RTOS VxWorks 7.x

Ing. Michal Sojka, Ing. Zdeněk Šebek

October 9, 2024

Czech Technical University in Prague,
Faculty of Electrical Engineering,
Department of Control Engineering

Topics

- VxWorks 7.x kernel – components, properties
    - Latest VxWorks versions are no longer numbered as 7.x, but by release date: 22.06 (June 2022)
    - This lecture applies to these versions too.
- Device drivers
- Kernel customization to a specific application

VxWorks – basic properties I.

- UNIX type real-time operating system
- Proprietary, WindRiver
- Safety certification (aerospace etc.)!

- Real-Time scheduling
  - Unlimited number of tasks
  - Preemptive scheduling
    - Priority-Based
    - Round-Robin
  - 256 priority levels
- Fast and flexible inter-process communication

VxWorks – basic properties II.

- Inter-task/process communication
  - Binary, counting and mutex semaphores
    - Supports priority inheritance
  - Message queues, Signals, Pipes, Sockets
  - Shared memory
- Connectivity
  - IPv4, IPv6, Time-sensitive networking (TSN), USB, CAN

VxWorks – basic properties III.

- Multi-core support (symmetric/asymmetric)
- Virtualization
- Asynchronous I/O
- Filesystems
  - FAT file system
  - „raw“ file system
  - TrueFFS (for flash memories)
  - Fault-tolerant file system HRFS
- Multimedia: OpenGL (ES). OpenCV, Vulkan
- Security: OpenSSL, Secure boot, Arm TrustZone...

VxWorks – supported CPU architectures

- Intel x86 (32b., 64b.)
- ARM (32b., 64b.)
- RISC-V
- MIPS
- PowerPC
- …

VxWorks – application programming interfaces (API)

- How programs interact with the OS
- Two main options:
    - Wind API
    - POSIX API

VxWorks 7.x – Languages & Wind API

- C language
  - Why? C is a portable assembler.
- Rust for user space (RTP)

- Wind API
  - Basic API of VxWorks
  - Is not POSIX compatible
  - Less complicated
  - Usually solves drawbacks of POSIX specification
  - Using this API produces less portable code

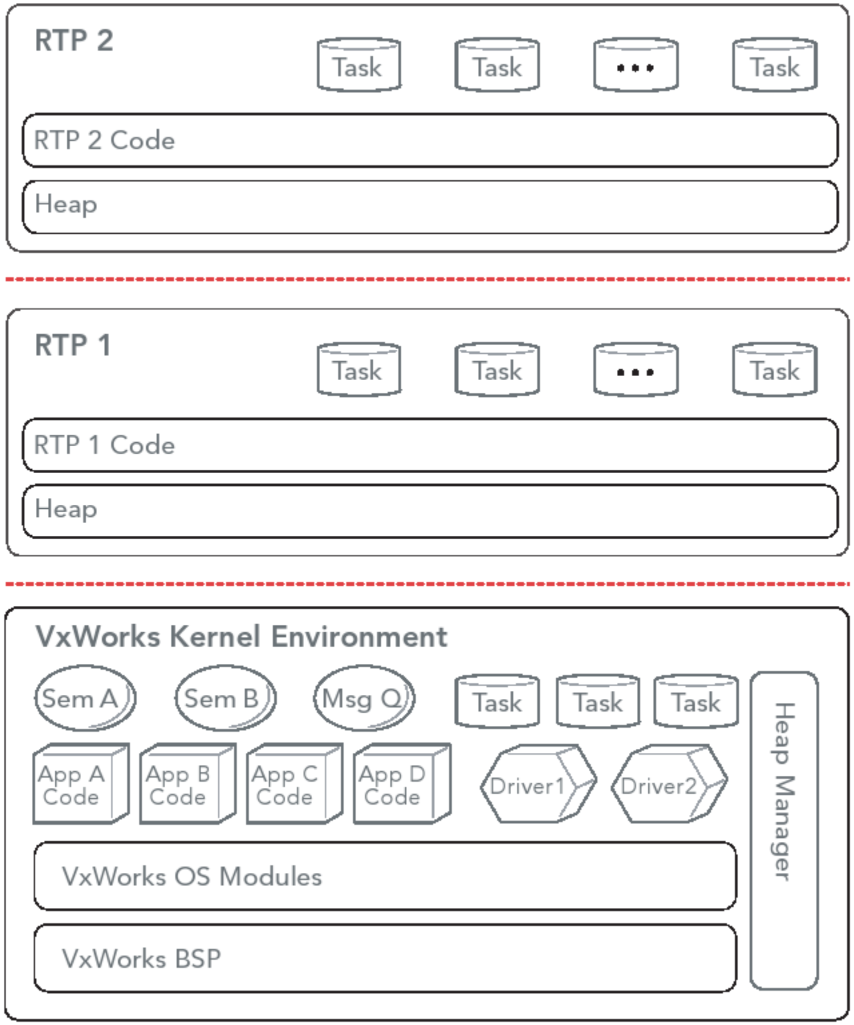
VxWorks 7.x – POSIX API

- Standard API compatible with POSIX 1003.1b specification for
  - Asynchronous I/O
  - Semaphores
  - Message queues
  - Memory management
  - Signals
  - Scheduler
  - Timers

Applications types

- Downloadable kernel module (DKM)
  - No memory protection
  - Direct access to HW
- Real-time process (RTP)
  - Employs memory protection
  - No direct access to HW
- DKM is similar to Linux kernel modules (drivers)
- WindRiver tries to provide the **same (similar) APIs** for both DKM and RTP applications.

Overall VxWorks OS Structure



Task Management I.

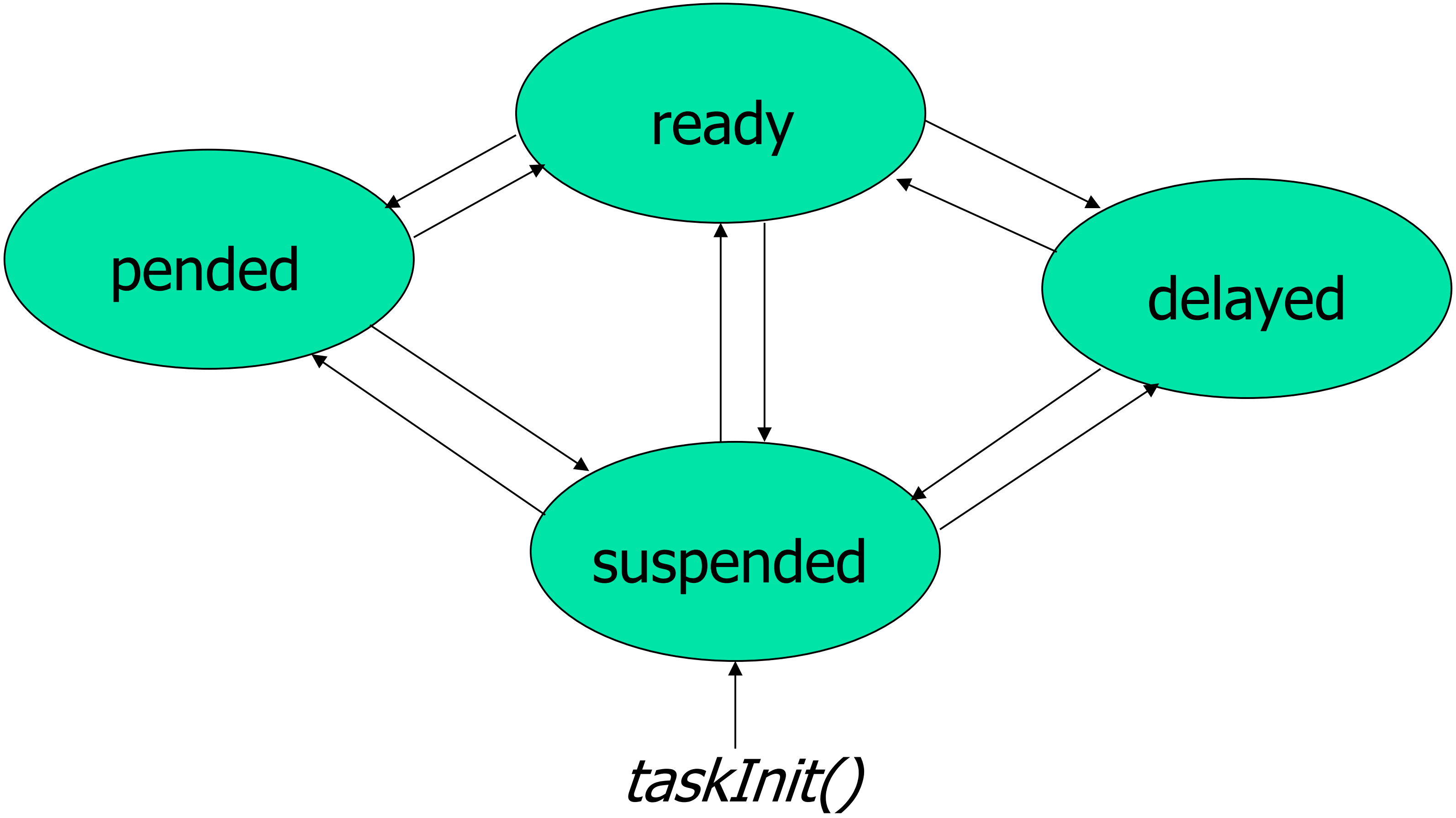
Task context a.k.a. task_struct (Linux)

- Program counter
- Content of CPU registers
- Stack
- Assignment of standard I/O
- Timer for function *delay*
- Timeslice timer
- Kernel control structures
- Signal handlers
- Debugging a monitoring variables

Task management II.

- All tasks run in one common address space (either kernel or RTP)
  - + Fast context switch
  - − Zero protection
- Besides other things, RTP implements protection mechanisms (if CPU has MMU)

Task state



READY state

- The task is ready to run
- Doesn't wait for any resources except for CPU
- VxWorks doesn't distinguish whether the task is running (has assigned CPU) or not.

