Operating systems and concurrency (B10)

David Kendall

Northumbria University

Introduction

This lecture looks at

- Some approaches to thread scheduling
  - Illustrates the use of thread attributes
- Some problems with the use of semaphores, including the problem of priority inversion in systems that use fixed priority preemptive scheduling
- How to use `pthreads` mutexes to overcome some of the problems of semaphores
  - Illustrates the use of mutex attributes

Thread scheduling in `pthreads`

- The *scheduler* is the part of the operating system that decides which process/thread to run next
- Usually, we are happy to use the default scheduling policy provided by the operating system running our application
- In Linux this policy is referred to as `SCHED_OTHER` (or `SCHED_NORMAL`)
- Under the `SCHED_NORMAL` policy, the operating system tries to allocate CPU resources to threads as fairly as possible by dynamically taking into account how long each thread has been waiting and allocating a CPU to the thread that has been waiting the longest (Linux calls this the *completely fair scheduler*)

Thread scheduling in `pthreads`

- However, it is also possible to specify that some threads are scheduled using a different policy:
  - the `SCHED_FIFO` policy allows threads to be given a *priority*; the scheduler tries to ensure that the highest priority, ready thread is able to run. It continues to run either until it blocks (e.g. for I/O or time delay) or until a higher priority thread becomes ready. This provides a *fixed priority preemptive* scheduling mechanism
  - the `SCHED_RR` policy is similar except that each thread running with this policy may also be preempted at the end of its *time quantum*
- `SCHED_FIFO` is often required in embedded Linux applications such as for smart TVs, set-top boxes, wireless routers, medical devices, cameras etc
- For example, this allows a thread that needs to respond promptly to input from a special device to be given a higher priority than a thread whose function is simply to log activity in the system

Thread scheduling in `pthreads`

A thread that will be scheduled using `SCHED_FIFO` must be created with the proper *thread attribute*.

- Declare variables of type `pthread_t` and `pthread_attr_t` for the thread and its attribute. Also declare an `int` to receive the result of the `pthread` operations

```
static pthread_t thread;
static pthread_attr_t attr;
static int rc;
```

- Initialise the attribute variable

```
rc = pthread_attr_init(&attr);
assert(rc == 0);
```

- Set the scheduling inheritance attribute to allow scheduling attributes to be set explicitly (otherwise they will be inherited from the thread that creates the new thread)

```
rc = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
assert(rc == 0);
```

Thread scheduling in `pthreads`

- Set the scope over which thread scheduling is performed to the whole system (i.e. all current processes and threads). This is in contrast to process scope, where the scheduling occurs only for threads within the the owning process and is effective only when that process is scheduled

```
rc = pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
assert(rc == 0);
```

- Set the scheduling policy to `SCHED_FIFO`

```
rc = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
assert(rc == 0);
```

- Finally, create the thread, using the attribute that we have just constructed

```
rc = pthread_create(&thread, &attr, thread_function, NULL);
assert(rc == 0);
```

