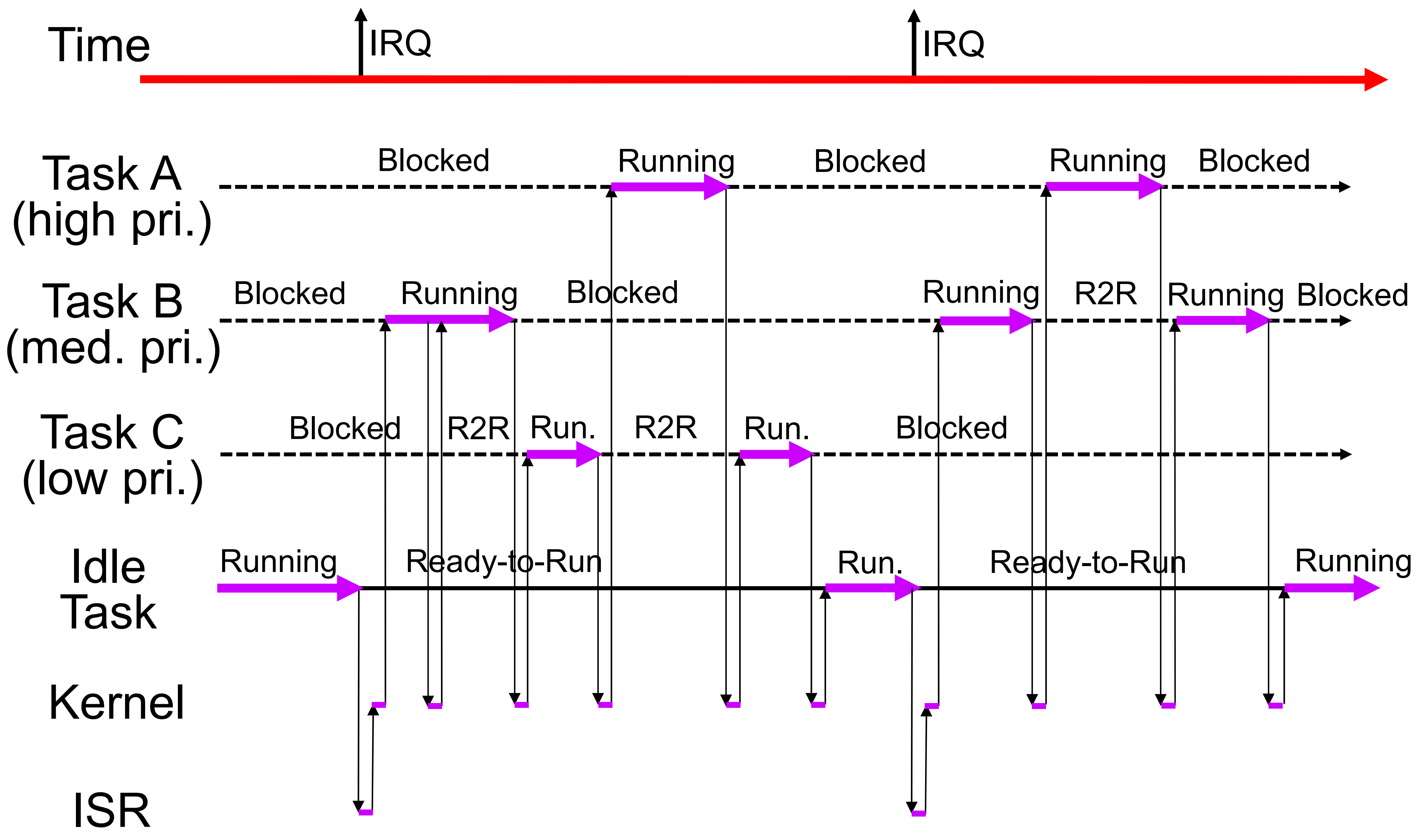
Ready-to-run: The process is available to start executing instructions as soon as the CPU becomes available.

Blocked: The process is waiting for a message to arrive, or is waiting for a flag or semaphore to become free.

Suspended: The process either suspended itself, or was suspended by another process.

Inactive: The process has been defined, but has either never been initiated, or was terminated later on.

Preemptive Multitasking with Fixed Priorities



Determinism: Predictability of Execution Timing

The highest priority task:
- Not directly affected by the activities of any other task
- Delayed for short times by interrupt service routines

Other relatively high priority tasks:
- Affected by the relatively few higher priority tasks
- Not affected by the execution of lower priority tasks
- Delayed for short times by interrupt service routines

Middle priority and lower priority tasks:
- Affected by the execution of all higher priority tasks
- Not affected by the execution of lower priority tasks
- Delayed for short times by interrupt service routines

The lowest priority task (the Idle task):
- Affected by the execution of all other tasks
- Delayed for short times by interrupt service routines

Setting the Task Priorities

- In pre-emptive multitasking, the task priorities can be ***fixed*** (or static) from the time of task creation, or they can be ***changeable*** (or dynamic) as the tasks execute.
- ***Fixed task priorities are generally preferred*** because this assumption makes system behaviour much more predictable.
- In preemptive multitasking with fixed priorities, the priorities should be assigned to tasks using the ***rate monotonic scheduling*** (RMS) rule: that is, tasks that must execute more frequently or that have tighter deadlines when they are enabled should be assigned higher priorities (*lower priority numbers* in MicroC/OS).
- A few ***safety-critical*** or other ***mission-critical*** tasks might be given higher priorities than less critical tasks.

