Resource Access Control (2)

Real-Time and Embedded Systems (M)

Lecture 14

Lecture Outline

- Resources access control (cont'd):
  - Enhancing the priority ceiling protocol
    - Stack-based priority ceiling protocol
    - Ceiling priority protocol
  - Resource access control for dynamic priority systems
  - Effects on scheduling
- Implementing resource access control
  - Locking primitives
    - Semaphores
    - Mutexes
    - Typical priority inheritance features
  - Messages, signals and events
    - Priority inheritance features for messaging

Enhancing the Priority Ceiling Protocol

- The basic priority ceiling protocol gives good performance, but the defining rules are complex

- Also, can result in high context switch overheads due to frequent blocking if many jobs contend for resources

- This has led to two modifications to the protocol:
  - The stack-based priority ceiling protocol
  - The ceiling priority protocol
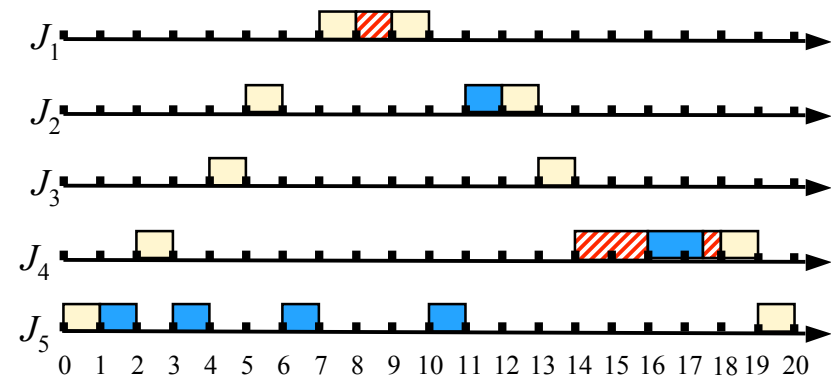
Stack-Based Priority Ceiling Protocol

- Based on original work to allow jobs to share a run-time stack, extended to control access to other resources

- Defining rules:
  - Ceiling: When all resources are free, $\Pi(t) = \Omega$; $\Pi(t)$ updated each time a resource is allocated or freed
    - $\Pi(t)$ current priority ceiling of all resources in currently use
    - Ω non-existing lowest priority level
  - Scheduling:
    - After a job is released, it is blocked from starting execution until its 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 the scheduling rule is not

Stack-Based Priority-Ceiling Protocol

- Consider an example system, with parameters are shown on the right →

- Jobs J_1, J_2, J_4 and J_5 attempt to lock their first resource after one unit of execution; J4 accesses ▢ after an additional 2 units of execution

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

Context switches are reduced compared to the basic priority ceiling protocol; no jobs finish later, but many jobs start later

