Real Time Operating System Support for Concurrency

Colin Perkins

http://csperkins.org/teaching/2003-2004/rtes4/lecture13.pdf

UNIVERSITY
of
GLASGOW

Lecture Outline

- Resources and Resource access control
  - Synchronisation and Locking
  - Implementing priority inheritance
  - Simpler priority ceiling protocols
  - Priority ceiling protocols for dynamic priority systems
- Messages, Signals and Events
  - Message queues and priority
  - Signals
- Clocks and Timers
  - Interval and watchdog timers
  - Sources of inaccuracy

Resources and Resource Access Control

- A complete system comprises both **tasks** and **resources**
  - System has ρ resources named R_1, R_2, …, R_ρ
  - Each resource R_k comprises ν_k units
- Tasks compete for the use of resources
  - Resources are used in a mutually exclusive manner
  - Assume a lock-based concurrency control mechanism
  - To use n_k units of resource R_k a task executes a lock to request them $L(R_k, n_k)$
    - If the lock fails, the requesting task is blocked until the resource is unlocked
  - When the task has finished with the resources, it unlocks them $U(R_k, n_k)$; Resources are released in LIFO order
  - The segment of a task between $L(R_k, n_k)$ and $U(R_k, n_k)$ is a **critical section**
- Tasks **contend** for a resource if one requests a resource another has been granted
  - A **resource access control** protocol arbitrates between tasks

Resources and Resource Access Control

- What types of resource are available?
- How do you lock resources?
- How is the resource access control protocol implemented?
  - A priority inheritance protocol
  - A priority ceiling protocol
  - Others?

Resource Types and Locking

- Program objects and data structures
- Files
- Devices
- Network interfaces
- System stack space

Access arbitrated by the operating system

Need to be locked by applications to ensure exclusive access

Semaphores

Mutexes

Condition Variables

Provided by POSIX and/or by real-time operating systems

POSIX Semaphores

```
int sem_init(sem_t *semloc, int inter_process, unsigned init_val);
int sem_destroy(sem_t *semloc);
int sem_wait(sem_t *semaphore);
int sem_trywait(sem_t *semaphore);
int sem_post(sem_t *semaphore);
```

- Memory based version of POSIX semaphores, lets you embed a semaphore within an object:

```
struct my_object {
    sem_t   lock;
    char   *data;        // For example…
    int     data_len;
}

struct my_object *m = malloc(sizeof(my_object));
sem_init(&m->lock, 1, 1);
…
```

- Generic semaphore; no special real-time features

POSIX Mutexes

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_mutexattr_setprotocol(pthread_mutex_attr_t *attr, int  protocol);
int pthread_mutexattr_getprotocol(pthread_mutex_attr_t *attr, int *protocol);

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- As with semaphores, a mutex is embedded in an object at a location of the programmers choosing
- Lets you lock access to that object/resource
  - Need to manually lock and unlock the resource
  - Compare to Java synchronised methods, classes

POSIX Mutexes and Resource Access Control

- A useful feature of POSIX threads is the ability to specify a resource access protocol for a mutex
- Use `pthread_mutexattr_setprotocol()` during mutex creation
  - **`PTHREAD_PRIO_INHERIT`**
    - The priority inheritance protocol applies when locking this mutex
  - **`PTHREAD_PRIO_PROTECT`**
    - The priority ceiling protocol applies when locking this mutex
  - **`PTHREAD_PRIO_NONE`**
    - Priority remains unchanged when locking this mutex
- Useful in conjunction with real-time scheduling extensions
  - Allow implementation of fixed priority scheduling with resource access control protocols
  - Control priority inversion
- If the priority ceiling protocol is used, can adjust the ceiling to match changes in thread priority:
  - **`pthread_mutexattr_getprioceiling(…)`**
  - **`pthread_mutexattr_setprioceiling(…)`**