Preemptive Multitasking

Advantages:

- Rate monotonic scheduling ensures that tasks that face tighter deadlines will have higher priority in getting time on the CPU. All real-time constraints will likely be met provided the system has sufficient idle time (e.g., $\geq$ 30%).
- Some of the context switching time of time slicing multi-tasking is avoided. The highest priority task can run efficiently, without interference, to its next blocking point.
- Inter-task interactions are minimized, which increases modularity and simplifies system design and analysis.

Disadvantages:

- Low-priority tasks, by being slow to post to synchronization primitives (e.g., semaphores) or by hogging shared resources, can block higher priority tasks from running.

Related “Multi-” Terminology

Some definitions from the website “techtarget.com”:

- **Multiprogramming** is “the interleaved execution of two or more programs by a processor”.

- **Multitasking** is “the management of programs and the system services they request as tasks that can be inter-leaved (while running on the same CPU)”

- **Multithreading** is “the management of multiple execution paths through the computer or of multiple users sharing the same copy of a program”.

- **Multiprocessing** is “the coordinated processing of programs by more than one computer processor”.

Tasks vs. Processes vs.Threads

- In larger computers systems (say those with full-scale operating systems, like Unix), it is more common to use the terms “process” and “thread” instead of “task”.

- Like a task, a ***process*** is associated with a program and a context. A process may also be associated with allocated memory regions, pointers, open files, sockets, etc.

- A ***thread*** (or light-weight process) is similar to a process, except that some resources may be shared with other active threads as part of a single process (e.g., files, memory space, memory pointers, input/output resources). Each thread has enough state information to be stopped and restarted independently (e.g., CPU register contents, its own stack space).

Operating Systems

- An ***operating system*** is software infrastructure that provides services to human users and other software modules, while ensuring efficient and reliable access to computer resources.
- Services to human users include: login shells, shell scripts, a graphical user interface (GUI), a file system, access to input/output devices, access to the Internet, etc.
- Services to software modules include: system calls for accessing computer resources, inter-process synchronization and communication, time slicing and priorities to share the CPU’s time among multiple active processes/tasks, etc.
- An operating system shields users and other software from much of the complexity of the computer system.

Real-Time Operating Systems

- A ***real-time operating system (RTOS)*** is one that needs to provide services in a timely manner so that the computer system can interact effectively with on-going events and processes that are occurring in the physical environment.

- ***Hard real-time constraints*** are constraints with relatively inflexible and demanding maximum response times. Failure to meet hard real-time constraints is unacceptable. Predictability in the response times is often very important. Specially-developed operating systems are required.

- ***Soft real-time constraints*** are constraints that require response times that can be met successfully by many conventional operating systems (e.g., Unix/Linux and Windows) provided the system load is not excessive.

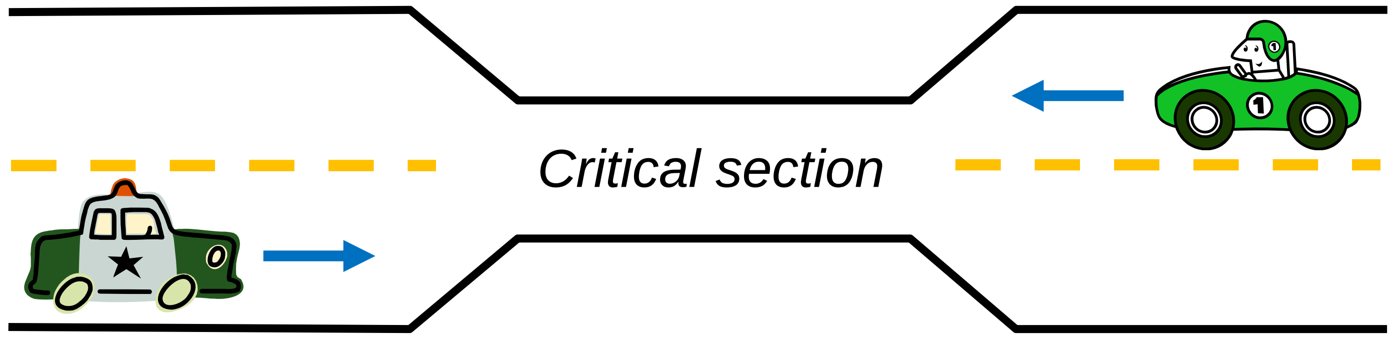
Real-Time Kernels

- A ***kernel*** provides the most basic services that would be expected in a multitasking operating system:
  - task/process creation, initiation, termination
  - task/process suspension and unsuspension
  - (possibly) timer-controlled time slicing
  - (possibly) process priorities and pre-emption rules
  - exception and interrupt handling mechanisms
  - basic inter-process communication services
  - mechanisms for protecting critical sections
  - inter-process synchronization features
- A small-scale system may have an operating system that is really just a kernel on its own.
- A ***real-time kernel*** is one that is intended to be used in systems that must meet hard real-time constraints.