Jobs blocked from starting since $\pi_i < \Pi$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\Pi=$	Ω	2	2	2	2	Ω	2	Ω	1	Ω	Ω	Ω	Ω	Ω	1	1	1	1	Ω	Ω	Ω
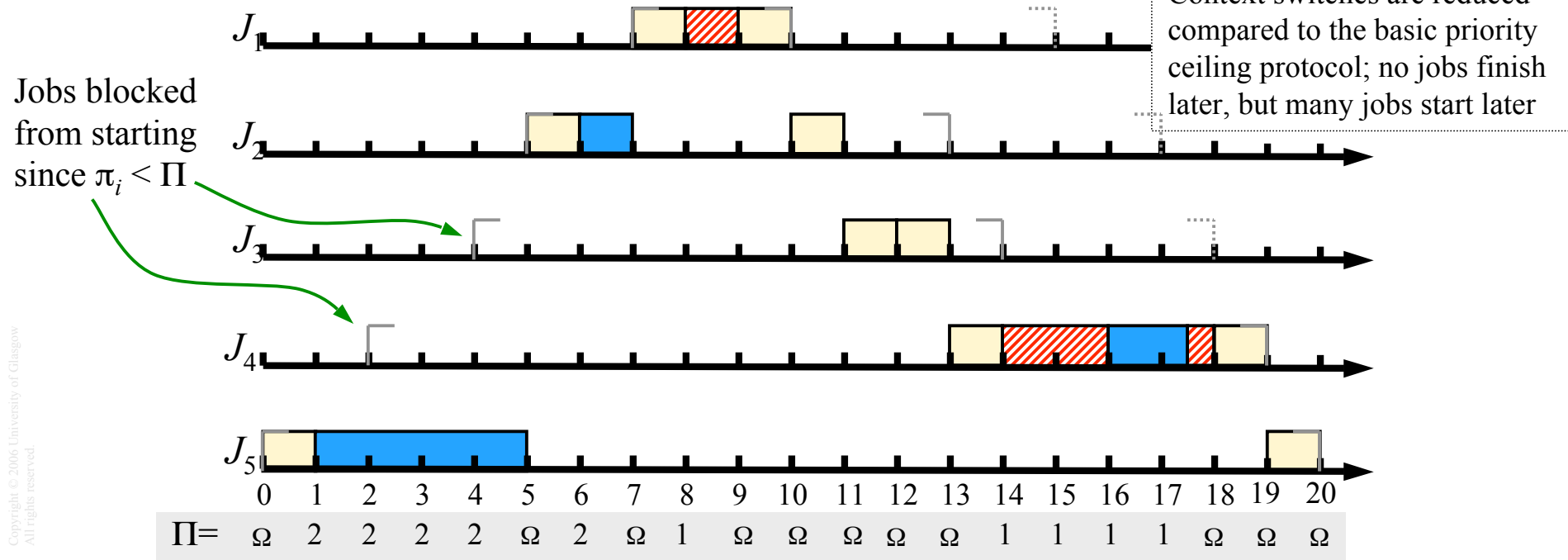
Stack-Based Priority Ceiling Protocol

- Characteristics:
  - When a job starts to run, all the resource it will ever need are free (since otherwise the ceiling would be ≥ priority)
    - No job ever blocks waiting for a resource 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
    - i.e. not worse than the duration of one critical section

Ceiling Priority Protocol

- A similar algorithm is the *ceiling priority protocol*

- Defining rules:
  - Scheduling:
    - 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
    - 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* schedules to the stack-based priority ceiling protocol

- Again, simpler than the basic priority ceiling protocol

Choice of Priority Ceiling Protocol

- If tasks never self yield, the stack based priority ceiling protocol is a better choice than the basic priority ceiling protocol

  - Simpler

  - Reduce number of context switches

  - Can also be used to allow sharing of the run-time stack, to save memory resources

- Both give better performance than priority inheritance protocol

  - Assuming fixed priority scheduling, resource usage known in advance

Resources in Dynamic Priority Systems

- The priority ceiling protocols 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 changes over time
  - Example:

$T_1 = (2, 0.9)$
$T_2 = (5, 2.3)$

EDF

$\pi(T_1) = 1$ $\pi(T_1) = 2$ $\pi(T_1) = 1$
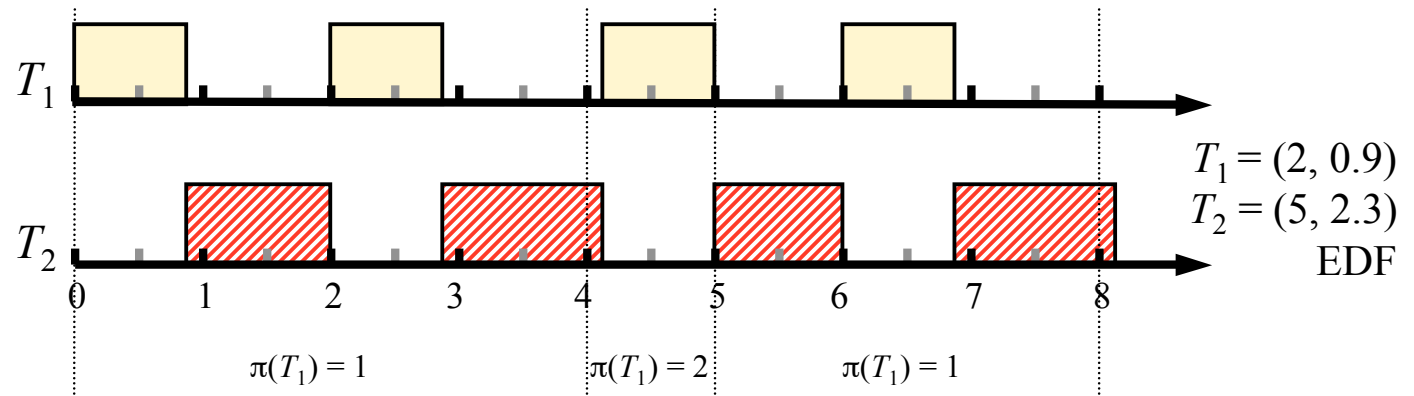
  - What happens if T_1 uses resource X, but T_2 does not?
    - Priority ceiling of X is 1 for $0 \leq t \leq 4$, becomes 2 for $4 \leq t \leq 5$, etc. even though the set of resources required by the tasks remains unchanged