POSIX Condition Variables

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex
                           struct timespec *wait_time);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Combine a condition variable with a mutex to wait for a condition to be satisfied:

```
lock associated mutex
while (condition not satisfied) {
    wait on condition variable
}
do work
unlock associated mutex
```

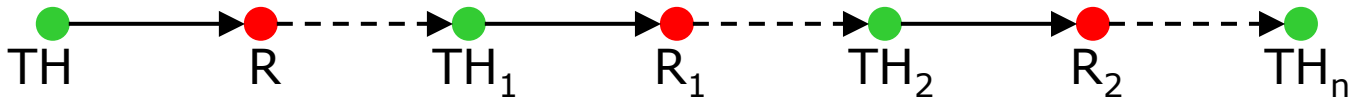
Implementing Priority Inheritance

- How to implement priority inheritance for mutexes within an operating system?
- Assume that the operating system provides two functions:
  - inherit_pr(*TH*)
    - Called when a thread is denied a resource, *R*, and becomes blocked
    - Causes all threads directly or indirectly blocking *TH* to inherit *TH*'s priority, increasing their priority to π_{TH}
      - They inherit *TH*'s priority through resource *R*
    - Assume: current priority of all threads blocking *TH* is lower than *TH*'s priority
      - True if there is one processor, deadlock cannot occurs, and a thread holding a resource never yields the processor
  - restore_pr(*R*, *TH*)
    - Called when a thread, *TH*, releases a resource, *R*.
    - The priority of *TH* is restored to π_r

  These are used by a resource manager within the system to raise and restore thread priorities as resources are acquired and released

inherit_pr()

- The inherit_pr() function looks up the thread TH_1 that is holding R
- That may be blocked on another resource, R_1
- Follow the blocking chain back to the thread TH_n, that is ready to run at a lower priority than TH



Example: Blocking chain from TH to TH_n

- Requires:
  - Each resource has a pointer to the thread holding that resource (the **owner** pointer)
  - In the TCB of each blocked thread, there is a pointer to the resource for which the thread waits (the **wait-for** pointer)
  - The scheduler is locked when following the blocking chain
- For every thread found, change it's priority to π_{TH}

restore_pr()

- A thread may hold multiple resources at the same time
  - As a result, it may inherit a number of different priorities as other threads block and contend for these resources
- Need historical information on how a thread inherited its current priority to ensure it is restored to the correct priority when releasing a resource
  - Maintained in the TCB of each thread as a linked list of records, the Inheritance Log, one for each resource the thread holds and through which it has inherited some priority
  - When restore_pr*(R, TH)* called it searches the inheritance log of *TH* for the record of *R*. If it exists, compute the new thread priority using the inheritance log and delete the record of *R.*

Implementing Priority Inheritance

- Should be clear that implementing priority inheritance and restoration is a heavy-weight operation
  - Many operations on tasks and resources
  - Basic priority ceiling protocol has similar issues
    - Needs to look at all the resources
- Would like a less complex alternative

The Stack as a Resource

- One resource we have not considered is the memory used by the run-time stack
- In the usual case, each tasks has it's own stack
- This is wasteful, since memory must be reserved for the maximum stack depth needed, and can fragment memory
- In systems that have limited memory, would like all tasks to share a common stack area
  - Claims of 90% storage saving have been made
- Stack space allocated in a last-in-first-out manner
  - When task *J* executes its stack space is at the top of the stack
  - If pre-empted by another job, *K*, the pre-empting job will have space in the stack above *J*
  - Clearly *J* can only resume after all tasks holding space in the stack above its space complete, free their stack space, and leave *J*'s space on the top of the stack again
- Restricts feasible schedules; tasks have ordering constraints

