Real-Time Architectures 2003/2004

Mapping on execution platform

Johan Lukkien

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Static organization

- Layering
- Architecture of Application: single executive or modular
- Architecture of RTOS: monolithic or micro kernel

Application
RTOS interface: system calls
RTOS kernel
Hardware/software interface: drivers
Hardware

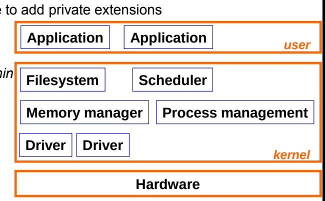
Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Contents

- OS architecture
- Standardization: POSIX

RTOS Architecture: Monolithic OS

- OS as a (shielding) layer
  - all device control, i/o, (virtual) memory etc. under OS-supervision
  - each OS-API-call generates a trap into the kernel
  - difficult to tailor to specific application
    - requires special generation, compilation, versioning
    - need source code to add private extensions
  - good performance
    - no traps to communicate *within* the kernel
  - memory protection only in application

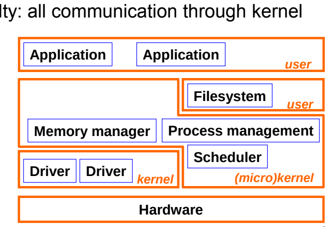


RTOS Architectures: Micro-kernel

- General (as a software architecture):
  - *"separates a minimal functional core from*
    - *extended functionality and*
    - *customer-specific parts*
  - *and serves as a socket for plugging-in these extensions and coordinate their collaboration"*
    (from "Pattern-oriented software architecture", Buschmann et.al.)
- Minimal functional OS core:
  - memory management
  - basic interrupt handling
  - task management, scheduling
  - "plugging" support for the implementation of "internal services"
    - drivers etc.
- nano-kernel, pico-kernel

Micro-kernel RTOS

- Drivers, filesystem, IO and many other typical OS tasks go outside the kernel as *external* or *internal* services
  - easy to change ('hot' pluggable) or to remove
  - standard memory protection: no system crash in case of error
- Performance penalty: all communication through kernel
- Typical communication facility between system components: message passing
  - natural view on distribution



OS Requirements

- Predictable in time (bounded, deterministic event response)
  - *known* performance versus *high* performance
  - maximum latencies (response times) of API calls
  - provide handles to deal with blocking calls
    - e.g., dealing with priority inversion, buffering
  - pre-emption, also of Interrupt Service Routines
- Predictable in memory use
  - footprint known, usually small
  - good relation between functionality and memory use
  - no 'cost' for unused functionality
    - adjustable, e.g., leave out file-system in embedded application
- Extensible
  - support for adding user-specific functionality
- Scalable
  - wide range of environments, functionalities

OS Requirements (cnt'd)

- Derived requirements for RT systems
  - dependable
    - robust, correct, safe & secure
- Support for real-time control
  - (pre-emptive) scheduling policies
  - explicit control over resources
  - real-time facilities: clocks and timers
- Regular OS tasks
  - multi-threading, priorities [enough!], pre-emption, memory management

Choice of RTOS

- OS defines programmers view on the system
  - architectural limitations
  - expressiveness of API
    - view on concurrency
      - processes & threads
    - supported synchronization & communication facilities
      - preferred ones [performance]
      - expressiveness
- OS dependency limits portability
  - systematic porting of model may be inefficient
  - essential elements of the API may not be supported on another OS
- Solution, in principle: portable OS interface

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Contents

- OS architecture
- Standardization: POSIX
  - introduction
  - processes & threads
  - synchronization
  - timing & scheduling
  - message passing

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Standardization: POSIX

- Portable Operating System Interface
  - UNIX-like
- Goal: source-code portability of applications
  - in practice: just reduce portability effort
- Standard, IEEE, ANSI, ISO, developed in chapters:
  - mandatory ('base') & optional parts, per chapter
  - set of chapters covers 'everything'
- Versioning: Posix 1003.1x-year, where x is the chapter
  - no x: basic set of systems calls, like UNIX
  - x=b: real-time extensions (also: POSIX.4)
  - x=c: multi-threading support (also: POSIX.4a)
  - x=g: sockets