Resources in Dynamic Priority Systems

- If a system is job-level fixed priority, but task-level dynamic priority, a 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

- Has been proven to work and have the same properties as priority ceiling protocol in fixed priority systems
  - But: very inefficient, since priority ceilings updated frequently
  - May be better to use priority inheritance protocol, accept longer blocking

Maximum Duration of Blocking

- Assume J_1 and J_2 contend for a resource, R, where J_1 is the higher priority job

Blocking time

J_1

J_2

J_2 pre-empted immediately after it locks resource R
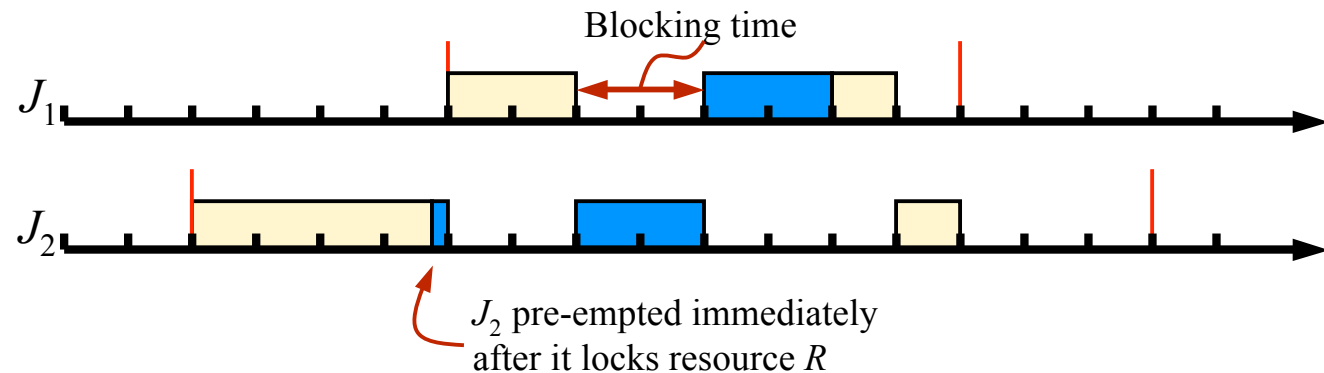
- Worst case blocking time tends towards the duration of J_2's critical section over R

- When using priority inheritance protocol, J_2 might be transitively blocked for the duration of the next priority job's critical section
  - Worst case: it is blocked by *every* other lower priority job, for the full duration of each lower priority job's critical section

Maximum Duration of Blocking

- The priority ceiling protocols implement avoidance blocking, and so do not exhibit transient blocking
  - Block for *at most* the duration of one low priority critical section
    - Direct blocking: low priority jobs locks resource; can be blocked for up to the duration of the critical section of that job
    - Avoidance blocking: resource is free, but priority ceiling rules deny access