Critical Sections

- It is common to have shared data structures that require the execution of several instructions in order to finish updates from one consistent state to a new consistent state.
  - shared data structures in the operating system (e.g., TCBs, synchronization primitives, I/O queues)
  - shared application data structures (e.g., two or more pointers need to be updated in a consistent way)
  - multiple registers in shared system hardware
  - etc.
- In a multitasking environment, there is the danger that shared data structures will be corrupted if a context switch occurs in the middle of a data structure update operation.
- A ***critical section*** is a section of code (e.g., data structure update) that must be executed to completion without being interrupted by task context switches or IRQs that could change/corrupt data that is updated in the critical section.

Example: A Single-Lane Bridge



- Only one car can be on the bridge at a time. The bridge is like a critical section, and the cars are like two tasks.
- However, the analogy is not quite correct because the two cars could be moving at exactly the same time, whereas in a multi-tasking system on a single-threaded CPU, two tasks are never executing at the same time.
- One task will always arrive first at a critical section.

Basic Strategies for Protecting Critical Sections

1. Disable all interrupts, allowing only one task to run
   - *Advantage*: Simple
   - *Disadvantage*: Too disruptive to the system?
2. Disable only those interrupts that could possibly threaten the critical section
   - *Advantages*: Simple and less disruptive than disabling all IRQs.
   - *Disadvantage*: Sometimes hard to determine those interrupts. The situation may change as new interrupts are introduced into a system that is modified and updated over time.
3. Disable multitasking, to allow only one task to run
   - *Advantage*: Simple
   - *Disadvantage*: All tasks are disturbed, which is disruptive to system. Critical sections must still be protected from interrupts.
4. Use semaphores to protect critical sections
   - *Advantage*: Minimizes the number of tasks that are affected.
   - *Disadvantage*: Critical sections within the semaphore functions must be protected from interrupts by disabling interrupts.

Disabling all Interrupts

(1) Mask out all interrupts (& disable multitasking) at the start of the critical section.
(2) Restore interrupts (& re-enable multitasking) at the end of the critical section.

Example using a ColdFire processor (supervisor mode only):

```
MOVE.W  SR,-(SSP)   /* save SR on supervisor stack */
ORI.L   #0x0700,SR  /* disable IRQs */
/* critical section appears here */
MOVE.W  (SSP)+,SR   /* restore SR from supervisor stack */
```

Advantages:
- Simple to implement. Use a macro in high-level code.
- Fast to execute on the CPU

Disadvantages:
- Can be executed only in Supervisor Mode (uses SR,SSP)
- All non-level-7 interrupts in the system are affected
- Unnecessarily disruptive to the system?

Using a Binary Flag to Protect a Critical Section

- A simple binary variable (often called a “flag”) can be used to enable or disable a task. Before proceeding into the critical section, the task must first consult the flag.
- If the flag is set to 1 (green light), the task can start executing the critical section.
- If the flag is cleared to 0 (red light), the task cannot enter the critical section. It must either do something else, or give up the CPU to another task and then try again later.
- The binary flag is usually implemented in a standard way as a *binary semaphore* (often called just a *semaphore*).
- A binary semaphore is a special case of a more general software object, called a *counting semaphore*.

Synchronization Using Counting Semaphores

A counting semaphore is a software object with two parts:

(1) an **integer count**, which is initialized to the number of available shared resources of a particular kind;

(2) a **queue** of tasks that are blocked and waiting for the semaphore count to exceed a value of 0.

- "Count > 0" implies at least one resource is available. The value of count gives the number of available resources.
- "Count = 0" implies that no resources are available, and that no other tasks are blocked on the semaphore.
- "Count < 0" implies that no resources are available. The count's absolute value gives the number of blocked tasks.