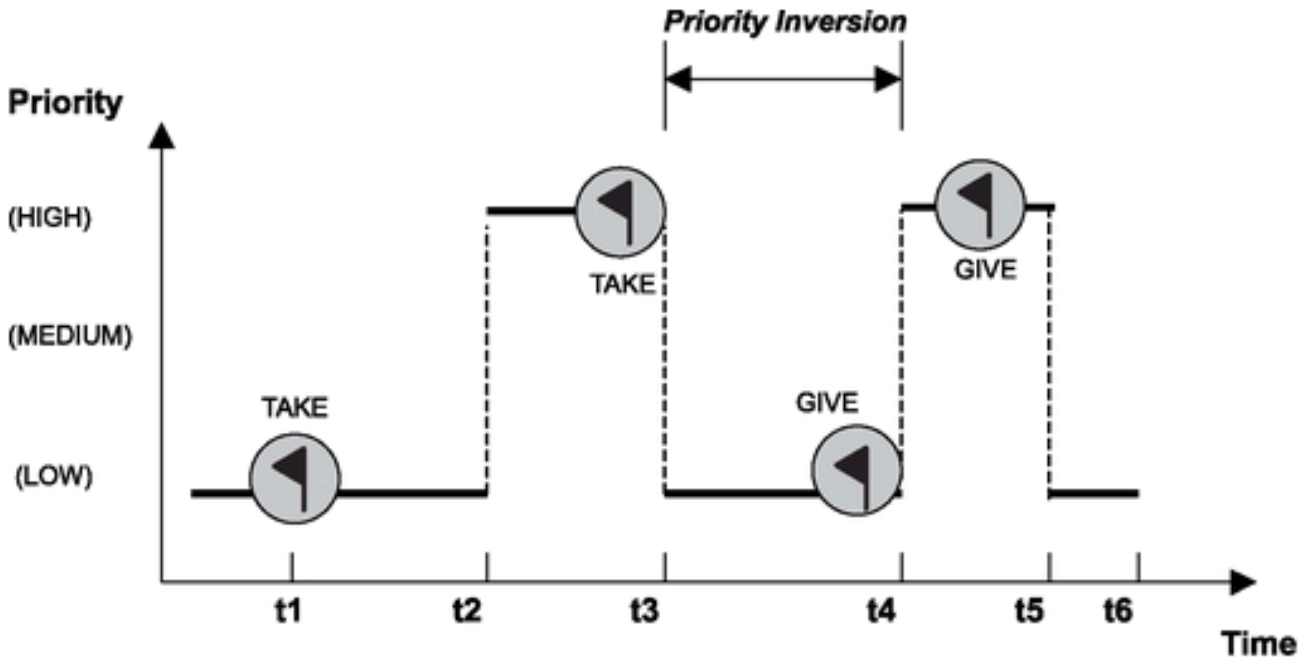
Problems with the use of semaphores

- Accidental release
- Recursive deadlock
- Thread-death deadlock
- Priority inversion
- Semaphore as a signal
- For details see: Cooling, N, *Mutex vs Semaphores* Parts 1 and 2, Sticky Bits Blog, 2009

Priority Inversion

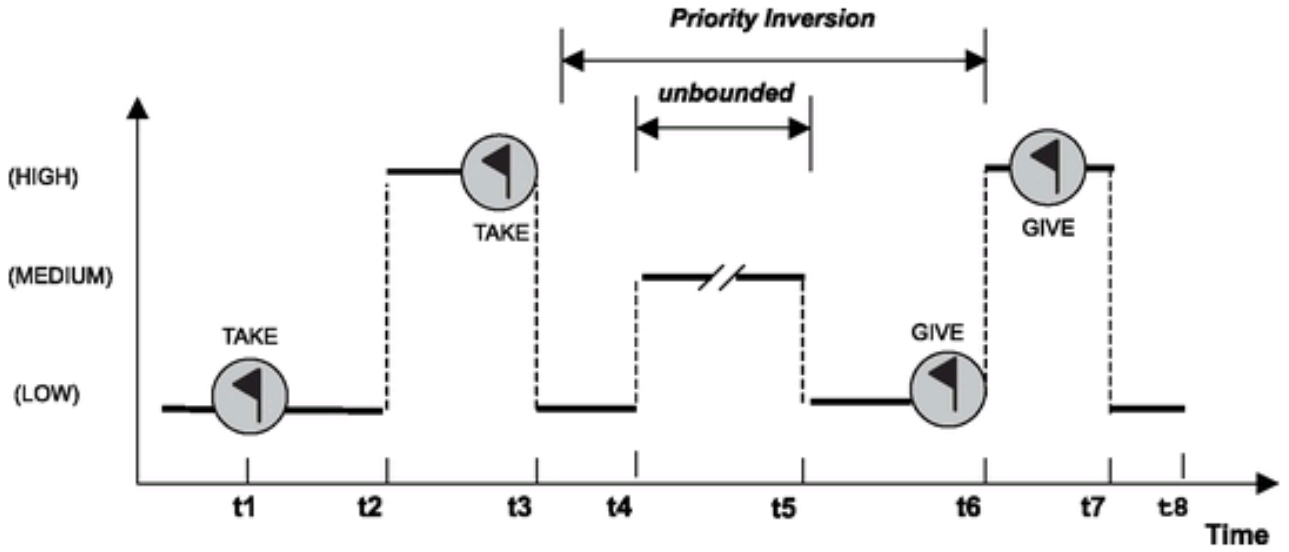
- Fixed priority preemptive scheduling with blocking on access to shared resources can suffer priority inversion.
- This can cancel the benefits of `SCHED_FIFO` scheduling
- Priority inversion occurs when
  - Low priority thread is allowed to execute while higher priority thread is blocked.
- Look at priority inversion in more detail
- Consider possible solution to the priority inversion problem