PEND state

- Task is blocked, waits for some resource to be assigned to it.
- Typical examples are waiting for a semaphore, reading from an empty message queue etc.
- Most of the time caused by calling ***semTake***, ***msgQReceive*** etc.

DELAY state

- The task waits for some time interval to elapse
- Caused by calling taskDelay() or nanosleep()
- Warning! This is different from elapsing of timeout in some calls.

SUSPEND state

- The execution of the task is forbidden
- Typically used when the task is debugged
- Doesn't forbid change of task state, only its execution
- This state can be set by calling ***taskSuspend***

STOP state

- also used by debugger
- signalizes the task was stopped by a breakpoint

Task State – Combinations I.

- DELAY+S
  Simultaneously delayed and suspended, e.g. call to ***taskDelay*** during debugging
- PEND+S
  Simultaneously pended and suspended e.g. waiting for a semaphore (***semTake***) during debugging

Tasks state – combinations II.

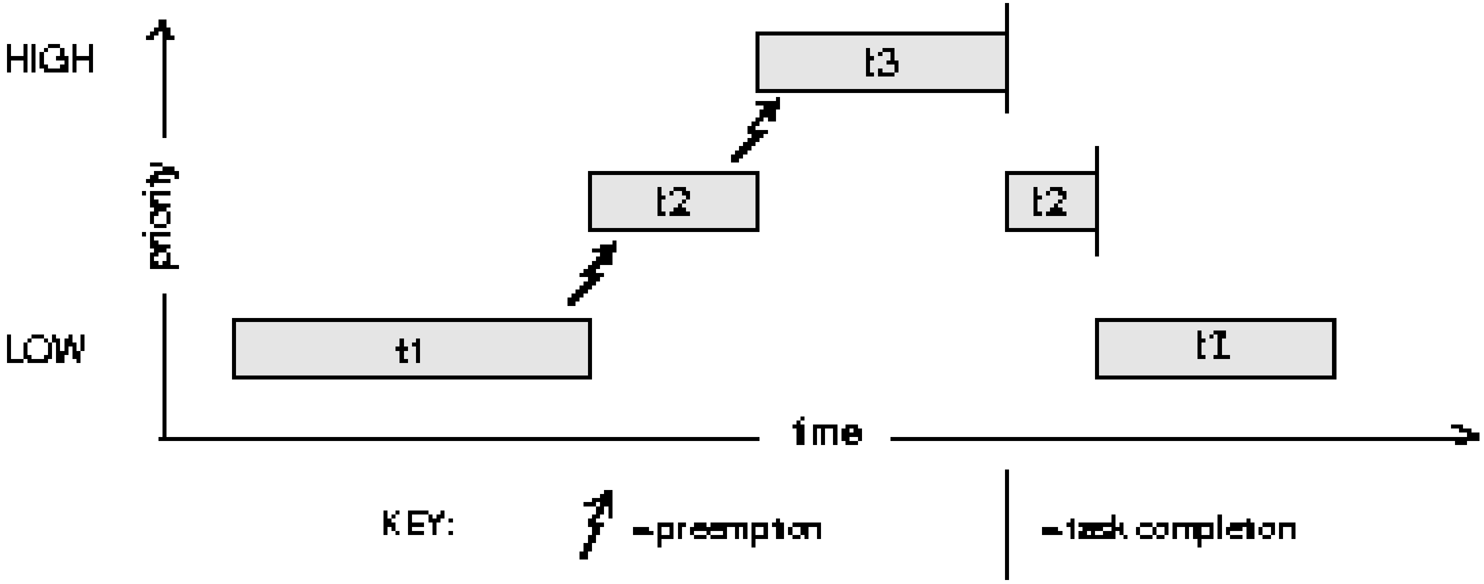
- PEND+T
  waiting for a resource with timeout
- PEND+T+S
  same as PEND+T, but suspend because of debugging
- State+I
  arbitrary state, priority inheritance mechanism is active

Task priorities

- Tasks have priorities in range 0 (highest) through 255 (lowest)
- Priority can be read or set at runtime (***taskPriorityGet***, ***taskPrioritySet***)
- When creating the task manually (debugger, shell) the priority is set to the default value 100
- Recommended priority ranges:
  - Applications: 100 – 255
  - Drivers: 51 – 99
  - Network handling (tNet0): 50

Preemptive fixed-priority scheduling

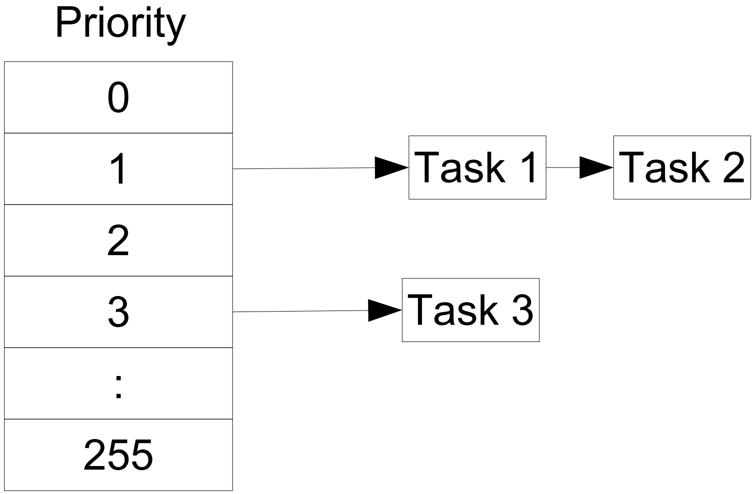
- Default scheduler – selects which task will be run next
- Reflects only task priorities (difference from GPOS, needs to be deterministic)



How the scheduler selects the next task to run?

- What is the complexity of the scheduling algorithm?
  - O(n·log n), O(n), O(log n), O(1)?
    n is the nuber of tasks in the system

Run queue data structure



Priority

0

1 → Task 1 → Task 2

2

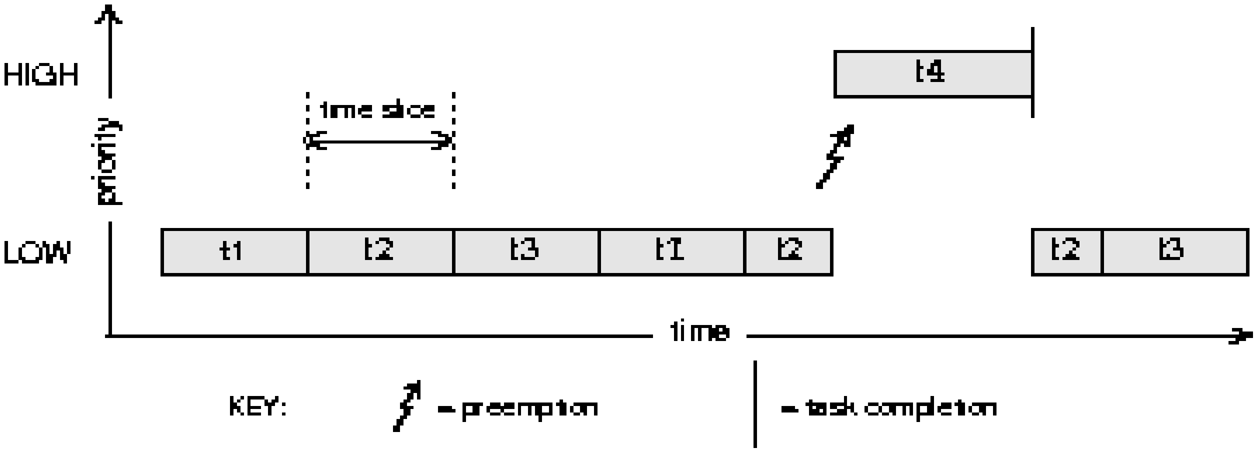
3 → Task 3

:

255

Round-Robin Scheduling

- Limits time (*timeslice*), when the CPU is assigned to one task, then rescheduling to different one is forced.
- *Timeslice* can be set by system call ***kernelTimeSlice()***
- Task priority remains the main criterion .



Scheduler invocation

- When is the scheduler executed?
  - After every interrupt – there might be new work to do
    - Timer (system tick)
    - I/O device
  - As a part of some system calls
    - taskDelay
    - semTake, semGive
    - ... and many more
- What exactly is the context switch?

Disabling of Scheduling

- Every task can disable/enable rescheduling to other task using ***taskLock***/***taskUnlock*** *calls*
- In locked state, rescheduling is possible only if the task is blocked (PEND state) or suspended
- Interrupts are not blocked at all

- What is it good for?
- What is better for real-time? Using taskLock() or mutexes?

Task creation

- ***taskInit*** – create a task
- ***taskActivate*** – run a task
- ***taskSpawn*** = ***taskInit*** + ***taskActivate***
  Creates and runs the task according to the parameters:
  - Task name
  - Stack size (why this needs to be specified?)
  - Code (entry function)
  - Entry function parameters

Task Creation Options

- **VX_FP_TASK** – must be specified when the task uses floating-point operations. Why?
- VX_NO_STACK_PROTECT – Create without stack overflow or underflow guard zones.
- VX_TASK_NOACTIVATE – Used with taskOpen( ) so that the task is not activated.
- VX_NO_STACK_FILL – Does not fill the stack with 0xEE.
  - Filling stacks is useful during development for debugging with the checkStack( ) routine.