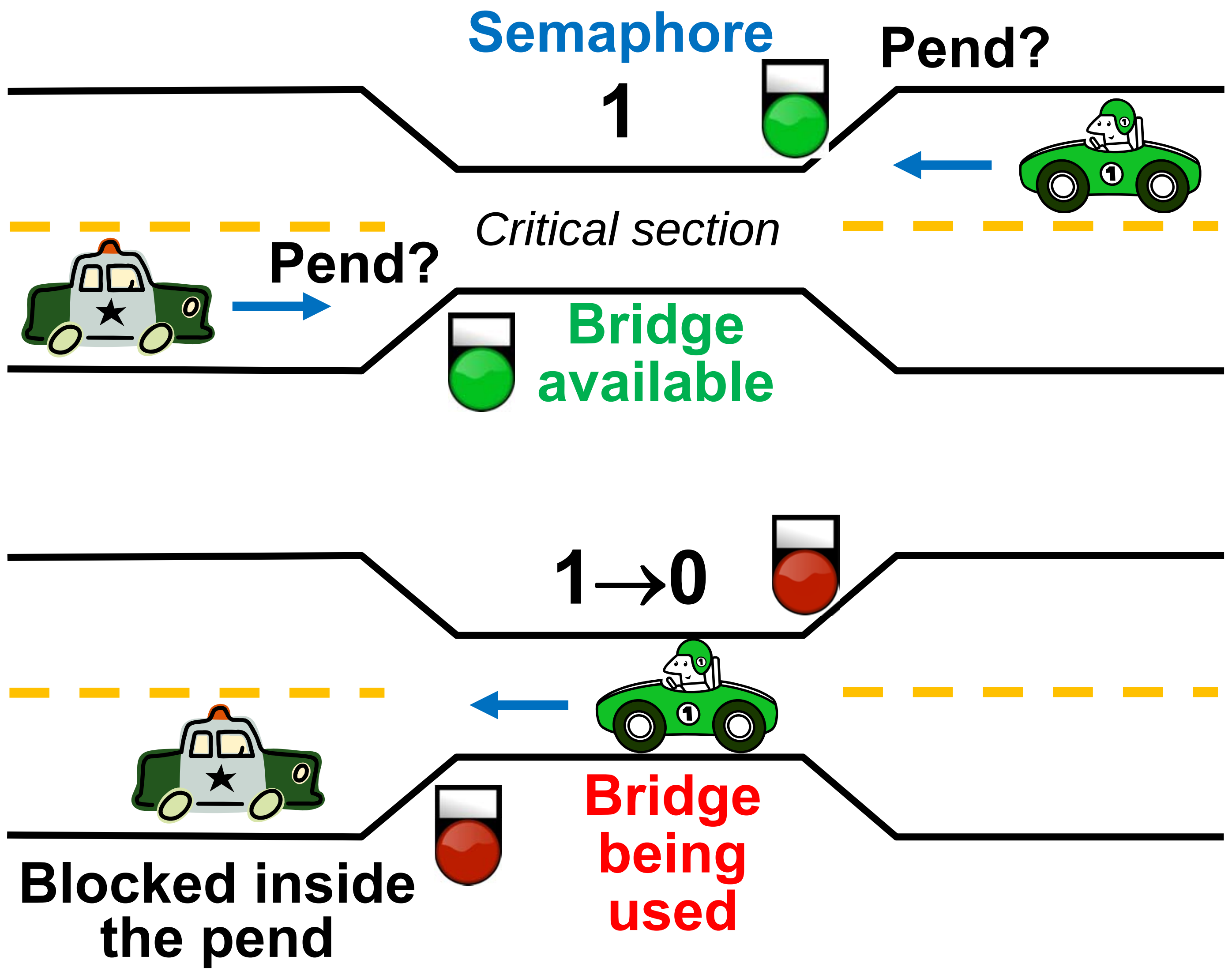
Updating Counting (and Binary) Semaphores

- **Pend** operation: A task needs exclusive use of a resource.
  - The count value is decremented by 1.
  - If the new count ≥ 0, then the task is allocated one of the available resources from the shared pool.
  - If the new count < 0, then the task is switched off the CPU and is added to a queue of blocked tasks. A context switch occurs to a new running task.
- **Post** operation: A task has just finished using a resource.
  - The count value is incremented by 1.
  - If no task is blocked on the semaphore, simply carry on.
  - If tasks are blocked on the semaphore, then the highest priority blocked task is moved to the ready-to-run queue. A task context switch might occur as a result.
- *Note*: Both Pend and Post are critical sections that must be protected from corruption by ISRs. Lock out IRQs!