Bounded priority inversion



- At time t_1 low priority (LP) thread acquires semaphore
- At time t_2 high priority (HP) thread preempts LP
- At time t_3 HP tries to acquire semaphore and is blocked
- At time t_4 LP returns semaphore, HP acquires semaphore, is unblocked and runs
- At time t_5 HP finishes and LP runs again

Unbounded priority inversion



- Priority inversion again occurs at time t_3
- At time t_4 LP is preempted by a medium priority thread (MP) that is able to run because it does not need to acquire the locked semaphore
- MP runs until completion at time t_5
- Duration of period from t_4 to t_5 is very difficult to predict (unbounded)

Problems caused by priority inversion

- Thread completion times vary more widely
- e.g. MP completes earlier in the priority inversion case than in the cases when priority inversion does not occur:
  - HP runs first, acquires semaphore, runs to completion, MP runs, ...
  - MP runs first, preempted by HP, ...
- Thread completion time of HP is delayed in the priority inversion case – may miss deadline

Solar system's most famous priority inversion

Mars Pathfinder (1997)

- Mars Pathfinder mission
  - Launched Dec 1996, landed July 1997
  - Lander and rover
  - Total project cost approx. $280 million
- Almost failed due to a software problem
- NASA engineers attempted to debug the software over a radio link to Mars

Solar system's most famous priority inversion

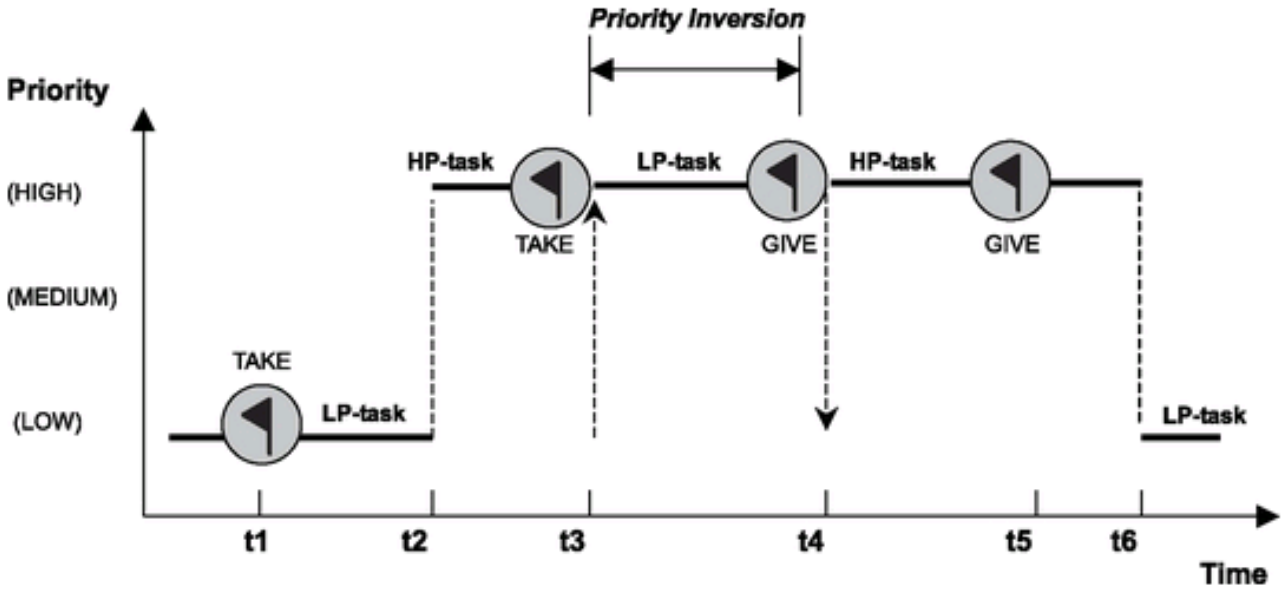
Mars Pathfinder (1997)

- Smallest recorded distance between Earth and Mars is $5.6 \cdot 10^{10}$ m
- Assuming radio waves travel through space at the speed of light, the smallest propagation delay of a message from Earth to Mars is over 3 minutes
- Communicating at these speeds, they managed to work out that the problem was caused by priority inversion
- They transmitted a software upgrade that turned on a priority inheritance protocol in the VxWorks OS.
- The mission continued successfully

What happened on Mars?