- Calculate worst case blocking duration:
  - Simple:
    - Assume can block for duration of longest critical section of lower priority jobs
    - Probably overestimates blocking duration; likely not too significant
  - More efficient:
    - Trace direct conflicts with lower priority jobs, find longest critical section
    - Trace indirect conflicts with lower priority jobs that may inherit priority and cause avoidance blocking, find longest critical section
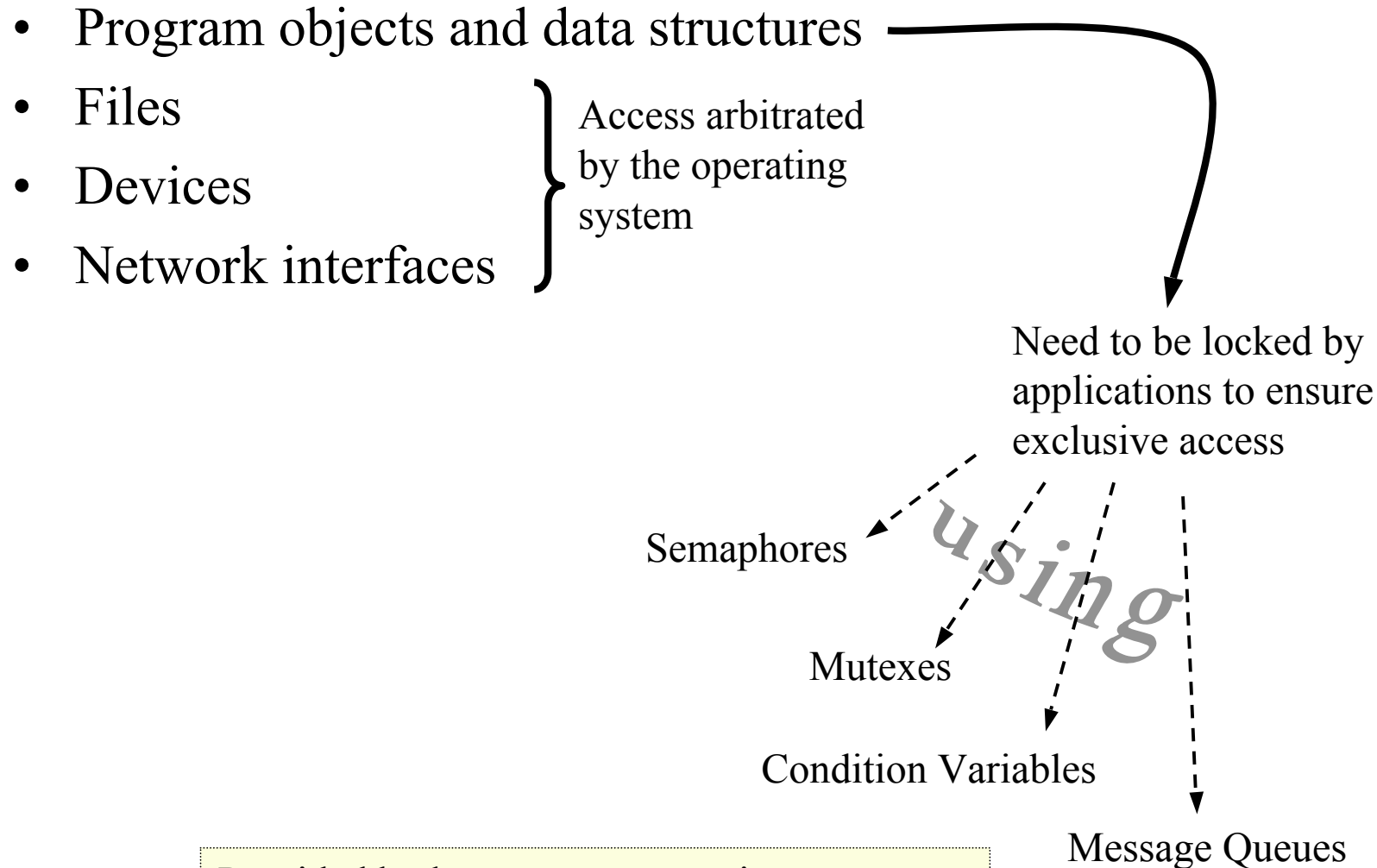   - Greatest of these is maximum possible blocking time

Effects on Schedulability Tests

- Jobs which block due to resource access impact schedulability

- How to adjust schedulability test?