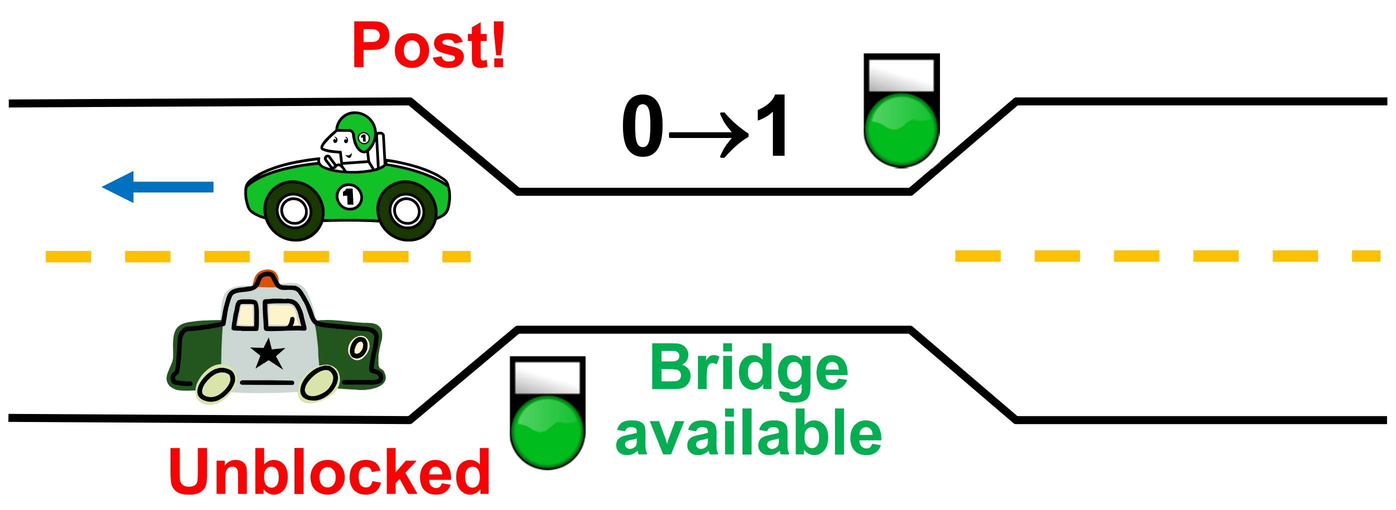
Binary Semaphores

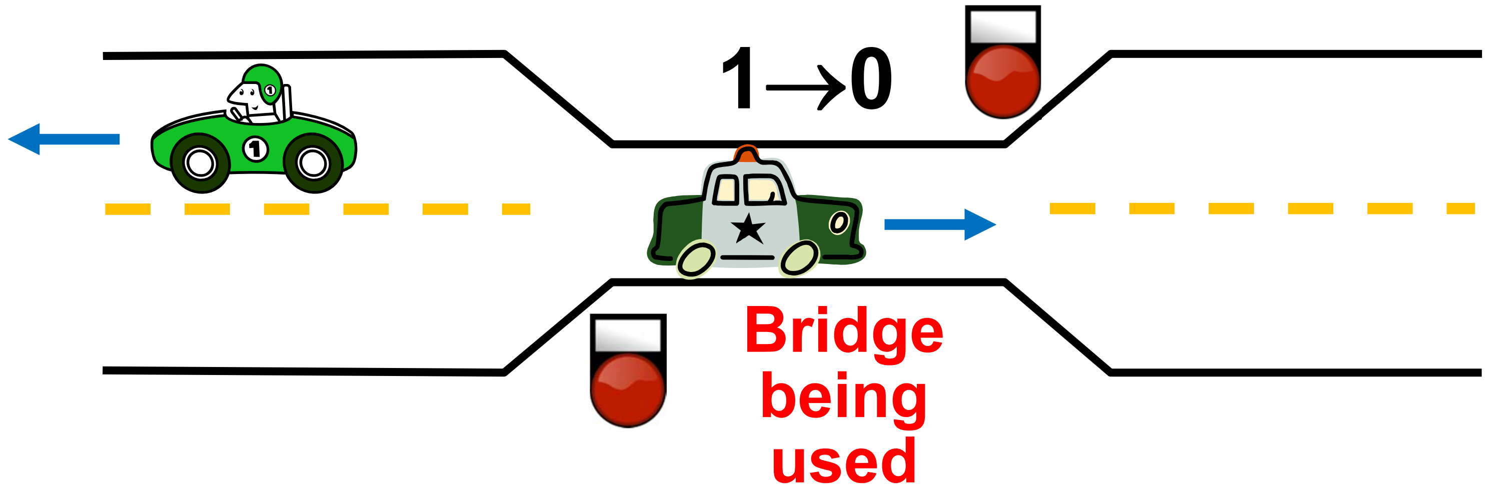
- A binary semaphore is a counting semaphore in which the count value is limited to 0 (**red light**) and 1 (**green light**).
- A binary semaphore can be used as a flag variable to enable the execution of a task at some point in the code.
- If the semaphore count is 0, then the task is blocked from executing (temporarily) when it does a "pend" operation on the semaphore. The kernel automatically schedules the next ready-to-run task, to allow the CPU to keep executing.
- If the semaphore count is 1, then the semaphore is decremented automatically to 0 by the "pend" function, and the task is allowed to enter the critical section.
- The semaphore count is incremented from 0 to 1 by a "post" operation to the semaphore, called by the task when it exits the critical section. Note: ISRs can also do a post.