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

POSIX compliance

- POSIX standardizes on the API !!
  - coupling with RTOS architecture mainly a mapping problem
- A system supporting POSIX provides
  - a host language and compiler
  - interface definition files (e.g., C-header files)
    - including standardized ways to define included optional parts
  - interface implementation binary or code (e.g., C-libraries)
  - a run-time system (a platform: OS or the like)
- **NOTES**:
  - POSIX is NOT Unix System V
  - Many RTOS'es have a POSIX face
    - though it may not be the most efficient way to use the RTOS
    - micro-kernel: POSIX as an external server (API)

Kernel entities

- Kernel entities are objects under exclusive control of the kernel
  - access and manipulation only through the kernel
- Examples of possible kernel entities
  - a process
  - a thread
    - but often not
  - a shared data object
    - e.g., message queue, semaphore or just a memory segment
  - internal kernel data structures
- Operations involving kernel entities
  - require at least one switch to kernel mode and back
  - are therefore more expensive

Contents

- OS architecture
- Standardization: POSIX
  - introduction
  - processes & threads
  - synchronization
  - timing & scheduling
  - message passing

Create new process

- *fork()* creates two identical copies; only '*child*' differs
- *exec(lp)* overwrites process with argument (an executable)

```
pid_t child;

child = fork();
if (child<0) /* error occurred */  perror ("fork");

if (child == 0) /* the child */ {
      execlp ("child program", "own name", arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror ("execlp");
}
else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from this routine */
}
```

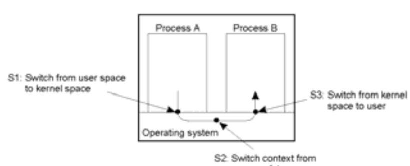
Termination of children

- Use *exit(status)* to terminate
- Need to wait for children to free child's resources
  - on many systems at least
- Functions *wait(), waitpid()*
- Asynchronous notification: signals

```
child:
    ..... exit (23);

parent:
    pid_t child, terminated; int status;

    /* blocking wait */
    while (child != wait (&status)) /* nothing */;

    /* polling wait */
    terminated = (pid_t) 0;
    while (terminated != child) {
        terminated = waitpid (child, &status, WNOHANG);
        /* other useful activities */
    }
    /* both cases: status == 23 */
```


Process overhead

- Creation
- Switching
  - for each Inter Process Communication – no shared memory
  - TLB, MMU



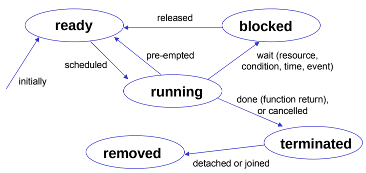
Multi-threading (pthreads, 1003.1c)

- Multiple threads within a process
  - introduces all the advantages and disadvantages of shared memory
- Rule: blocking of a thread must not influence other threads
  - also not in system calls
  - avoids simple thread-simulation implementation
    - run-to-completion
    - not real-time!
  - some older UNIX (SOLARIS, Distributed Computing Environment) implementations do not abide by this

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

The life of a thread (in POSIX)

- Main program: own thread
- Additional threads:
  - created on demand or on receipt of signal
  - thread code: a function in the program



Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

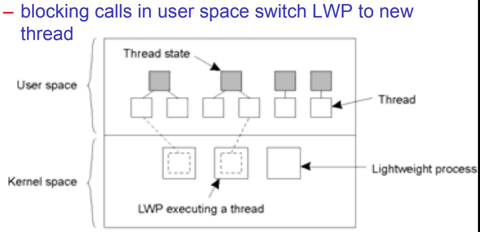
Thread execution model

- **Concurrency level**: number of "engines" (virtual processors) actually executing the threaded program
- Virtual processors are scheduled by the kernel
- Concurrency level choices
  - 1: no concurrency, no kernel activity in switching
  - # threads: switching always becomes kernel activity
  - in between: only blocking kernel calls and virtual processor scheduling requires kernel activity

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Thread execution: Solaris

- Virtual processor: lightweight process
  - LWP just executes threads (no memory space)
  - blocking calls in user space switch LWP to new thread