The Stack Based Priority Ceiling Protocol

- Leads to a modified version of the priority ceiling protocol: the **stack-based priority ceiling protocol**
- Use terms:
  - $\Pi(t)$ current priority ceiling
  - Ω non-existing lowest priority level
- Defining rules:
  - **Ceiling:** When all resources are free, $\Pi(t) = \Omega$; $\Pi(t)$ updated each time a resource is allocated or freed
  - **Scheduling:** After a job is released, it's blocked from starting execution until it's assigned priority is higher than $\Pi(t)$; non-blocked jobs are scheduled in a pre-emptive priority manner; tasks never self-yield
  - **Allocation:** Whenever a job requests a resource, it is allocated the resource
- The allocation rule looks greedy, but scheduling rule is not
- Simpler allocation and scheduling rules, compared to basic priority ceiling protocol

The Stack Based Priority Ceiling Protocol

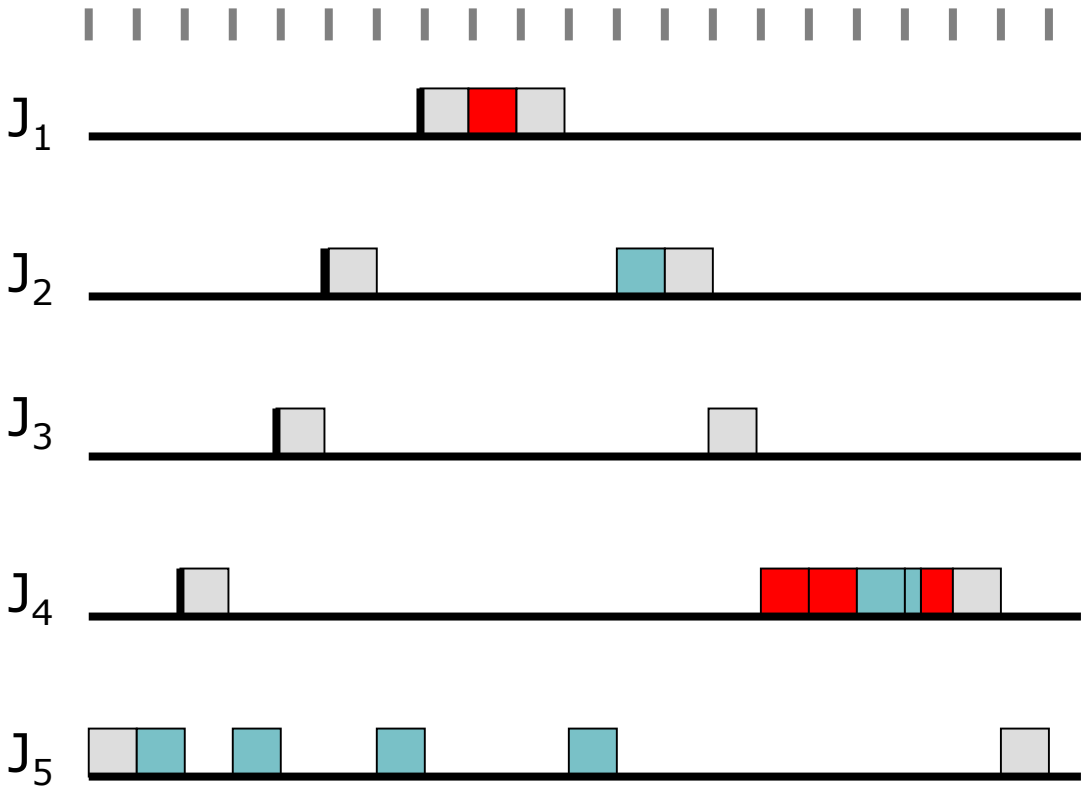
- When a job starts to run, all the resource it will ever need are free (otherwise the ceiling would be ≥ priority)
  - No job is ever blocked once its execution has begun
  - Implies low context switch overhead
- When a job is pre-empted, all the resources the pre-empting job will require are free, ensuring it will run to completion
  - Deadlock can never occur
- Longest blocking time provably not worse than the basic priority ceiling protocol