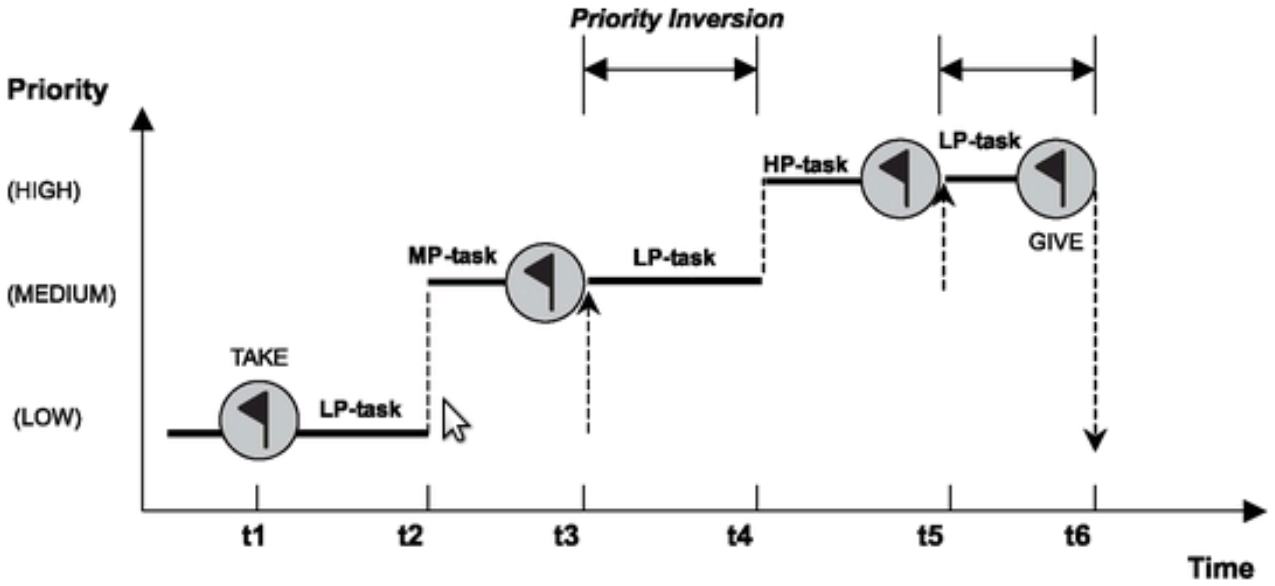
- Various devices communicated over a data bus.
- Activity on bus managed by pair of high-priority threads, one of which communicated through pipe with low-priority meteorological science thread.
- The meteorological thread was preempted by medium-priority threads while it held a mutex related to the pipe.
- While low-priority thread preempted, the high-priority bus distribution manager tried to send more data to it over the same pipe.
- Mutex still held by meteorological thread, so bus distribution manager made to wait.
- When other bus scheduler active, it noticed that the distribution manager hadn't completed its work for that bus cycle and forced a system reset.

Priority Inheritance Protocol (PIP)



- Consider a thread T trying to acquire a resource R
- If R is in use, T is blocked
- If R is free, R is allocated to T
- When a thread of higher priority attempts to access R, priority of T is raised to priority of higher priority thread
- When it releases R, priority of T is set to maximum of its original priority and priorities of any threads it's still blocking by holding other resources

Transitive priority promotion in PIP



- PIP is *dynamic* – a thread does not have its priority raised until a higher priority thread tries to acquire a resource that it holds
- Priority continues to rise as other higher priority threads try to acquire its resource. . . and falls again as it releases resources

Pros and cons of PIP

- Pros
  - All priority inversions are bounded when PIP is used
  - Supported by POSIX (`PTHREAD_PRIO_INHERIT`)
- Cons
  - Frequent changes of priority may become a significant overhead
  - Deadlock is possible – MP acquires some resources needed by HP, HP acquires some resources needed by MP, when LP releases resource, HP runs to deadlock

Priority ceiling protocols

- Two main versions of priority ceiling protocol
  - Original Ceiling Priority Protocol (OCPP)
  - Immediate Ceiling Priority Protocol (ICPP)
- Both intended to reduce the number of priority changes required and to prevent deadlock
- Only ICPP is considered here - this is the version of the protocol supported by POSIX (`PTHREAD_PRIO_PROTECT`)