Task termination

- Task is terminated when either
  - The entry function returns or
  - ***taskDelete(taskId)*** *is called*
- Enabling/disablig task deletion – ***taskSafe***/***taskUnsafe*** *calls*
- If the task is in Safe state, other tasks calling ***taskDelete*** on the task are blocked.
- Beware: deleted task does not release held locks (mutexes)

Task control

- ***taskSuspend*/*taskResume*** – suspends/resumes task
- ***taskRestart*** – recreates the task with the original creation arguments
- ***taskDelay*** – delays the execution for specified time. Time is measured in **ticks of the system timer** (default frequency is 60 Hz, can be changed/read by ***sysClkRateSet*/*sysClkRateGet***)
- POSIX
  - ***nanosleep*** – delay, time in nanoseconds

Tasks in POSIX = Threads

- pthread library
- pthread_create() – no two phase initialization
- pthread_cancel()
- Thread cancellation (see POSIX:2008 2.9.5)

Scheduler – POSIX API

- **POSIX priority numbering is inverse to VxWorks**
- POSIX allows setting the scheduling algorithm independently for each task
- Lowest and higher priority level is not defined
- VxWorks supports only one algorithm for all tasks in the system

Scheduler – POSIX API (1)

```
/* Header file */
#include <sched.h>
```

```
/* Constants */
```

- **SCHED_FIFO** – Preemtive priority-based scheduling
- **SCHED_RR** – Round-robin scheduling
- **SCHED_OTHER** – Other, implementation dependent scheduling
- **SCHED_SPORADIC** – Sporadic server scheduling

```
/* Get/set scheduling algorithm */
int sched_getscheduler(pid_t pid);
int sched_setscheduler(pid_t pid, int policy, struct
   sched_param *params);
```

Scheduler – API (2)

```
/* Get and set scheduling parameters */
int sched_getparam(pid_t pid, struct sched_param
   *params);
int sched_setparam(pid_t pid, struct sched_param
   *params);
int sched_rr_getinterval(pid_t pid, struct
   timespec *t);
/* Explicitly execute rescheduling */
int sched_yield(void);
/* Get minimal and maximal priority applicable to
   a given scheduler */
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Inter-task/Inter-process Communication (IPC)

- shared memory
- semaphores
- message queues and pipes
- sockets
- signals
- events

Shared memory

- All tasks (threads) in a multi-threaded program share memory.
- Tasks can communicate by writing and reading to the memory.
- Shared memory is the fastest IPC mechanism – there is no software-induced overhead.
- It might not be as easy to use as it seems...

Memory consistency

- When data are accessed/modified from multiple places (e.g. tasks), extra care has to be taken.
- We don’t want tasks to randomly overwrite data used by other tasks.
  - This type of programming error is known as a “race condition”
  - Race conditions are very hard to debug!
  - Race conditions are not deterministic – typically they happen only from time to time, e.g. once per week
- Solution: synchronize the tasks somehow

Maintaining data consistency

- If shared data is accessed from:
  - multiple tasks => **mutexes**
  - Tasks and interrupts => **disable interrupts**
  - Interrupts on multiple processors (SMP) => **spinlock**
- Other methods (scalable in SMP)
  - Non-blocking synchronization (atomic instructions)
  - Per-CPU variables
  - Read-Copy-Update (RCU, SMP)
    - Details are out of scope of this lecture
    - Covered in Efficient Software course (B4M36ESW)

Semaphores

- Basic synchronization mechanism
- Internal variable has the value 0 or 1 (binary, mutex semaphore) or arbitrary non-negative integer (counting semaphore)
- Two primitives for accessing semaphore
  - **semTake** – takes the semaphore (internal variable is decremented), if the semaphore is not available (variable = 0), calling task is blocked (PEND state)
  - **semGive** – „returns“ the semaphore (increments the internal variable and optionally wakes a waiting task up)

Simple semaphore implementation (on uniprocessor, buggy!!!)

```
struct task *current;

struct Sem {
  int count;
  struct task *queue;
};
```

```
void semTake(sem) {
  if (sem->count > 0) {
    sem->count--;
    return;
  }
  current->state = PEND;
  runq_del(curent)
  listAppend(sem->queue, current);
  schedule(); // select new current
}
```

```
void semGive(sem) {
  waiting = listDelHead(sem->queue);
  if (waiting) {
    waiting->state = READY;
    runq_add(waiting);
    schedule();
  } else {
    sem->count++;
  }
}
```

Simple semaphore implementation (on uniprocessor)

```
struct Sem {
  int count;
  struct task *queue;
};
```

```
void semTake(sem) {
  intLock();
  if (sem->count > 0) {
    sem->count--;
    intUnlock();
    return;
  }
  current->state = PEND;
  runq_del(curent)
  listAppend(sem->queue, current);
  intUnlock();
  schedule();  // select new current
}
```

```
void semGive(sem) {
  intLock();
  waiting = listDelHead(sem->queue);
  if (waiting) {
    waiting->state = READY;
    runq_add(waiting);
    intUnlock();
    schedule();
  } else {
    sem->count++;
    intUnlock();
  }
}
```

Semaphores – API I.

Semaphore Creation

semBCreate(int options, SEM_B_STATE initialState)

semCCreate(int options, int initialCount)

semMCreate(int options)

initialState: SEM_FULL (1), SEM_EMPTY (0)

initialCount: initial value of the internal variable

options: specifies how the tasks waiting for the semaphore are queued i.e. who will get the semaphore first after the semaphore is returned.

- **SEM_Q_FIFO** – according to the order in which tasks asked for the semaphore
- **SEM_Q_PRIORITY** – first according to the priority, then according to the order

Semaphores – API II.

Asking for (Locking) the Semaphore

```
STATUS semTake(SEM_ID semId,/*semaphore to take*/
                int timeout /*timeout in ticks*/)
timeout:      WAIT_NOWAIT (0) don't wait
              WAIT_FOREVER (-1)
              timeout v system clock ticks
```

Returning (Unlocking) the Semaphore