Comparison: Basic and Stack Based PCP

Example task set with resources (from lecture 10):

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[Red; 1]
J_2	5	3	2	[Blue; 1]
J_3	4	2	3	
J_4	2	6	4	[Red; 4 [Blue; 1.5]]
J_5	0	6	5	[Blue; 4]

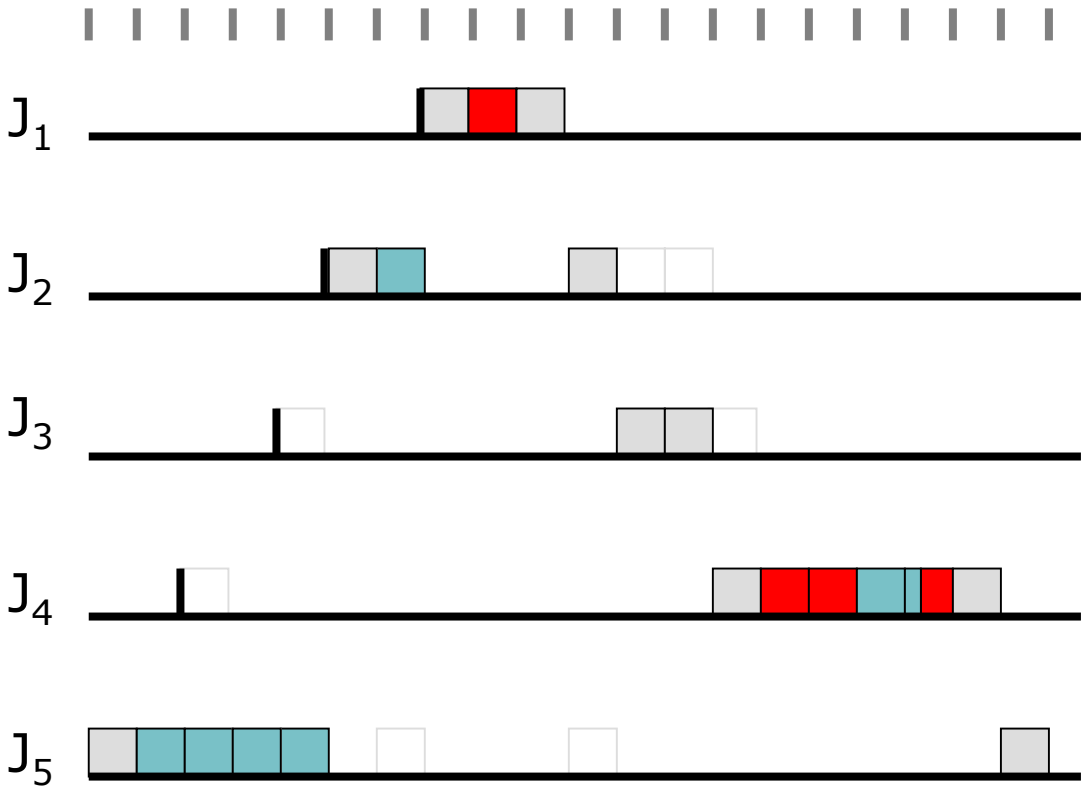
Comparison: Basic and Stack Based PCP



J_5 inherits priority when other jobs block on access to blue resource

Basic Priority Ceiling Protocol

Comparison: Basic and Stack Based PCP



Clear that context switches are reduced; nothing finishes later, but many tasks start later

Stack Based Priority Ceiling Protocol

The Ceiling Priority Protocol

- A similar implementation is the **ceiling priority protocol**
- Defining rules:
  - **Scheduling:**
    - **(a)** Every job executes at its assigned priority when it does not hold any resource. Jobs of the same priority are scheduled on a FIFO basis
    - **(b)** The priority of each job holding resources is equal to the highest of the priority ceilings of all resources held by the job
  - **Allocation:** whenever a job requests a resource, it is allocated
- When jobs never self-yield, gives identical schedule to the stack-based priority ceiling protocol
  - Essentially a reformulation of the stack based priority ceiling protcol
- Again, simpler than the basic priority ceiling protocol