Back to the Single-Lane Bridge (1)

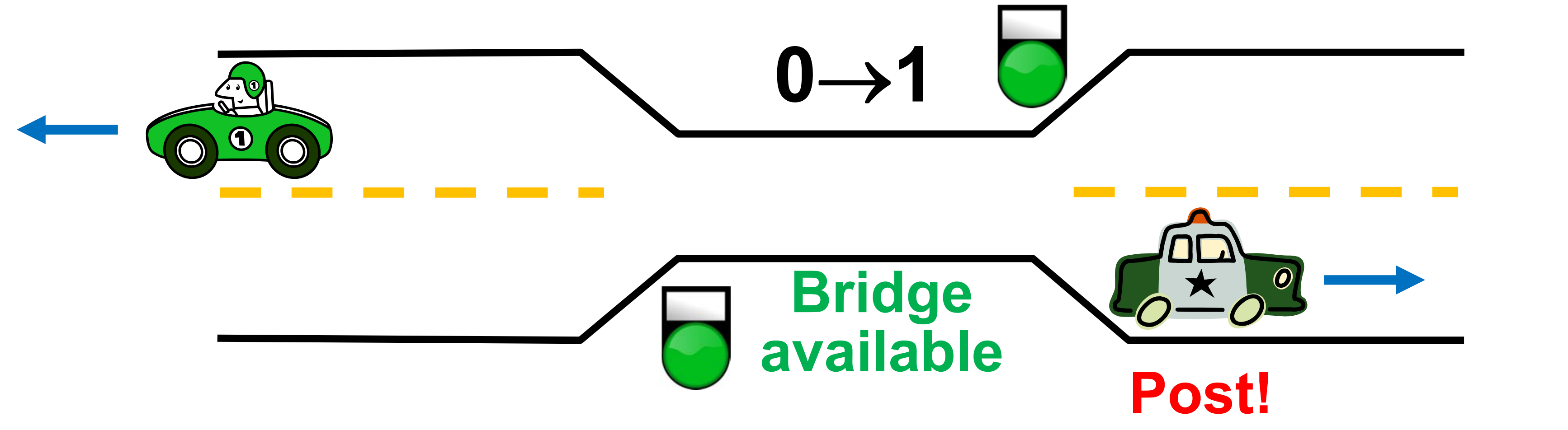


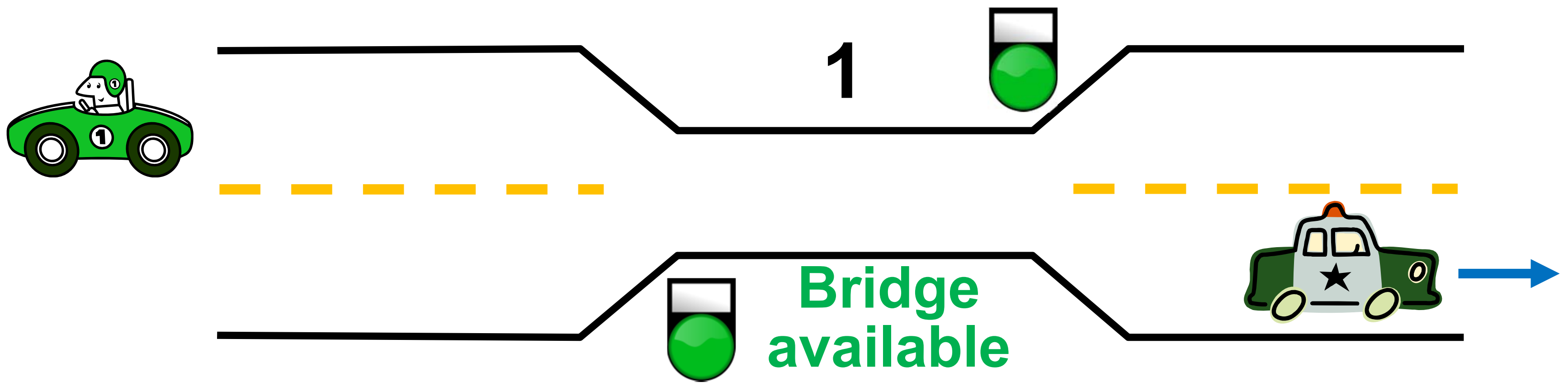
Back to the Single-Lane Bridge (2)





Back to the Single-Lane Bridge (3)





Binary vs. Counting Semaphores (1)

- A **binary semaphore** is a **flag** that is used to synchronize the execution of two pieces of code (e.g., a task and an ISR).
  - The semaphore would typically be initialized to 0 to block a task from proceeding before some condition (e.g., some data has arrived to be processed) has been met.
  - The semaphore is set (posted or signalled) to 1 to unblock the task after the condition for execution is met.
  - For example, a message handling task might be blocked (when it pends on the semaphore) up until the time when all of the bytes in a new message have been fully transferred from an input hardware interface into a data structure by an interrupt service routine (which posts the semaphore when the data transfer is completed).

Binary vs. Counting Semaphores (2)

- A **counting semaphore** is typically used to keep track of the number of available resources in a finite pool of resources.
  - The semaphore count is initialized to the size of the pool. All resources are initially available.
  - Each time that a resource is assigned and becomes busy, the count is decremented by 1 (pend); each time a resource is returned, the count is incremented by 1 (post)
- A counting semaphore can be used to implement a **binary semaphore**. The post and pend routines are the same, but the initial value of the count in a binary semaphore will be restricted to 0 (blocked) or 1 (unblocked).