```
STATUS semGive ( SEM_ID semId)
```

Deleting the Semaphore

```
STATUS semDelete ( SEM_ID semId)
```

Use of Semaphores

- Mutual exclusion
  - The semaphore (called mutex) is initialized as full
  - A task wanting to access the resource takes it, uses the resource and gives the mutex back
  - The code in between is called **critical section**
  - Mutex has a concept of **owner** (this is needed to prevent priority inversion – see later)
- Synchronization (producer-consumer)
  - The semaphore is initialized as empty
  - A task trying to wait for an event tries to take the semaphore and gets blocked
  - Whenever the event (e.g. IRQ) occurs, the semaphore is “given” by semGive (e.g. in an interrupt handler)

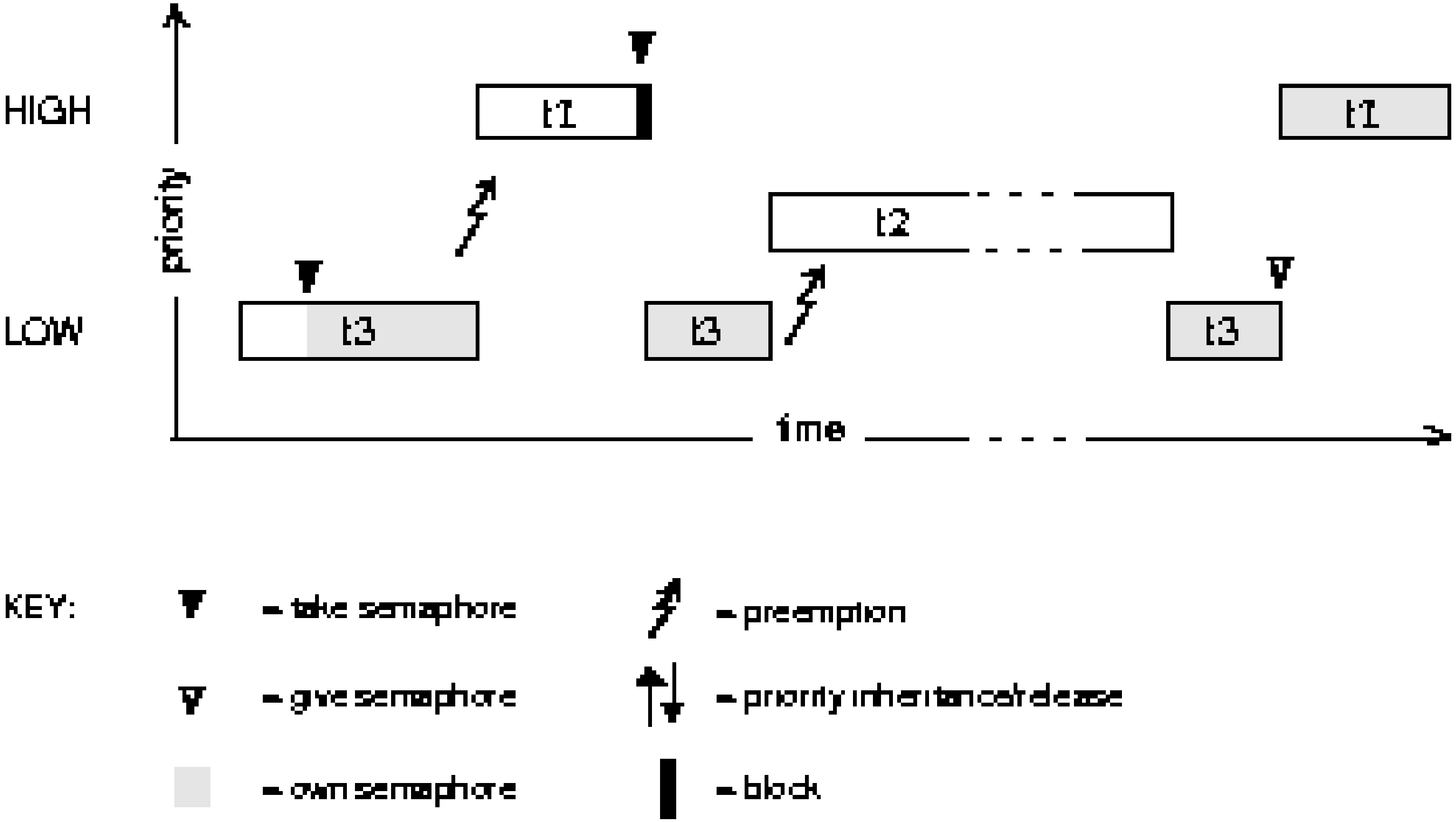
How does mutex protect things?

- Mutex can "protect" many things, e.g.
  - data structure
  - hardware device
- Association between the mutex and the thing it protects is just a **software abstraction**
  - Other tasks can access the data (without the mutex) even if another task has locked the mutex
  - It is often necessary to add comments to the code about what the mutex protects
  - Higher-level languages make this easier: monitors, synchronized methods in Java
  - Fine-grained locking

Options – mutex semaphore

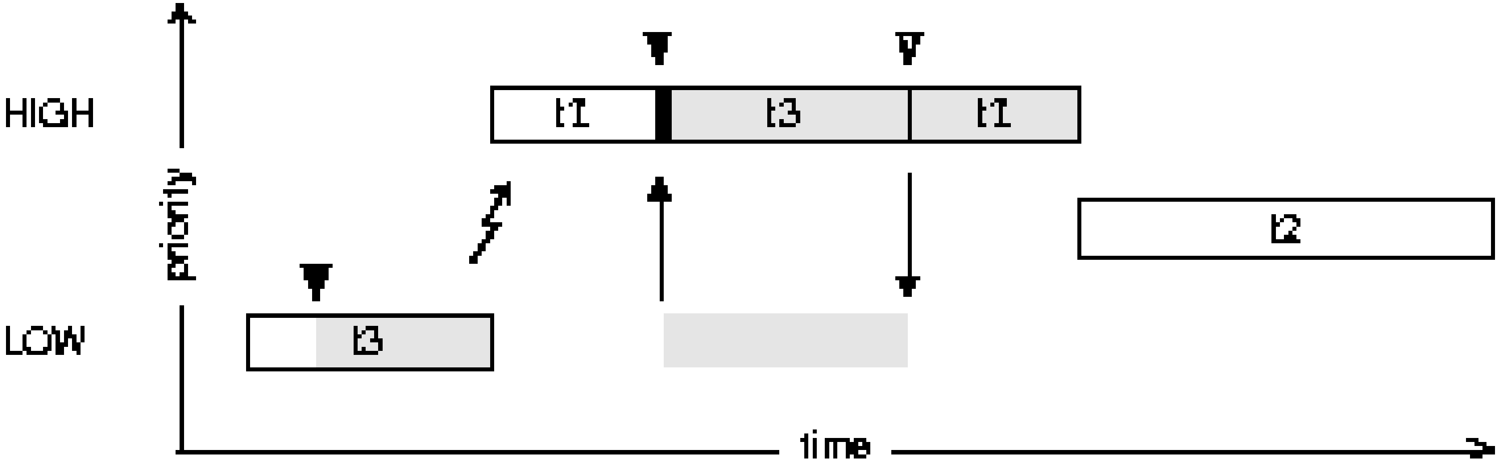
- **SEM_INVERSION_SAFE** – activates priority inheritance mechanism (priority inversion avoidance)
- **SEM_DELETE_SAFE** – it is not possible to delete the task owning this semaphore (corresponds to **taskSafe**)
- **SEM_INTERRUPTIBLE** – waiting for the semaphore can be interrupted by a signal.

Priority Inversion Problem



Possible Solution – Priority Inheritance

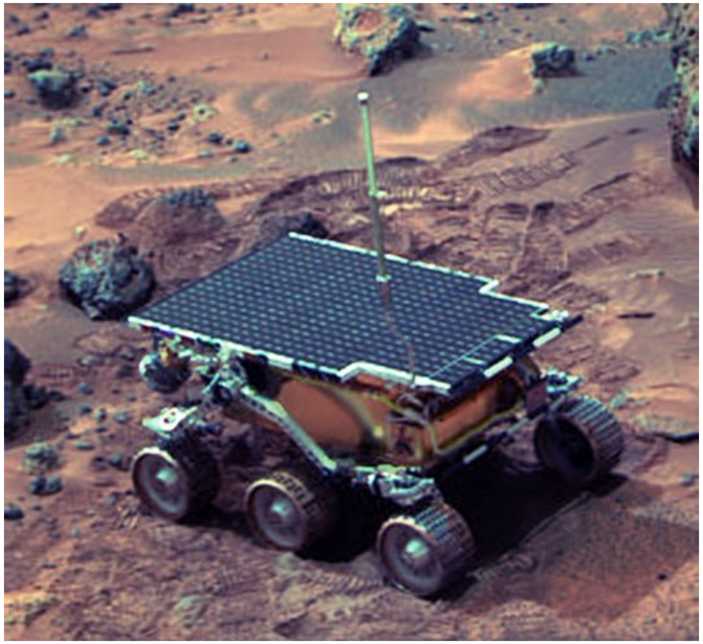
- The priority of tasks having “locked” some resource is temporarily increased to the highest priority among tasks waiting for that resource.



- Inheritance must work across the locking chain (when multiple resources and tasks are involved)

Mars Pathfinder & priority inversion

- Mars Pathfinder began experiencing total system resets
- One task missed a deadline and safety software caused the reset.
- A mutex without priority inheritance enabled was used inside the select() system call.
- It was sufficient to enable the priority inheritance by default.



- http://research.microsoft.com/~mbj/Mars_Pathfinder/

Recursive Use of Mutex Semaphore

- One task can lock the mutex repeatedly even if it is already locked by the same task.
- What is it good for?
- The number of semTake calls has to be the same as the number of semGive calls
- Mutex semaphore can be only returned by the task, which has locked the mutex.

Semaphores – POSIX API I.

- POSIX semaphore is always counting
- Can have a name (for sharing between processes/address spaces)

Semaphores – API (1)

```
/* Header file */
#include <semaphore.h>
```

```
/* Useful constants */
```

- **SEM_VALUE_MAX** – maximal available value of a semaphore (>= 32767)

```
/* Useful constants, named variant */
```

- **O_RDONLY, O_WRONLY, O_RDWR** – see. message queues
- **O_CREAT, O_EXCL** – see. message queues

Semaphores – API (2)

```
/* Create/destroy memory-based (unnamed)
  semaphore*/
int sem_init(sem_t *sema, int pshared, unsigned
  int initial_value);
int sem_destroy(sem_t *sema);

/* Connect to/open, close, delete named semaphore
  */
sem_t sem_open(const char *sem_name, int oflag,
  mode_t creat_mode, unsigned int init_val);
int sem_close(sem_t *sema);
int sem_unlink(const char *sem_name);
```

Semaphores – API (3)

```
/* Semaphore operations common to named and unnamed variants */

/* Enter critical section – blocking/nonblocking variant */
int sem_wait(sem_t *sema);
int sem_trywait(sem_t *sema);
/* Leave critical section */
int sem_post(sem_t *sema);
/* Read the value of semaphore */
int sem_getvalue(sem_t *sema, int *value);
/* wait with an absolute timeout (only CLOCK_REALTIME) */
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Shared code, reentrancy

- Every part of the code can be called from any task within the current address space (RTP, kernel)
- Functions that can be safely called this way are called **reentrant**. Such functions either
  - do not use global variables or
  - protect global variables with mutexes.
- Almost all VxWorks API functions are reentrant (exceptions have two variants with and without **_r** suffix, e.g., strtok() and strtok_r()).
- When using legacy non-reentrant code, it is possible to use so called *task variables*

Task variable

- global variable with a separate copy for each task
- ***taskVarAdd(int *ptr)*** – global variable of the length 4 bytes is added to the task context.
  - Each task, which called this function have its own copy of this variable.
  - At every context switch, the content of every task variable is saved/restored to/from the task context.

Real-Time processes (RTP) I.

- Similar to processes in different OSes (Unix)
- Optimized for RT
- Each RTP contains one or more tasks (sometimes called threads in other OSes)
- RTP can be thought as an organizing unit that groups several tasks. RTP alone is not scheduled, only the tasks within RTP are scheduled.
- Each RTP has its own address space
- User application can also be run as a kernel module. In that case its tasks are not part of any RTP.

Real-Time processes (RTP) II.
(optimizations for real-time)

- Entire process is always loaded in memory (no swapping/page faults)
- New RTP is spawn in two phases.
  - 1st phase runs with the priority of the calling process
  - 2nd phase (load) is executed with the priority of the new process, i.e. lower-priority processes do not influence the task that created them.

RTP creation

- ***rtpSpawn** call*
  - *filename on filesystem*
  - *Initial task is created*
  - *Starts with main() function*

RTP Termination

- *main() function returns*
- When last task exits
- If any task in process calls ***exit()***
- *By calling **rtpDelete***

Shared memory between RTPs

- Part of the address space is shared between multiple processes (not within a single VxWorks RTP)
- Mostly implemented in HW (memory management unit)
  - OS only sets up page tables
- To maintain data consistency, exclusive access must be ensured by some means, e.g.:
  - disabling interrupts (***intLock***/***intUnlock***) – it works (only on one CPU), but is not good with respect to real-time behavior
  - disabling of rescheduling (*taskLock*/*taskUnlock*) – better, but still not good
  - binary or mutex semaphore – the best approach is most cases

Shared Memory – API (1)

```
/* Header file */
#include <sys/mman.h>
```

```
/* Useful constants */
```

- **`O_RDONLY, O_RDWR, O_CREAT, O_EXCL`** – see message queues
- **`O_TRUNC`** – truncate file to zero bytes (default)
- **`PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC`** – enable none / read / write / code execution in shared memory
- **`MAP_FIXED, MAP_SHARED, MAP_PRIVATE`** – map shared memory block to a given address / writes are visible by others / non-visible for others (copy on write – COW)

Shared Memory – API (2)