Choice of Priority Ceiling Protocol

- If tasks never self yield, the stack based and ceiling priority protocols are a better choice
  - Simpler
  - Reduce number of context switches

- Stack based can be used to allow sharing of run-time stack, to save memory resources
- The ceiling priority protocol supported by the Real-Time Systems annex of Ada95

Resource Control in Dynamic Priority Systems

- Priority ceiling protocols described to date assume fixed priority scheduling
- In a dynamic priority system, the priorities each periodic tasks change over time, while the set of resources required by each task remains constant
  - As a consequence, the priority ceiling of each resource may change over time
- If a system is job-level fixed priority, but task-level dynamic priority, the basic priority ceiling protocol can still be applied
  - Each job in a task has a fixed priority once it is scheduled, but may be scheduled at different priority to other jobs in the task
    - Example: Earliest Deadline Scheduling
  - Update the priority ceilings of all jobs each time a new job is introduced; use until updated on next job release
  - Very inefficient: $O(\rho)$ complexity for ρ resources each time a job is released
- Stack based priority ceiling protocol can also be used

Messages, Signals and Events

- In addition to controlling access to resources, tasks often need to communicate information to other tasks
- Can be implemented using a shared data structure – a resource – that is managed as described previously
  - Example: a queue protected by a mutex and condition variable
  - Requires synchronisation between tasks
- May wish to communicate with another task without an explicit synchronisation step
  - Send another task a message
  - Signal another task that an event has occurred

POSIX Message Queues

Create and destroy message queues:

```
mpd_t mq_open(char *mqname, int flags, mode_t mode, struct mq_attr attrs);
int   mq_close(mpd_t mq);
int   mq_unlink(char *mqname);
```

Send and receive messages:

```
int  mq_send(mpd_t mq, char *msg, size_t msgsize, unsigned msg_prio);
int  mq_receive(mqd_t mq, char *msg, size_t buflen, unsigned *msg_prio);
```

Set and get attributes:

```
int  mq_setattr(pqd_t mq, struct mq_attr *newattr, struct mq_attr *oldattr);
int  mq_getattr(mpd_t mq, struct mq_attr *attrbuf);
```

Register for notifications:

```
int  mq_notify(mqd_t mq, struct sigevent *notification);
```

POSIX Message Queues

- Message queues are usually blocking:
  - mq_send() will block until there is space in the queue to send a message
  - mq_receive() will delay the caller until there is a message
- Can be set to non-blocking, if desired
- A receiver can register to receive a signal when a queue has data to receiver, rather than blocking

- Messages have priority, and are inserted in the queue in priority order
- Messages with equal priority are delivered in FIFO order

Message Based Priority Inheritance

- A message is not read until the receiving thread executes mq_receive()
- Problem:
  - Sending a high priority message to a low priority thread
  - The thread will not be scheduled to receive the message

- Solution: message based priority inheritance
  - Assume message priorities map to task priorities
  - When a task is sent a message, it provides a one-shot work thread to process that message
  - The work thread inherits the priority of the message
  - Allows message processing to be scheduled as any other job
  - Supported by the QNX service providers and message queues
  - Not supported by POSIX message queues
    - Kludge: run a service provider at highest priority so it receives all messages immediately, then lower the priority to that of the message while processing the request

Signalling Events

- Need a way of signalling a task that an event has occurred
  - Completion of asynchronous I/O request
  - Expiration of a timer
  - Receipt of a message
  - etc

- Many different approaches:
  - Unix signals
    - Event number N has occurred; no parameters; unreliable (non-queued)
  - POSIX signals
    - Allow data to be piggybacked onto the signal (a `void *` pointer)
    - Signals are queued, and not lost if a second signal arrives while the first is being processed
    - Signals are prioritised
  - Windows asynchronous procedure call and event loop