Synchronization Using Message Passing

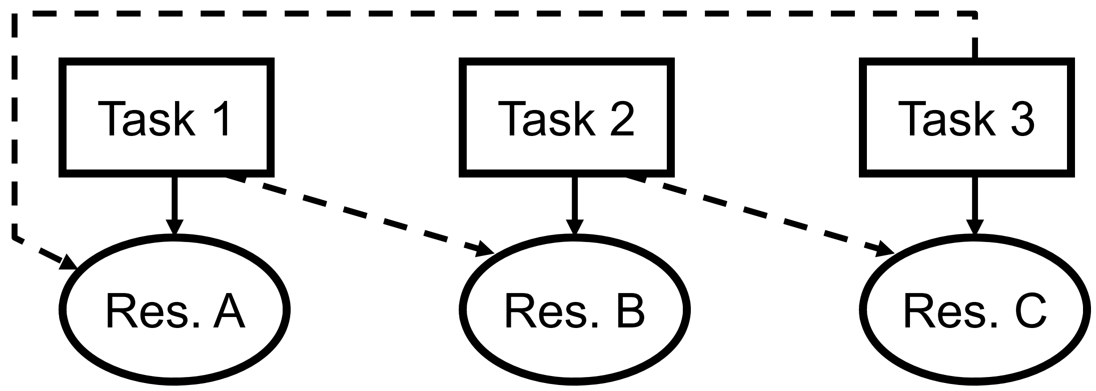
- Tasks (e.g. one *producer* task and one *consumer* task) can synchronize themselves to exchange data safely.
- Messages can be exchanged as follows:

(1) The producer task ***sends*** a message to the consumer task once newly produced data becomes available.

(2) The consumer task performs a ***block-and-receive***. If the message is immediately available, then the consumer task can process the new data (which may be enclosed or pointed to in the message). If the message has not yet been received, then the consumer task is switched off the CPU and is added to a queue of blocked tasks. When the next message arrives for the blocked consumer task, the consumer task is moved to the ready-to-run queue.

Deadlock

- Multitasking systems can be vulnerable to a condition called "deadlock", where two or more tasks become permanently blocked from running because they are waiting for resources, which are held by other tasks, to become available.
- Coffman's four conditions for deadlock to be possible:
  1. Resources can be assigned exclusively to tasks.
  2. One task can request two or more resources.
  3. A task cannot be forced to release a resource.
  4. Two or more tasks form a circular chain where each task is waiting for a resource that is held by the next task in the chain.