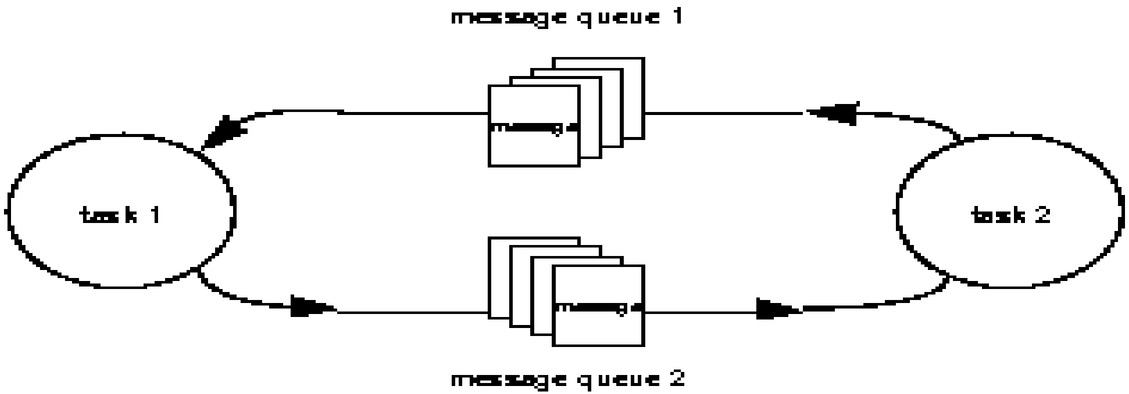
```
/* Create(open), close, delete named mapped
  memory */
int shm_open(char *name, int oflag, mode_t mode);
int close(int fd);
int shm_unlink(const char *name);

/* Set shared memory size */
int ftruncate(int fd, off_t total_size);

/* Map a file to the address space */
void * mmap(void *where_i_want_it, size_t length,
  int mem_protection, int map_flags,
  int fd, off_t offset_within_shared_mem);
```

Shared Memory – API (3)

```
/* Unmap the memory from the process address
   space */
int munmap(void *begin, size_t length);

/* Extension: change memory protection for a
   mapped memory (whole or only a portion) */
int mprotect(void *begin, size_t length, int
   mem_protecion);

/* Extension: synchronize the memory with mapped
   file (only for mmaped files) */
int msync(void *begin, size_t length, int flags);
```

Message Queues

- Transfer of messages of arbitrary length
  - The maximal length must by specified in advance
- FIFO
- One queue = one direction, for both directions two queues must be used



Message Queues – API

- ***msgQCreate*** – creation
- ***msgQSend*** – insert a message to the queue
- ***msgQRecv*** – get a message from the queue
- ***msgQDelete*** – delete queue and free used memory
- ***msgQNumMsgs*** – find out the number of messages in the queue

Message Queues – API II.

```
MSG_Q_ID msgQCreate( int maxMsgs,
                     int maxLen,
                     int options)
```

maxMsgs – max number of msg. in the queue

maxLen – max length of one message (Bytes)

options – MSG_Q_FIFO, MSG_Q_PRIORITY

how are ordered waiting tasks

Message Queues – API III.

```
STATUS msgQSend ( MSG_Q_ID msgQId,
                  char *buffer,
                  UINT nBytes,
                  int timeout,
                  int priority)
```

buffer, nBytes – data and its length

timeout – how long to wait for freeing the queue if it is full

priority – message priority (MSG_PRI_NORMAL, MSG_PRI_URGENT)

Message Queues – API IV.

```
int msgQReceive(MSG_Q_ID msgQId,
                char *buffer,
                UINT maxNBytes,
                int timeout)
```

buffer, maxNBytes – where to store received data. Longer messages will be truncated

timeout – how long to wait for getting something from an empty queue

Returns the length of the received message

Message Queues – POSIX API

mq_open – open named queue

mq_close – close it

mq_unlink – delete it

mq_send – insert message to the queue

mq_receive – get the message from the queue

mq_notify – ask for sending a signal when a message is inserted to the empty queue

mq_setattr/***mq_getattr*** – setting/getting of queue parameters

Message Queues – Wind/POSIX API Comparison

	Wind	POSIX
Number of priority levels	2	32
Ordering of waiting tasks	FIFO of priority	priority
Timeout waiting	yes	no
Notification by a signal	no	yes (one proces)

Pipes

- Message queue that looks like a file
- Created by calling ***pipeDevCreate***
- Then standard I/O operation (read, write) can be used
- Unlike msg. queue, pipe can be used in ***select*** call (waiting for multiple I/O events)

Signals

- Asynchronous events with respect to task execution
- Very similar to interrupts (generated by HW)
- Signals are generated by SW (OS or apps)
- When a signal is delivered, task execution is stopped and a signal handler is executed
- Bit-field in task_struct
- Two possible APIs:
  - UNIX-BSD
  - POSIX 1003.1 including queued signal extensions POSIX 1003.1b

Signals – BSD/POSIX API Comparison

POSIX	**BSD**	**funkce**
signal	*signal*	handler assignment
kill	*kill*	send signal to given process
raise	---	send signal to self
sigaction	*sigvec*	get/set handler
sigsuspend	*pause*	suspend process until a signal is delivered
sigpending	---	find out delivered signals blocked by mask
sigemptyset, sigfillset, sigaddset, sigismember, sigdelset, sigprocmask	*sigsetmask, sigblock*	signal mask manipulation

Signals – which ones to use

- The number of signals differs across platforms
- Some signals are used by the OS
- Availability and meaning of signals is different across platforms, see manual, *sigLib* library
- There are 7 signals starting with SIGRTMIN, for user application

Signals – multiple reception I.

- Handler executes with the priority of receiving task
- Problem: what happens when another signal is delivered before executing the handler of the same previously delivered signal?
- In that case the handler is executed **only once** (each signal is represented by one bit)
- Solution – queued signal extensions (POSIX 1003.1b)

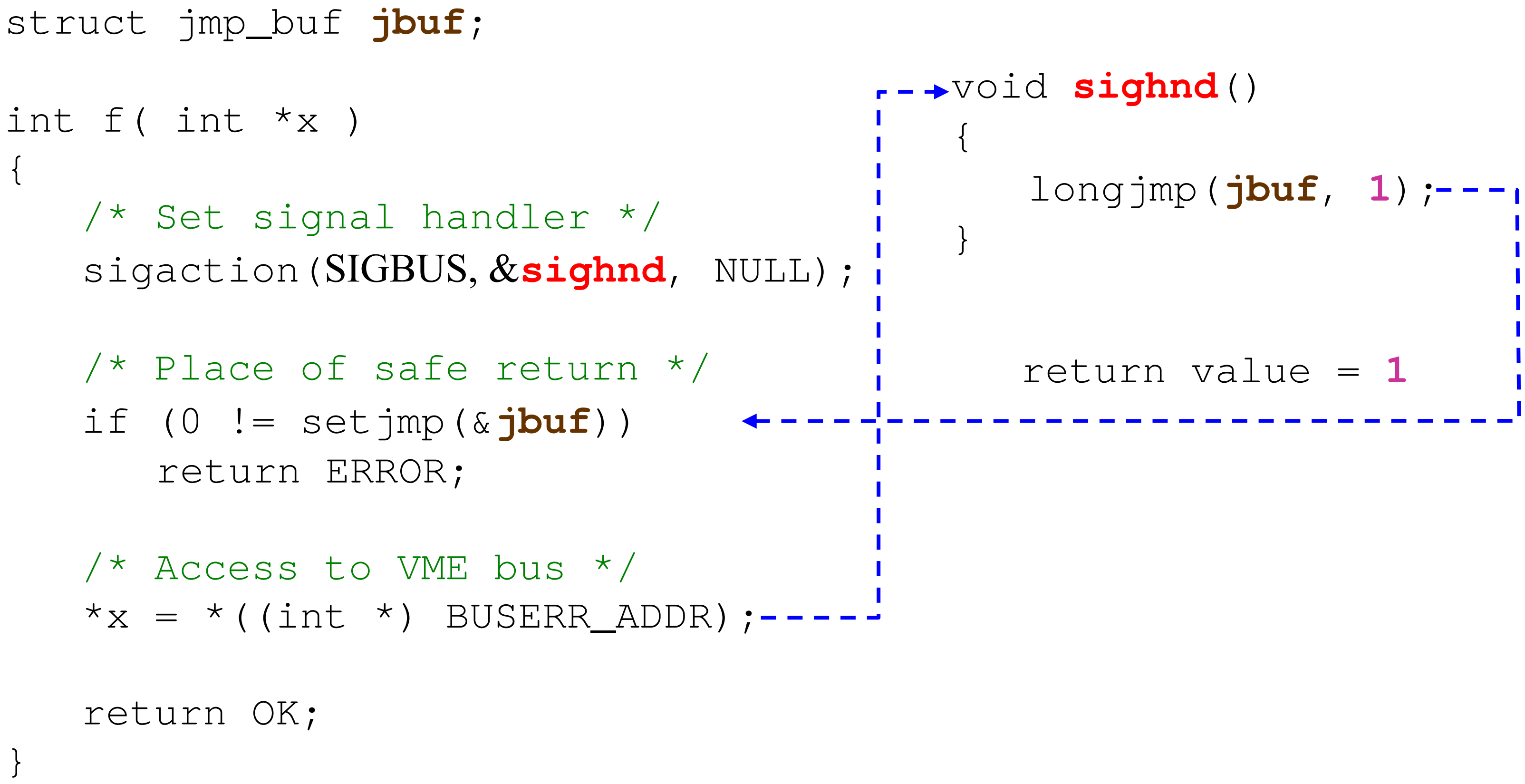
Signals – multiple reception II.

- Signal is sent by calling ***sigqueue***
- Sent signals are queued
- For each signal instance, the handler is executed
- It is possible to wait for signal (synchronous reception) without installing a handler – ***sigwaitinfo, sigtimedwait*** *calls*
- Queued signals can carry additional value specified by the user. The type of the value is pointer. Type casting can be used for other simple types.