  - Incorporate maximum blocking time as part of execution time of job; schedulability test then runs as normal

  - Priority ceiling protocols clearly preferred where possible

Implementing Resource Access Control

- Have focussed on resource access control algorithms which can be implemented by an operating system

- How are these made available to applications?
    - Some implemented by the operating system
    - Some implemented at the application level

Resource Types and Locking

- Program objects and data structures
- Files
- Devices
- Network interfaces

Access arbitrated by the operating system

Need to be locked by applications to ensure exclusive access

using

Semaphores

Mutexes

Condition Variables

Message Queues

Provided by language or operating system – focus on POSIX as a representative example

POSIX Semaphores

- Semaphores provide a simple locking abstraction:

```
int sem_init(sem_t *sem, int inter_process, unsigned init_val);
int sem_destroy(sem_t *sem);


int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

- Embed a semaphore within an object for resource access control:

```
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);
```

- Example of a feature with no special real-time features or priority control

POSIX Mutexes

- A higher level locking mechanism for real-time applications is a POSIX mutex, which controls priority during resource access
    - As with semaphores, a mutex is embedded in an object at a location of the programmers choosing to control access to that object/resource
    - Basic API:

```
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);

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  proto);
int pthread_mutexattr_getprotocol(pthread_mutex_attr_t *attr, int *proto);
```

POSIX Mutexes: Priority Inheritance

- Can specify the resource access protocol for a mutex:
  - Use `pthread_mutexattr_setprotocol()` during mutex creation
    - `PTHREAD_PRIO_INHERIT`   Priority inheritance protocol applies
    - `PTHREAD_PRIO_PROTECT`   Priority ceiling protocol applies
    - `PTHREAD_PRIO_NONE`   Priority remains unchanged
  - If the priority ceiling protocol is used, can adjust the ceiling to match changes in thread priority (e.g. dynamic priority scheduling):
    - `pthread_mutexattr_getprioceiling(…)`
    - `pthread_mutexattr_setprioceiling(…)`

- Used with POSIX real-time scheduling:
  - Allow implementation of fixed priority scheduling with a known resource access control protocol
  - Controls priority inversion, scheduling; allows reasoning about a system

POSIX Condition Variables

- POSIX also defines a condition variable API:

```
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
```

(timed wait with priority inheritance)

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

- But may want to communicate with another task without explicit synchronisation step
    - Send another task a message
    - Signal another task that an event has occurred

POSIX Message Queues

- A message queue abstraction provided for this purpose:

```
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);


int   mq_send(mpd_t mq, char *msg, size_t len, unsigned prio);
int   mq_receive(mqd_t mq, char *msg, size_t len, unsigned *prio);


int  mq_setattr(pqd_t mq, struct mq_attr *new, struct mq_attr *old);
int  mq_getattr(mpd_t mq, struct mq_attr *buf);
```

- Blocking `mq_send()` and `mq_receive()` typical
  - Can be set to non-blocking, if desired
  - Receiver can be signalled when data arrives, rather than blocking
- Messages have priority, inserted in the queue in priority order
  - Messages with equal priority are delivered in FIFO order

Message Based Priority Inheritance

- Messages not read until 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, which inherits the priority of the message
  - Allows message processing to be scheduled as any other job
  - Implemented by some RTOS (e.g. QNX); not common
    - Typically simulate using a queue with a priority inheriting mutex

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 kernel trap, context switch, etc

- Add unpredictable delay

  - 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

- May be better to use synchronous communication where possible in real time systems, since easier to predict

Implementation Summary

- As seen, many approaches to implementing resource access control

- POSIX provides useful baseline functionality
  - Priority scheduling abstraction, to implement Rate Monotonic schedules
  - A mutex abstraction using either priority inheritance or priority ceiling protocols to arbitrate resource access

- Similar, sometimes more advanced features, provided by other real-time operating systems
  - E.g The Ada language supports resource access control with the priority ceiling protocol
  - E.g. QNX support message based priority inheritance

Summary

- Illustrated operation of additional resource access control protocols, simplifying priority ceiling protocol

- Discussed impact on schedulability

- Described some methods to implement resource access control:

    - Use of POSIX real-time extensions and mutexes for locking, to directly implement the ideas described

    - Other mechanisms: semaphores, message queues, signals, etc.