Thread safety and re-entrance

- Particular problem: shared libraries and system data
  - e.g. errno
    - pthreads functions don't change the processs errno but rather return an error state
    - pthreads functions have an optional extra pointer argument for useful additional information, also in case of error
    - there is an errno per thread
  - libraries should be *thread-safe*
    - can be called by multiple threads
    - doing it yourself: watch out for code versus data protection
      - don't just protect a call
  - Other term: re-entrant
    - "efficient thread-safe" – usually affecting the interfaces
      - e.g., by putting library data into user space

Contents

Communication facilities

- Shared memory [with multi-threading]
  - just shared variables, e.g., event flags
  - semaphores, mutexes
  - condition variables
  - readers/writers locks
- Message passing
  - streaming: pipes, fifos, sockets
  - structured: message queues
- Signals

Namespace

- Set of names
  - Specification/algorithm as how to generate names
- Methods
  - hierarchical, mostly
    - /user/johanl/....
    - absolute, relative names
  - prefixing
    - embedding a set of identifiers uniquely into a larger set
      - e.g. <uml: specific tag>
- Application
  - file system
  - more general: referring to any resources
- Posix names
  - needed to have references across processes and invocations
  - for portability,
    - start names with '/'
    - do not use any subsequent '/'

Shared memory (1003.1b)

- Memory that can be accessed by two or more processes
- The situation after shared memory initialization:
  - a piece of memory has been created that can be referred to by two or more processes at the same time
  - the shared memory is a kernel entity but references to it are in user space
- Processes are 'free' to place any data in shared memory
  - can use this to optimize operation and avoid kernel operations
  - synchronization own responsibility, as with threads (see later)

Shared memory interface

- Naming is more general than parent-child relationships
  - new name within kernel
  - persistent until re-boot
- Two-step approach
  - handle is a file descriptor
  - this file is mapped into memory

```
int fdes;

fdes = shm_open (name, flag, mode); /* file descriptor with limited functionality */
status = ftruncate (fdes, totalsize);     /* set the size or resize it                  */
status = shm_close (fdes);                /* memory object remains there                */
status = shm_unlink (name);               /* destroy                                    */

shm_area = mmap (WhereIWantIt, len, protection, flags, fd, offset);
                                      /* finally gives a pointer to the shared memory */
```

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Counting semaphores (1003.1b)

- Naming and creation similar as with shared memory
  - new name within kernel, persistent until re-boot
  - also “unnamed” semaphores, for use in shared memory
    - two interfaces for creation and destruction
    - can also be used between threads

```
sem_t *sem;
sem = sem_open (name, flags, mode, init_val); /* name is system-wide       */
status = sem_close (sem);                     /* semaphore still reachable */
status = sem_unlink (name);                   /* now it is removed         */

status = sem_init (sem, pshared, init_val);  /* sem must be defined, e.g. through shm */
status = sem_destroy (sem);                  /* pshared: whether multiple processes
                                               * access sem; should be true */
```

28

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Semaphore operations

- Basic interface, designed for speed
  - no nice error recovery handles, e.g. in case of destruction during use
- Obtaining the value is tricky
  - value is unstable
  - negative value: interpret as number of waiters (length of queue)
- Queueing discipline
  - must be priority based

```
status = sem_wait (sem);
status = sem_trywait (sem);          /* returns error (EBUSY?) if sem == 0        */
status = sem_post (sem);
status = sem_getvalue (sem, &val);  /* current value
                                      * when negative: absolute value = # waiters */
```

29

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Synchronization among threads

- Can use POSIX 1003.1b primitives
  - e.g. across process boundaries
  - but these are relatively heavy
- Special, two-state semaphore: *mutex*
  - specifically for *mut*ual *ex*clusion
  - don’t use copies of a mutex
  - *lock()* and *unlock()* always by same thread (“ownership”)

```
pthread mutex_t m = PTHREAD_MUTEX_INITIALIZER;
                              /* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL; should return 0     */
status = pthread_mutex_destroy (&m);  /* should return 0                   */
status = pthread_mutex_lock (&m);     /* should return 0                   */
status = pthread_mutex_trylock (&m);  /* returns EBUSY if m is locked      */
status = pthread_mutex_unlock (&m);   /* should return 0                   */
```