POSIX 1003.1b realtime signals – API

```
/* Send a signal */
int sigqueue(pid_t victim_id, int sig, union
   sigval extra_info);

/* Wait for one or more signals */
int sigwaitinfo(const sigset_t *mask, siginfo_t
   *extra_info);
int sigtimedwait(... , const struct timespec
   *timeout);
```

Usage of signals and setjmp/longjmp for handling of bus error states during device detection



```
struct jmp_buf jbuf;

int f( int *x )
{
    /* Set signal handler */
    sigaction(SIGBUS, &sighnd, NULL);

    /* Place of safe return */
    if (0 != setjmp(&jbuf))
        return ERROR;

    /* Access to VME bus */
    *x = *((int *) BUSERR_ADDR);

    return OK;
}
```

```
void sighnd()
{
    longjmp(jbuf, 1);
}
```

- It is not possible to just set a global variable in the handler as the CPU would retry the bus access.
- Modern HW buses (PCI(e), USB, ...) allow to enumerate devices and this kind of device detection is not needed there.

VxWorks Events

- Lightweight task-to-task and ISR-to-task synchronization
- Notifications from message queues or semaphores
- Similar to signals – sent asynchronously, but received only synchronously
- 32 different events (25–32 are reserved to VxWorks)

Events API

- `eventSend(int taskId, UINT32 events)`
- `eventReceive(UINT32 events, UINT8 options, int timeout,`
- `            UINT32 *pEventsReceived)`
- `semEvStart(MSG_Q_ID msgQId, UINT32 events, UINT8 options)`
- `semEvStop()`
- `msgQEvStart()`
- `msgQEvStop()`

Static Instantiation of Kernel Objects

- Creation of kernel objects (tasks, semaphores, ...) requires memory allocation – slow, not always succeeds, ...
- It is possible to allocate the memory statically (required by many safety standards)

```
VX_TASK(myTask,4096);
int myTaskId;

STATUS initializeFunction (void)
{
myTaskId = VX_TASK_INITIALIZE(myTask, 100, 0, 4096, pEntry, \
                              0,1,2,3,4,5,6,7,8,9)
```

Timing

- taskDelay
- nanosleep
- POSIX timers
- Timestamp timers
- Watchdog timers

System tick

- In VxWorks, most of time-related operations are driven by the system tick
- Periodic timer interrupt (by default 50 or 60 Hz, depending on BSP)
  - Visible in system viewer
- Scheduler is invoked at least after every interrupt, including the system tick
- System clock frequency can be changed during runtime (**sysClkRateSet/Get**)
- When setting the system clock, return value must be checked. Too high frequency gives an error.

TaskDelay

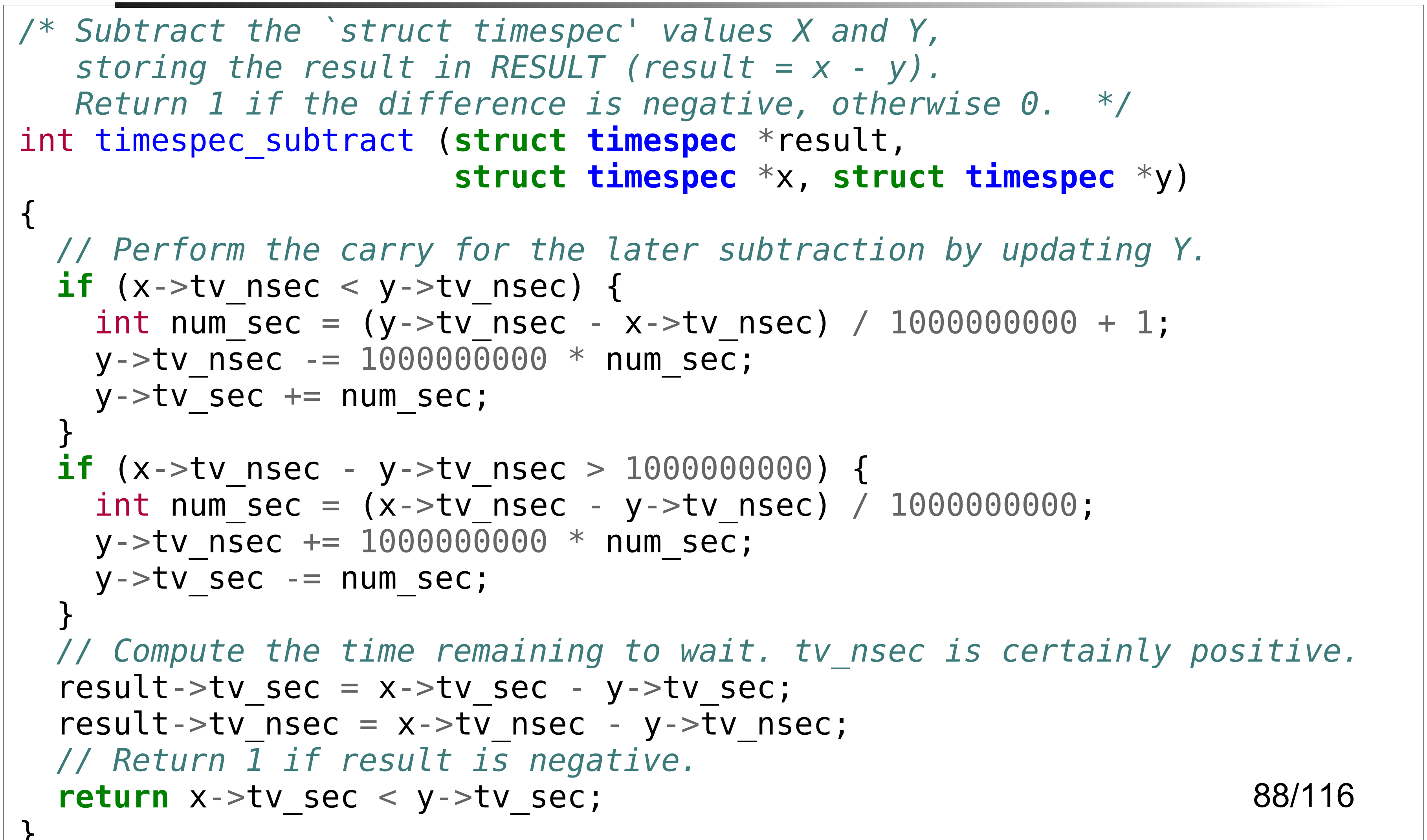
- Task execution is stopped for given number of system timer ticks
- taskDelay(0) only puts the task at the end of ready queue.
- taskDelay(1) waits for the end of the current tick
  - by default any time between 0 and 20 ms!
- Waiting is terminated when a signal is delivered to the delayed task

nanosleep

- Task execution is delayed for a given amount of time
- Time is specified in seconds and nanoseconds

  -
    ```
    struct timespec {
        time_t tv_sec;  /* seconds */
        long  tv_nsec;  /* nanoseconds */
    }
    ```
  - Arithmetic operation (±) are not trivial
- Delivery of a signal terminates waiting

timespec_subtract implementation recommended by the GNU C Library