Immediate Priority Ceiling Protocol

- Each thread has a static default priority
- Each resource has a static ceiling value that should be at least as high as the maximum priority of any of the threads that use it
- A thread also has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked
  - i.e. if a thread successfully locks a resource whose ceiling value is higher than the current dynamic priority of the thread then the thread's dynamic priority is set to the ceiling value of the resource that it has locked

Properties of ICPP

- No deadlock
- A thread can be blocked only at the very beginning of its execution
    - Once a thread starts executing, all the resources it needs must be free
    - If they were not, some thread would have an equal or higher priority and the thread's execution would be postponed

Reminder: Mutexes in `pthreads`

The `pthreads` mutex was introduced briefly in an earlier lecture. Here is a reminder.
In `pthreads` the mutex type is `pthread_mutex_t` and the main operations available for use on variable `m` of type `pthread_mutex_t` are:

- `pthread_mutex_init(&m , NULL)`
  initialises the mutex `m`; instead of `NULL` we can pass a pointer to a mutex attributes structure — focus on this next
  Can also initialise `m` statically, e.g.
  `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- `pthread_mutex_lock(&m)`
  used to lock the mutex; if the mutex is already locked, the calling thread will be suspended
- `pthread_mutex_unlock(&m)`
  used to unlock the mutex; if there are threads suspended waiting for the mutex, one of them will be made ready to run

Mutex: fixing the semaphore?

- Mutex introduces the principle of ownership
  - a mutex can be released only by the thread that acquired it
  - a thread that tries to release a mutex that it didn't acquire causes an error and the mutex remains locked
- accidental release much more difficult
  - signalling not allowed
- Depending on implementation, additional features can be supported:
  - recursive locking can be allowed - mutex must be released as many times as it has been acquired
  - thread-death deadlock can be recovered by recognising that mutex is owned by thread that no longer exists and released
  - priority inversion can be reduced using e.g. priority inheritance protocol or priority ceiling protocol
  - POSIX `pthreads` supports all of these ideas through *mutex attributes*

Mutex attributes in `pthreads`

- We give an example of the use of `pthreads` mutex attributes here by showing how to create and use a mutex that supports the priority ceiling protocol, so avoiding the problem of priority inversion
- The use of other `pthreads` mutex attributes is very similar
- For more details refer to the POSIX standard and search for `pthread_mutexattr`

Creating a PCP mutex in `pthreads`

- Declare variables of type `pthread_mutex_t` and `pthread_mutexattr_t` and an `int` variable to receive the return value of the `pthreads` operations

```
static pthread_mutex_t mutex;
static pthread_mutexattr_t mutex_attr;
int rc;
```

Note that `mutex` must be declared globally but `mutex_attr` and `rc` can be local to e.g. `main`

- Initialise the attribute variable

```
rc = pthread_mutexattr_init(&mutex_attr);
assert(rc == 0);
```

Creating a PCP mutex in `pthreads`

- Set the protocol for PCP (the priority ceiling protocol)

```
rc = pthread_mutexattr_setprotocol(
      &mutex_attr, PTHREAD_PRIO_PROTECT);
assert(rc == 0);
```

- Set the ceiling priority to a value that is greater than the priority of any of the threads that use the mutex (we call this `CEILING_PRIORITY` here for the sake of example)

```
rc = pthread_mutexattr_setprioceiling(
      &mutex_attr, CEILING_PRIORITY);
assert(rc == 0);
```

- Finally, initialise the mutex

```
rc = pthread_mutex_init(&mutex, &mutex_attr);
assert(rc == 0);
```

Acknowledgements

- Cooling, N., Mutex vs Semaphores Parts 1 and 2, Sticky Bits Blog, 2009
- Kalinsky, David and Michael Barr. "Priority Inversion," Embedded Systems Programming, April 2002, pp. 55-56.
- Li, Q. and Yao, C., Real-time concepts for embedded systems, CMP, 2003
- Sha, L. Rajkumar, R. and Lehoczky, J. *Priority Inheritance Protocols: An Approach to to Real-Time Synchronization*. IEEE Transactions on Computers, 39 (9): 1175–1185; September, 1990
- POSIX standard