Signalling Events

- Signals are delivered asynchronously at high priority
  - As a result of a timer event
  - As a result of a kernel operation completing
  - As a result of action by another process
- High overhead
  - Require a trap to the microkernel, context switch, etc
- Add unpredictable delay
  - The executing process is delayed when a signal occurs, by the time taken to switch to the signal handler of the signalled task, run the signal handler, and switch back to the original task
- Work well for soft real time on general purpose systems
  - Overheads and unpredictability small compared to the other issues of running on such systems
- May be better to use synchronous communication where possible in hard real time systems
  - Easier to predict behaviour, since receiving the event can be scheduled

Clocks and timers

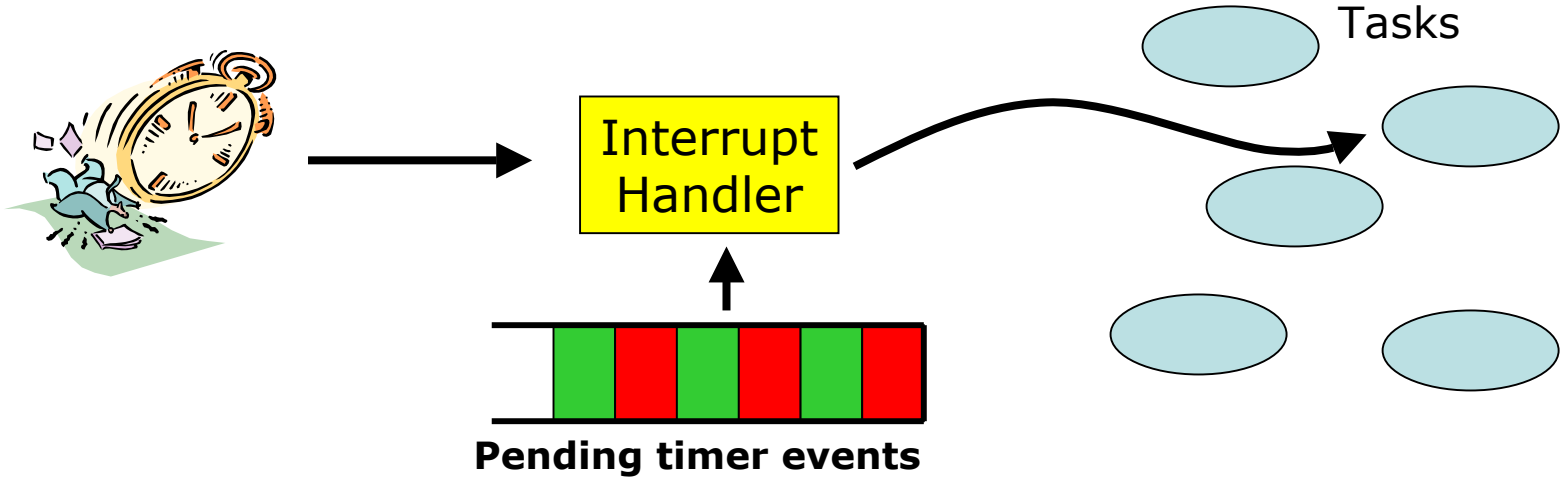
- Systems provide a timer interface, to request an event is delivered…
  - at a certain absolute time
  - after a certain delay

- In POSIX:

```
int timer_create(clockid_t clock, struct sigevent signal, timer_t *timer);
int timer_settime(timer_t *timer, int flag, struct itimerspec *new_interval,
                                            struct itimerspec *old_interval)
```