```
/* Subtract the `struct timespec' values X and Y,
   storing the result in RESULT (result = x - y).
   Return 1 if the difference is negative, otherwise 0.  */
int timespec_subtract (struct timespec *result,
                       struct timespec *x, struct timespec *y)
{
  // Perform the carry for the later subtraction by updating Y.
  if (x->tv_nsec < y->tv_nsec) {
    int num_sec = (y->tv_nsec - x->tv_nsec) / 1000000000 + 1;
    y->tv_nsec -= 1000000000 * num_sec;
    y->tv_sec += num_sec;
  }
  if (x->tv_nsec - y->tv_nsec > 1000000000) {
    int num_sec = (x->tv_nsec - y->tv_nsec) / 1000000000;
    y->tv_nsec += 1000000000 * num_sec;
    y->tv_sec -= num_sec;
  }
  // Compute the time remaining to wait. tv_nsec is certainly positive.
  result->tv_sec = x->tv_sec - y->tv_sec;
  result->tv_nsec = x->tv_nsec - y->tv_nsec;
  // Return 1 if result is negative.
  return x->tv_sec < y->tv_sec;
}
```

POSIX timers

- After the desired time interval elapses, the signal (SIGALRM by default) is delivered to the task
- Input parameters are:
  - Time to the first tick
  - The period of the other ticks
  - These can differ
  - time resolution in nanoseconds

POSIX timer – API

- ***timer_create*** – creates timer
- ***timer_settime*** – starts timer
- ***timer_gettime*** – find out remaining time
- (non POSIX) ***timer_connect*** – handler initialization (calls *sigaction*)
- (non POSIX) ***timer_cancel*** – stops the timer (calls *timer_settime* with zero interval)

Timestamp timers

- Thin interface to hardware timers with high resolution (i.e. higher than system tick) and minimal overhead.
  - sysTimestampConnect( )
  - sysTimestampEnable( )
  - sysTimestampDisable( )
  - sysTimestampPeriod( )
  - sysTimestampFreq( )
  - **sysTimestamp( )**
  - sysTimestampLock( )

Watchdog timer

- Timer that calls a specified function upon elapsing of the time interval
- May be used to drive hardware watchdog – if VxWorks freezes, HW watchdog will reset the system.
- Not available for RTP
- Executed as a part of timer interrupt
- API:
  - ***wdCreate*** – creates wdtimer
  - ***wdStart*** – runs wdtimer
  - ***wdCancel*** – cancels the timer
  - ***wdDelete*** – deletes wdtimer

Networking

- Wide range of supported protocols, IPv4/IPv6
- standard API – BSD sockets
- for high throughput applications: zbuf sockets
- supported booting from Ethernet (BOOTP+TFTP/FTP/RSH)

Supported protocols

- SLIP, CSLIP, PPP
- IP, UDP, TCP, ARP, DNS
- DHCP, BOOTP
- OSPF, RIP, NDP
- RPC, RSH
- FTP, TFTP
- NFS
- telnet

Network API – sockets

- standard API for BSD sockets
- Additional libraries: hostLib, ifLib, ftpLib, …
- more detailed description in VxWorks Network Programmer’s Guide

Alternative API – zbuf sockets I.

- Kernel tasks only, not in RTP
- BSD sockets use different buffers in applications and in the kernel – data must be copied between them
- zbuf sockets API enables to share the same buffer between all the layers – no need for copying
- almost all functions from BSD sockets API have corresponding counterparts in zbuf sockets API

Alternative API – zbuf sockets II.

- ***zbufSockSend*** – send zbuffer (TCP)
- ***zbufSockSendTo*** – dtto, UDP
- ***zbufSockBufSend*** – send data from user buffer (TCP)
- ***zbufSockBufSendTo*** – dtto, UDP
- ***zbufSockRecv*** – read data (TCP)
- ***zbufSockRecvfrom*** – dtto, UDP

Interrupts

- Handling interrupts is only possible in kernel tasks, not in RTPs
- Interrupt handler is set up by calling:
  - ***intConnect*** *(some architectures only)*
  - ***vxbIntConnect*** *(as a part of device driver)*
- There is a separate task context for all the interrupt handlers
- Handlers use a separate stack to not overflow application's stack
- Interrupts can be globally disabled/enabled by calling ***intLock***/***intUnlock***
- Interrupt mask can be set by ***intLevelSet***

Interrupt Handlers
(Interrupt Service Routines – ISR)

- Should be as short as possible to minimize interrupt latency (why?)
- **Cannot** call functions that can **cause blocking** e.g.
  - *semTake* (but can call *semGive*), no mutex semaphores
  - *msgQReceive* (be aware of *msgQSend*! If the queue is full, the message is thrown away.)
  - *taskDelay*
  - *taskSuspend*
  - the full list can be found in the documentation
- **Cannot use floating point** calculations
- Debugging: **logMsg()**

Minimizing The Work Performed Within an ISR

1. Program the interrupting device to stop interrupting the CPU
2. Prepare and queue a data structure describing what needs to be done later with the device (status register, ...)
3. Use a semaphore to unblock a task (with appropriate priority) that will perform the necessary work later (when the ISR completes and the task is scheduled).
4. Return from the ISR. The OS runs the scheduler and the just unblocked task will run unless a higher priority task is ready.

Signals vs. interrupts

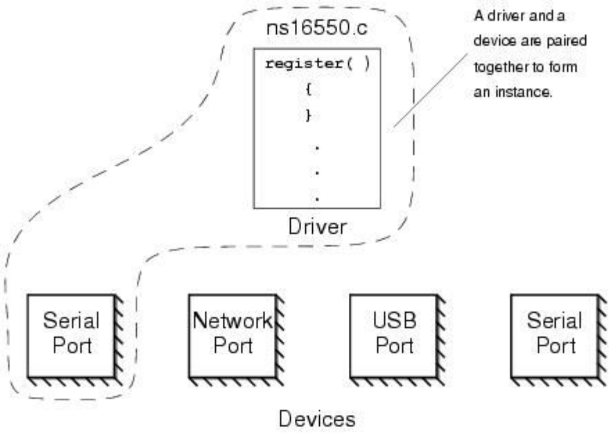
- Interrupts are similar to signals
- In both handlers it is not allowed to call services which **block**
- Need to maintain data consistency (we can't use mutexes)
  - Signal mask in the OS vs. interrupt masking in the CPU
- Signal delivery interrupts some system calls
  - taskDelay etc.; see also SEM_INTERRUPTIBLE flag
  - Interrupts don't influence system calls but a signal can be sent from an interrupt handler

VxWorks Device Drivers

- Means of communication between VxWorks and hardware devices.
- VxBus device drivers
    - VxBus is a software framework for driver development
    - Every driver is a plugable component (object-oriented approach)
    - **Similar architecture as Linux drivers**

VxBus

- Software framework for writing device drivers in VxWorks.
- Terminology: *device* = HW, *driver* = SW
- Drivers publishes *methods* (entry points)
- vxbDevMethodGet(): query which instance supports the given method (e.g. {vxbTimerFuncGet}() for timer instance)
- Driver classes:
  PCI, USB, **FDT**, ...
- Every class defines mandatory methods



Device tree
(Flat Device Tree, FDT)

- Data structure describing HW components and their interconnection
  - Only for devices that cannot be automatically detected/enumerated, e.g. not for USB and PCIe devices
- Allows having a single kernel image running on different boards/CPUs with the same architecture
- Heavily used in Linux (PowerPC, ARM)
  - x86 have ACPI tables for similar purpose
  - Linux device trees are not compatible with VxWorks device trees
  - Device tree source (*.dts) -> compiler (dtc) -> Device tree binary (*.dtb)

Device tree (example)

```
/ {
  memory {
    device_type = "memory";
    reg = <0x00000000 0x10000000 0x30000000 0x10000000>;
  };
  sampdev_1: sampdev@f8002000 {
        compatible = "ctu,sample-driver";
        reg = <0xf8002000 0x1000>;
        clock-frequency = <111111111>; // Hz
        interrupt-parent = <&intc>;
        interrupts = <69>;
    };
    intc: interrupt-controller@f8f01000 {
        compatible = "arm,gic";
        ...
  };
...
}
```

Driver methods

```
LOCAL VXB_DRV_METHOD sampleMethods[] =
{
    /* DEVICE API */
    { VXB_DEVMETHOD_CALL(vxbDevProbe), sampleProbe },
    { VXB_DEVMETHOD_CALL(vxbDevAttach), sampleAttach },
    { VXB_DEVMETHOD_CALL(vxbDevShutdown), sampleShutdown },
    { VXB_DEVMETHOD_CALL(vxbDevDetach), sampleDetach },
    { 0, NULL }
};
VXB_DRV vxbSampleDrv =
{
    { NULL },                   /* Linked list header */
    "sample",                   /* Name */
    "Sample VxBus driver",      /* Description */
    VXB_BUSID_FDT,              /* Class */
    0,                          /* Flags */
    0,                          /* Reference count */
    sampleMethods               /* Method table */
};
VXB_DRV_DEF(vxbSampleDrv);
```

Probe method

- Called by the kernel to ask the driver whether it can drive the detected device instance
    - In case of PCI or USB devices, this is a result of bus enumeration
    - In case of FDT devices, drivers are called for every node in the device tree

```
LOCAL VXB_FDT_DEV_MATCH_ENTRY sampleDriverMatch[] = {
    {“ctu,sample-driver”, NULL}, {} /* Empty terminated list */
};


LOCAL STATUS sampleProbe(VXB_DEV_ID pInst) {
    return vxbFdtDevMatch(pInst, sampleDriverMatch, NULL);
}
```

Attach method

(initialize the device and the driver data)

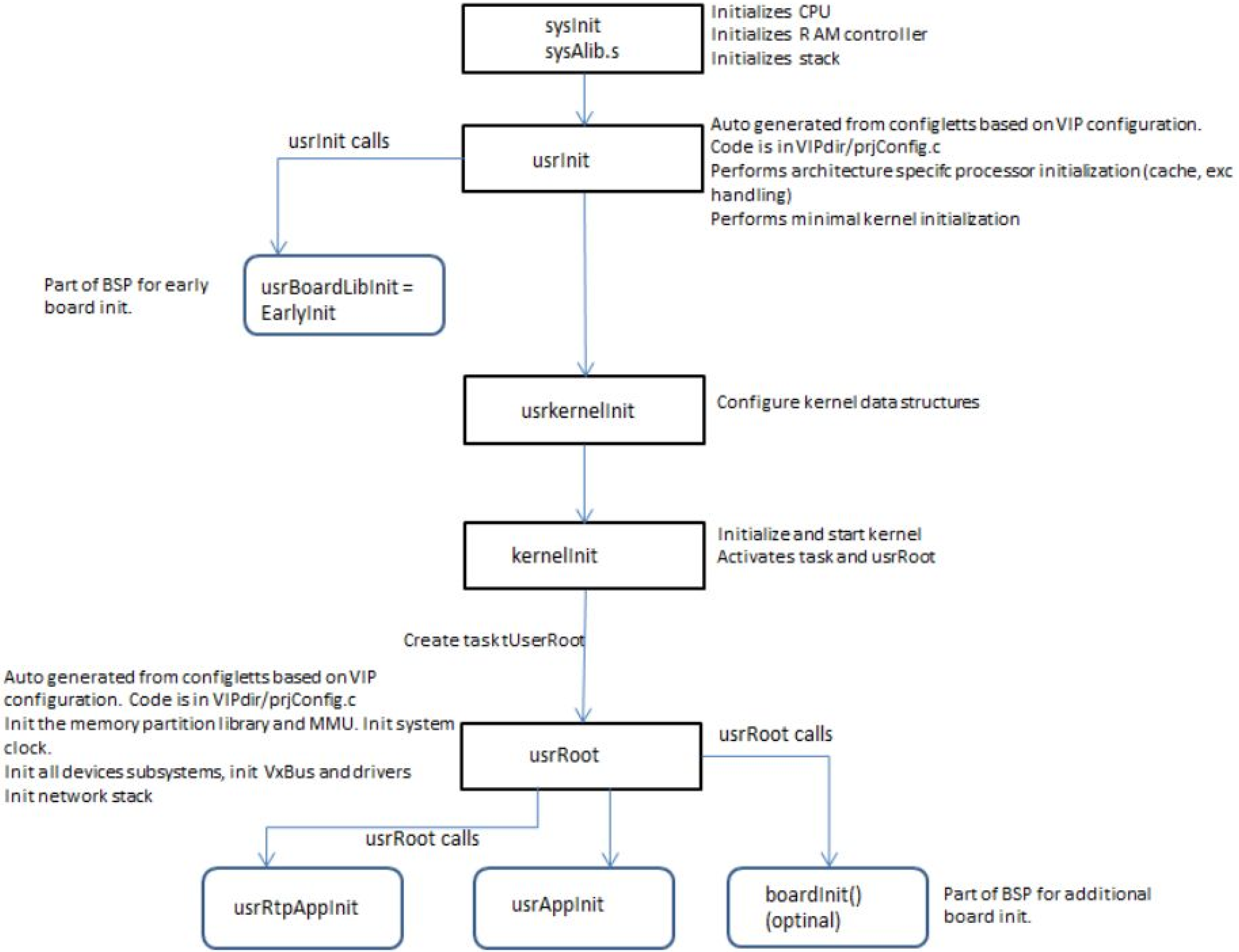
```
LOCAL STATUS sampleAttach(VXB_DEV_ID pInst) {
    VXB_RESOURCE *pRes = NULL;
    VXB_RESOURCE *pResIrq = NULL;

    pMyDeviceData = (struct my_device *)vxbMemAlloc(sizeof(struct my_device));
    if (pMyDevice == NULL) {perror("vxbMemAlloc"); goto error;}
    vxbDevSoftcSet(pInst, pMyDeviceData);