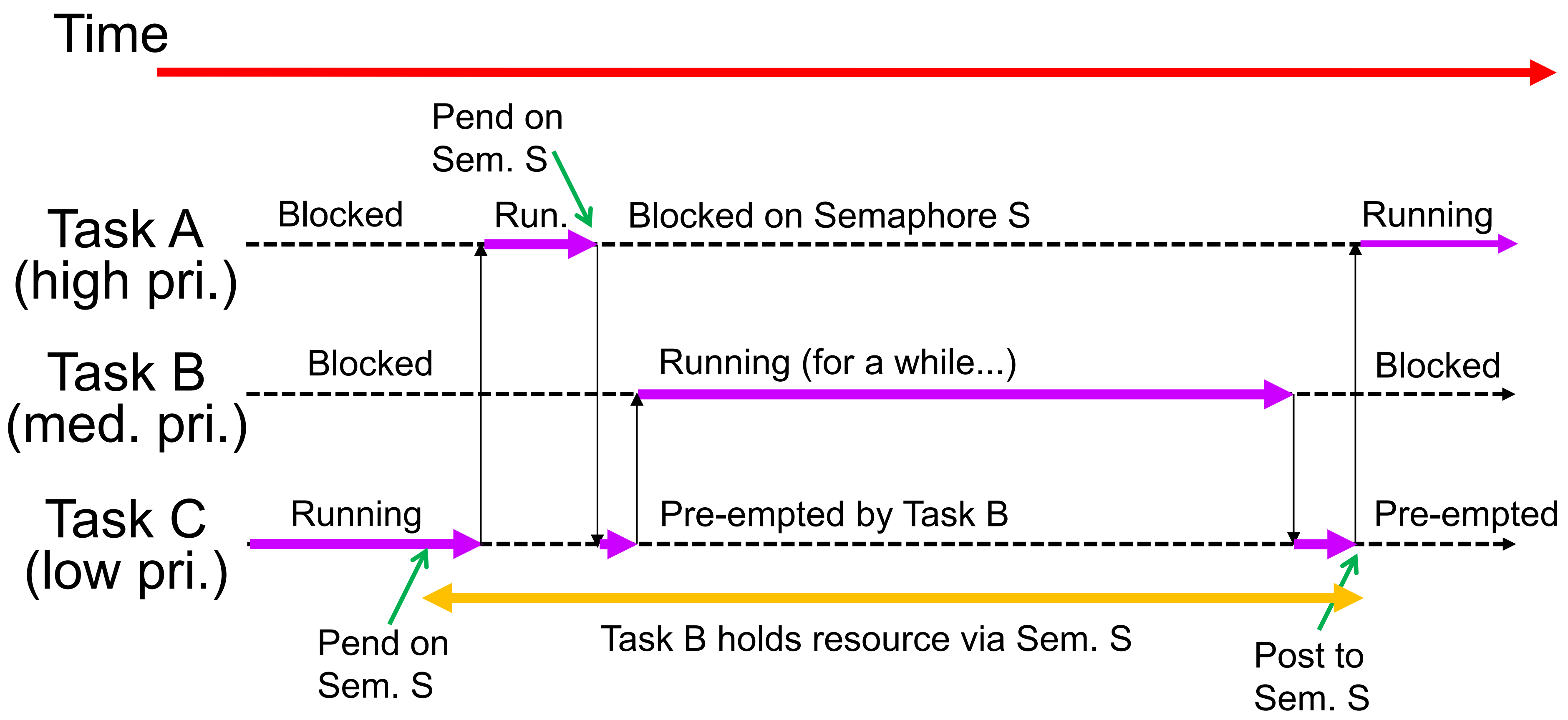
Watchdog Timers

- A watchdog timer provides a mechanism that allows an embedded system to, in many cases, recover from a catastrophic system failure, such as a memory leak, multitasking deadlock, or radiation-induced flipped bits.
- A ***watchdog timer*** is a hardware timer that counts down from some starting value at some frequency. If the count ever reaches zero, a hardware reset signal is activated which reboots the system. Rebooting will clear up many errors.
- The software system includes a ***watchdog timer service routine*** that is scheduled to run sufficiently frequently (e.g., a low priority task at a slow repetition rate). This routine re-loads the watchdog timer with the starting value each time that it executes. In a healthy embedded system, the watchdog timer value will never be allowed to reach zero. The H/W reset will only happen after serious system failure.

Priority Inversion (1)

- It is possible for a low-priority task to begin holding exclusive control of a resource (e.g., using a semaphore), and then later on a higher-priority task may start executing and then attempt (but fail) to gain access to the same resource that is currently controlled by the low-priority task. Execution may then return to the low-priority task when the high-priority task is blocked.
- This situation is called *priority inversion* because the low-priority task has blocked the execution of a higher priority task.
- The situation can be made even worse if a middle-priority task pre-empts the low-priority task, thereby blocking both the low-priority task and the high-priority task for some possibly long period of time.

Priority Inversion (2)



Note: Task A is prevented from executing by Task C (which holds already controls access to a resource using Semaphore S). Even if Task C has a short critical section, Task B may pre-empt it Task C and add further delay.

Solutions to the Priority Inversion Problem

- It is complicated to find safe ways of forcing a low-priority task to give up exclusive access to a resource in a critical section. The task may already have made partial changes.
- A common strategy for solving the priority inversion problem is to *temporarily raise the priority of the low-priority task to an even higher priority than that of any high-priority task that gets blocked on the semaphore* that protects the critical section. This allows the low-priority task to finish using the critical section quickly so that the high-priority task will not be delayed too long. Medium-priority tasks cannot interfere.
- MicroC/OS-II provides *mutual exclusion semaphores (mutexes)* which are binary semaphores that allow a high priority level to be reserved for automatic use by a low-priority task that happens to block a higher-priority task.

Disadvantages of Multitasking

- Time is wasted performing the context switches between running tasks.
- Memory space is required to store the kernel code.
- Critical sections and shared resources must be protected. Synchronization mechanisms must be provided.
- New problems, such as deadlock and priority inversion, can occur when multiple tasks require exclusive control of multiple shared resources. Precautions must be taken to avoid / detect such problems.
- Debugging multitasking software can be tricky because it may be difficult to understand and/or recreate the complex interactions between the tasks that led to a problem.

Figures of Merit for Real-Time Kernels

The properties and quality of a real-time kernel are measured using many different *figures of merit* (i.e. desirable properties):

Features: A kernel that has more features should make it easier and cheaper to implement the new system because the amount of required new software will be minimized.

Code size and Scalability: Many microcomputer systems provide a very limited amount of memory, hence a more compact kernel and/or a more customizable kernel is better.

Real-time response: The maximum amount of time that will elapse between the instant when an interrupt occurs and the time that the corresponding interrupt service routine starts executing. Faster response is better.

Figures of Merit (cont’d)

Time for a context switch: The time for one task to be suspended, and for execution to begin in another task or interrupt service routine. Faster is better.

Determinism: The predictability of the time between when the system receives an interrupt or receives a command and the time when external actions and/or internal updates actually begin. Greater determinism/predictability is better.

Certification: Has the quality of the kernel been certified by a reputable organization according to some recognized standard? Certification is better.

Portability: The ease with which the kernel, and the over-lying software, can be modified so as to function properly on a different microcomputer.