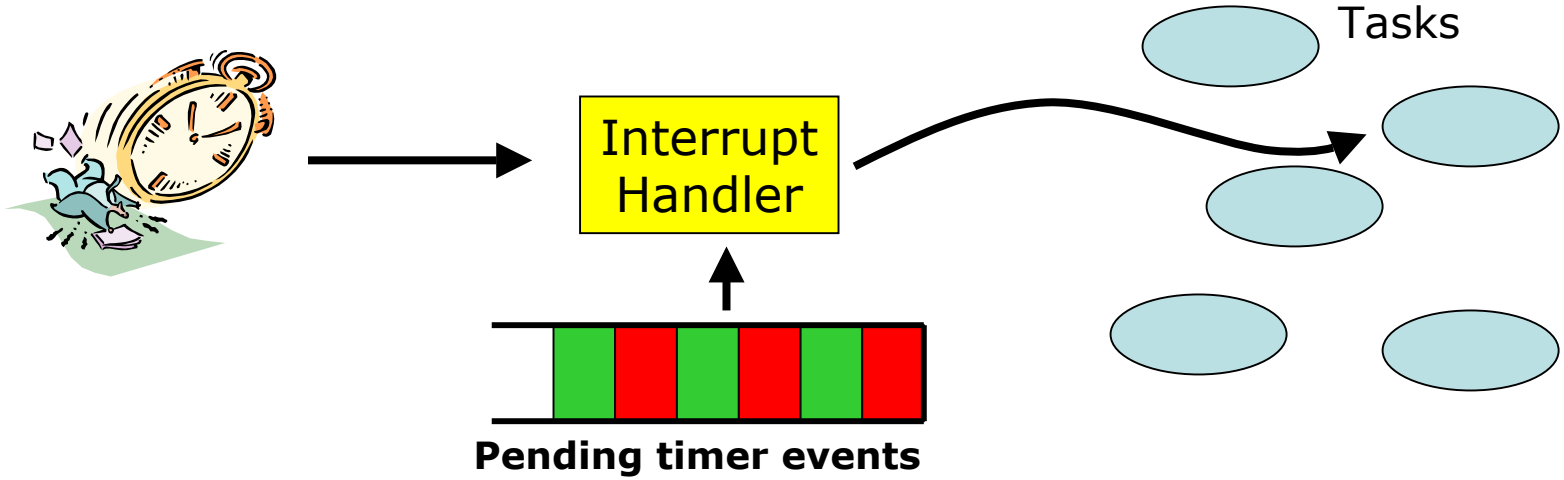
- Most systems support multiple timers, perhaps derived from different clocks

Time Services



- The clock will have certain resolution, accuracy and stability
  - Resolution and accuracy fixed, depend on hardware
    - Resolution: minimum interval that can be distinguished
    - Accuracy: is that interval correct? too large? too small?
  - Stability depends on environment and hardware
    - Is the accuracy constant?
    - Does it depend on temperature or system load?

Time Services



- Interrupt handler will have latency
  - Depends on the number of pending timer events
  - Decreases perceived accuracy of the clock
- Waking up the task receiving a timer event also has latency
- Both are non-deterministic, depending on system load
  - On general purpose system, may be 10s of milliseconds
    - Problematic for soft real time on general purpose systems
  - Smaller on dedicated RTOS

Watchdog Timers

- Most real time systems support a **watchdog timer**
- A high priority keep-alive timer
- Set to expire after a certain period
- Tasks regularly nudge the timer, to increase the expiry period
- If the timer expires, the system is assumed to have failed
- A high priority task is triggered to recover
  - E.g. reboot the system; restart the failed task

Summary

By now, you should know:

- Outline of POSIX synchronisation and lock facilities
  - Mutex support for priority inheritance/priority ceiling protocol
- How to implement priority inheritance
- Simpler alternatives to basic priority ceiling protocol
- Using priority ceiling in dynamic priority systems
- Outline of POSIX messages queues
  - Problem with message priority inheritance
- Outline of signals and timing services
  - Sources of inaccuracy
  - Normal and watchdog timers