    // Map device registers to virtual memory
    pRes = vxbResourceAlloc(pInst, VXB_RES_MEMORY, 0);
    if (pRes == NULL) {perror("vxbResourceAlloc(MEMORY)"); goto error;}
    pMyDeviceData->regs = ((VXB_RESOURCE_ADR *)(pRes->pRes))->virtAddr;

    // Setup the interrupt
    pResIrq = vxbResourceAlloc(pInst, VXB_RES_IRQ, 0);
    if (pResIrq == NULL) {perror("vxbResourceAlloc(IRQ)!\n"); goto error;}

    // Connect the interrupt handler
    STATUS s1 = vxbIntConnect(pInst, pResIrq,  my_irq_handler, 0);
    if (s1 == ERROR) {perror("vxbIntConnect"); goto error;}
    // Enable interrupts
    if (vxbIntEnable(pInst, pResIrq) == ERROR) {perror("vxbIntEnable"); vxbIntDisconnect(pInst, pResIrq); goto
error;}

    return OK;

error:
    // Free any allocated resources
    if (pResIrq != NULL)
        vxbResourceFree(pInst, pResIrq);
    if (pRes != NULL)
        vxbResourceFree(pInst, pRes);
    if (pMyDeviceData != NULL)
        vxbMemFree(pMyDeviceData);
    return ERROR;
}
```

BSP – board support package

- Enables VxWorks to run on the specific hardware (board)
- Provides
  - initialization of hardware and special device drivers
  - detection of size and type of memory
  - preparation of interrupt systems
  - preparation of timers
- Usually provided by hardware vendors
- Source code of BSPs can be fount at:
  - /opt/WindRiver/vxworks/22.06/source/os/arch/
  - Our board: /opt/psr/mzapo-image/xlnx_zynq7k_3_0_0_1

Writing own BSP – boot sequence
(similar for all “embedded” systems)

- Kernel image is located in FLASH/ROM memory or is loaded from network/disk by a bootloader to RAM.
- Initialize processor for running C (_romInit)
  - in assembler
  - initialize memory and a temporary stack
  - disable interrupts
- romStart is called (*installDir*/vxworks-7.x/target/config/all/bootInit.c)
  - copy (and decompress) data sections from ROM to RAM
- _sysInit() is called
  - initialize cache, vector table; perform board specific initialization
  - start multi-tasking and user-booting task



Initializes CPU
Initializes R AM controller
Initializes stack

Auto generated from configletts based on VIP configuration.
Code is in VIPdir/prjConfig.c
Performs architecture specifc processor initialization (cache, exc handling)
Performs minimal kernel initialization

Part of BSP for early board init.

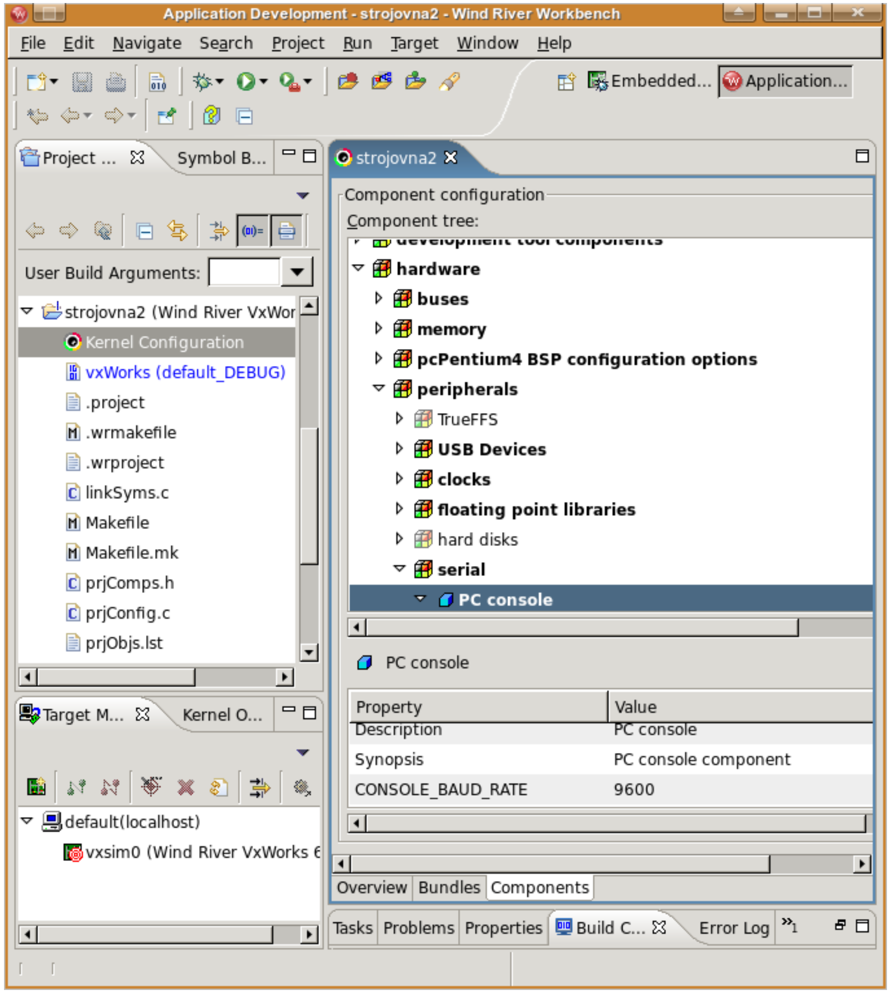
Configure kernel data structures

Initialize and start kernel
Activates task and usrRoot

Auto generated from configletts based on VIP configuration. Code is in VIPdir/prjConfig.c
Init the memory partition library and MMU. Init system clock.
Init all devices subsystems, init VxBus and drivers
Init network stack

Part of BSP for additional board init.

Preparing a Custom VxWorks Kernel



User Build Arguments:

- strojovna2 (Wind River VxWor
  - Kernel Configuration
  - vxWorks (default_DEBUG)
  - .project
  - .wrmakefile
  - .wrproject
  - linkSyms.c
  - Makefile
  - Makefile.mk
  - prjComps.h
  - prjConfig.c
  - prjObjs.lst

- default(localhost)
  - vxsim0 (Wind River VxWorks 6

Component configuration

Component tree:

- development tool components
- hardware
  - buses
  - memory
  - pcPentium4 BSP configuration options
  - peripherals
    - TrueFFS
    - USB Devices
    - clocks
    - floating point libraries
    - hard disks
    - serial
      - PC console

PC console

Property	Value
Description	PC console
Synopsis	PC console component
CONSOLE_BAUD_RATE	9600

- VxWorks Image Project
- Choose which components to include and their settings
- Run “build”
  - Most components are available as binary only objects
  - => linking

Multiprocessor systems

- SMP – Symmetric Multi-Processing
  - All CPUs share the whole memory
  - A task can run on arbitrary CPU
  - Need for different synchronization primitives
    - Spinlocks, memory barriers, cache coherency...
- AMP – Asymmetric Multi-Processing
  - Supported only on multicore systems
  - Each CPU runs independent VxWorks OS copy
  - Ability to send messages between CPUs

Differences between SMP and AMP

Figure 17-2 :SMP System

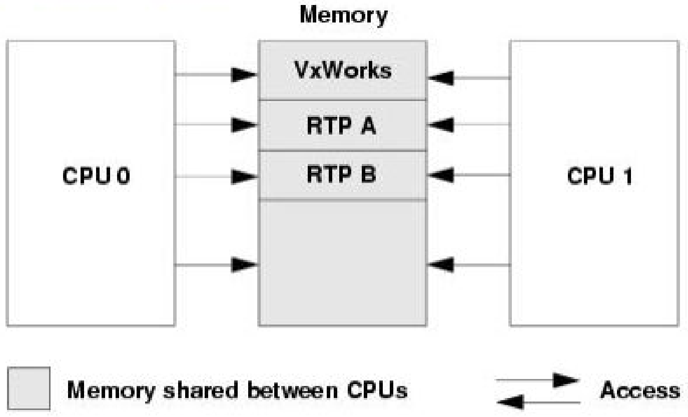
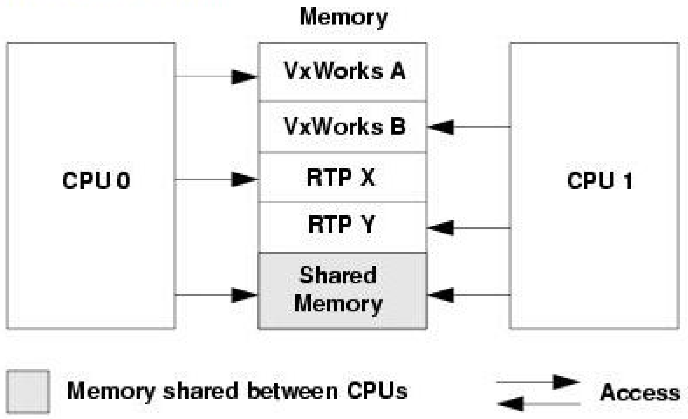


Figure 17-3 :AMP System



Linux vs. VxWorks

- Price and license
- VxWorks is much simpler than Linux
  - Less overhead (sometimes)
  - Smaller memory footprint
- VxWorks has not so wide HW support
- VxWorks is certified for “almost everything”
- Linux real-time support is already quite good (with the PREEMPT_RT patch, which was fully merged in Linux 6.12, Q4 2024)