30

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Priority inversion

- Mutexes solve the inversion problem through their attributes
  - selection of concurrency control policy
    - inheritance and ceiling (highest locker) protocols

```
attr = pthread_mutexattr_t;
status = pthread_mutex_setprotocol (&attr, protocol);
status = pthread_mutex_getprotocol (&attr, &protocol);
/* protocol can be: PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT */

/* in case of PTHREAD_PRIO_PROTECT: */
status = pthread_mutex_getprioceiling (&mutex, &ceiling);
status = pthread_mutex_setprioceiling (&mutex, ceiling);
status = pthread_mutexattr_getprioceiling (&attr, &ceiling);
status = pthread_mutexattr_setprioceiling (&attr, ceiling);

status = pthread_mutex_init (&m, &attr);
```


Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Condition variables

- Need exclusion on the variables, e.g., maintain $x \geq 0$

```
var cv: condition; m: Semaphore (initially 1)

|[ ......                                  |[ .....
  P(m);                                      P(m);
  while x<10 do                              x := x+100;
    V(m); { (*) } Wait (cv); P(m)   ||       Sigall (cv);
  od;                                        V(m);
  { x≥10 }                                   .....
  x := x-10;                               ]|
  V(m);
  ......
]|
```

- Point (*) represents a place where the signal may get lost
  - need to combine *V(m); Wait(cv); P(m)* atomically

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Condition synchronization

- Structure program in condition critical regions
  - truth of condition necessary for executing the section
    - protecting invariant
    - or avoid busy waiting
  - decide carefully which variables must be protected together
  - access to these only within critical sections
    - reads in other parts are harmless but unstable

- Condition variables
  - extended with timeout mechanism
  - with associated semaphore
    - can have several condition variables per semaphore for more accurate signalling
  - must only be signalled within critical sections

Condition variables

- Usage:
  - lock
  - while not condition do wait od;
  - critical section
  - possible signals
  - unlock

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
status = pthread_cond_init (&cond, attr);          /* should return 0          */
status = pthread_cond_destroy (&cond);             /* idem                     */
status = pthread_cond_wait (&cond, m);             /* semaphore m is associated with
                                                    * all critical sections    */
status = pthread_cond_timedwait (&cond, m, exp); /*  exp: max. waiting time; returns
                                                    * ETIMEDOUT after exp.*/
status = pthread_cond_signal (&cond);              /* one waiter               */
status = pthread_cond_broadcast (&cond);           /* all waiters              */
```


Additional attributes

- Both condition variables and mutexes can be created in shared memory between processes
  - admits communication between threads in two processes

Signals (1003.1)

- Software exception/interrupt/trap for
  - event notification (even synchronization)
    - timer, message arrival, error conditions, ...
  - poor man's concurrency
- Asynchronous
  - interrupt current execution (endanger predictability)
- Handler function
  - invoked upon receipt of signal
- Masking
  - to block specific signals temporarily
- Performance & use
  - fairly bad performance, in general, but can be optimized
    - signal delivery is a complicated operation
  - too few distinct types, signals carry no data, there is no precendence and signals may be lost

Real-time signals (1003.1b)

- Real-time signals extend 1003.1:
  - flexible signal identification
    - many signals, signals carry data
    - can be used as an asynchronous message passing facility
  - signals can be queued and have precedence
    - lowest number first
  - support for efficient signal catching
- Issues in using signals:
  - setting up a handler for a particular signal: *sigaction()*
  - sending a signal: *sigqueue()*
  - setting up an event (e.g. timer) that generates a signal: *sigevent* structure
  - waiting for signals to come in: *sigwaitinfo()*
    - more efficient than using handler functions

Threads and signalling

- Signals are process-wide
  - a ‘kill’ or ‘stop’ affects all threads
- Signals can be masked, per thread
- It is undetermined which thread will receive a signal
  - When two threads wait on a signal, only one will receive it
- Error signals go to the thread that caused it

- In general: avoid using signals and threads together

Contents

On time

- Scheduling
  - priorities
    - also in relation to blocking
  - scheduling disciplines
  - rules
    - immediate pre-emption
    - obligatory yielding
- Clocks
  - resolution, drift
  - delaying & interval timing

Scheduling

- Set and get scheduling parameters:
  - priority
  - discipline
    - SCHED_FIFO: run highest priority process until it blocks, gets pre-empted or stops
    - SCHED_RR: run collection highest priority round robin (timesliced)

```
struct sched_param {
    .... int sched_priority; ...
};
struct sched_param sp;

sp.sched_priority = .......;
status = sched_setscheduler (procid, policy, &sp);
    /* procid: id of process to set the scheduler for, e.g. myself: getpid()
     * policy: SCHED_FIFO, SCHED_RR or SCHED_OTHER */
policy = sched_getscheduler (procid);
```


Finding out about scheduling

- Each policy has a range of admitted priorities
- A process can retrieve scheduling information about other processes
- Changing the priority will
  - always put a process at the end of its queue
  - pre-empt the process immediately when a higher priority process becomes eligible

```
minpri = sched_get_priority_min (policy);
maxpri = sched_get_priority_max (policy);

status = sched_getparam (procid, &sp);
status = sched_setparam (procid, &sp);

sched_yield(); /* go to end-of-queue */
```

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Threads and scheduling

- Real-time scheduling similar as for processes
  - two policies, priorities from a range per policy, etc
  - functions with a *pthread_...* prefix
- Choices
  - contention scope: whether scheduling is limited to the process or is system-wide
    - system contention scope: predictable
      - threads become kernel entities
    - process contention scope: cheap

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Clocks and timers

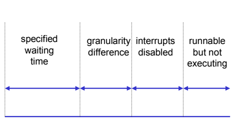
- *“What, then, is time? If no one asks me, I know what it is. If I wish to explain to him who asks me, I do not know”* (St.Augustine)
- Real-time is linear, transitive, irreflexive and dense
  - though discrete in computer systems
- Measuring time:
  - access directly the environments’ time (“share it”) or
  - approximate it with an internal clock
- POSIX 1003.1b
  - has clocks of type *clockid_t*
  - at least one clock named CLOCK_REALTIME
  - ...with minimal resolution 50Hz (actual: *clock_getres()*)

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Absolute and relative delay

- Relative: delay execution for a certain amount of time
  - *nanosleep()*, delays with respect to CLOCK_REALTIME
- Drift: inaccuracy in delaying
  - avoid cumulative drift
  - delay should be significantly longer than clock resolution

Drift




```
struct timespec {
    time_t tv_sec; /* # seconds since ... */
    long tv_nsec;  /* # nanoseconds      */
};

....
rtn = nanosleep (&request_time, &remaining);
/* rtn<0: sleep interrupted; delay remaining */
```

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Absolute and relative delay (cnt'd)

- Absolute: delay until a point in real time
  - repetitive or once
- POSIX 1003.1b: interval timers
  - created with respect to a clock
  - deliver specified (real-time) signal upon timer elapse
    - signal handling added to drift
  - also collect #missed signals

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

POSIX 1003b timers

```
struct itimerspec {
    struct timespec it_value;    /* interpreted by timer_settime() */
    struct timespec it_interval; /* interval for repetition, 0: once */
};

rtn = timer_create (CLOCK_REALTIME, &event_description, &my_timer);
/* 2nd argument NULL: SIGALRM is delivered */

rtn = timer_settime (my_timer, TIMER_ABSTIME, &new, &old);
/* 2nd argument: flag, determines absolute or relative time */

rtn = timer_gettime (my_timer, &time_remaining);

rtn = timer_getoverrun (my_timer); /* # missed signals */
```


Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Contents

- OS architecture
- Standardization: POSIX
  - introduction
  - processes & threads
  - synchronization
  - timing & scheduling
  - message passing

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Message passing

- Streaming
  - sequence of bytes transferred
  - no coupling between send and receive operations
    - internal structure not visible in primitives
    - only rudimentary support for control information
- Message queues
  - sequence of messages
  - send and receive operations are coupled

49

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Pipes & Fifo’s (1003.1)

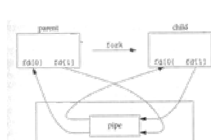
- Pipe:
  - connected pair of file(-descriptor)s
  - created before creation of child process
    - first descriptor for reading
    - second descriptior for writing
    - .....limited topology
  - buffered, one-directional *streams*
    - no message structure imposed
  - kernel entity with limitations
    - no discrimination in importance
    - no control over (kernel) buffer
    - no information on current state
    - limited number available

50

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Pipes & Fifo’s (cnt’d)

```
int fd[2];
pid_t child;

if (pipe (fd)<0) perror (“pipe”);

child = fork();
if (child<0) perror (“fork”);

if (child != 0) /* parent: writer */
    close (fd[0]);
    write (fd[1], ......);
else{ /* child: reader */
    close (fd[1]);
    read (fd[0], ......);
}
```



51

Pipes & Fifo's (cnt'd)

- Fifo ("named pipe"):
  - same as a pipe, but connection setup via namespace (i.e., filesystem)
  - create a unique name, e.g., "/tmp/fifo", using
    - *mkfifo ("/tmp/fifo", mode);*
  - Now use "/tmp/fifo" as a file: *open(), read(), write()*
  - Behavior and limitations similar as pipes

Message queues (1003.1b)

- A named priority queue of messages
  - *named*: more general use than parent-child relationships
  - *priority*: support discrimination based on importance
  - *messages*: structure is maintained: a *receive()* is coupled with a *send()*
- Fresh namespace
  - for portability,
    - start names with '/'
    - do not use any subsequent '/'
- Kernel entity
  - data is copied

```
struct mq_attr attr;
mqd_t mq;

attr.mq_maxmsg = 100; attr.mq_msgsize = 128;
attr.mq_flags = 0;
mq = mq_open("/MyQ", how, mode, &attr);

if (mq == (mqd_t)-1) perror ("mq_open");

/* how: e.g. O_CREAT | O_RDWR
 * mode: e.g. S_IRUSR | S_IWUSR | S_IXUSR */
```


Message queues (cnt'd)

- Manipulation through
  - *mq_setattr(), mq_getattr()*
    - queue size, message size – set upon creation only
    - current number of messages, (un)blocking use of communication functions
  - *mq_close(), mq_unlink()*
    - unlinking possible only after closing by all partners
- Communication:
  - *status = mq_send(mq, "hello world", 12, prio)*
    - *prio:* between 0 and MQ_PRIO_MAX (≥ 32)
  - *nbytes = mq_receive(mq, buf, maxsize, &prio)*
    - (oldest, highest prio) first

Message queues (cnt'd)

- Notification
  - upon transition from empty to non-empty
  - ...of precisely one process
  - ...through a real-time signal
- Message queues may be clogging the system
  - be careful to *unlink()*, e.g., after starting your system
  - there are doubts on their usefulness because of this
- Good for use in a distributed environment

Priority inversion and message passing

- Programs can be written using just message passing
  - modular, easier to distribute, less prone to errors
- Blocking on a resource is translated into
  - waiting until a message is accepted
  - or until a message is received
- Priority inversion here possible as well
  - inheritance: execute at highest priority of any waiter on communication
  - realization: attach this priority to messages
  - needs support of kernel / message passing system to actually *set* the priority – not in 1003.1b
- Message queues don't have this
  - but it is probably possible to implement it

Exercises

- **P.1** Given is a collection of threads that modify shared variable *s* through statements of the form

  $$s := s+E$$

  where *E* is the outcome of an expression. It is given that *s* is initially 0. The program must maintain: $s \geq 0$

  a. Make a data structure containing both *s* and the required variables for exclusion, and make conditional critical sections for the above statements.

  b. Make a function to update *s* so that the above statements can be replaced by a call to this function.
- **P.2** Can you give an implementation of general semaphores using condition variables?

Literature

- POSIX.4, Programming for the Real World, B.O. Gallmeister, O'Reilly & Associates INC, 1995, ISBN 1-56592-074-0
- Programming with POSIX threads, D.R. Butenhof, Addison-Wesley professional computing series, 1997, ISBN 0-201-63392-2
- Real-time Systems and Programming Languages, Ada 95, Real-Time Java and Real-Time POSIX, A. Burns and A. Wellings, Addison-Wesley, 2002, ISBN 0-201-72988-1