Programming

Real-Time Applications on

QNX

NEUTRINO RTOS

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mário de Sousa

msousa@fe.up.pt

NEUTRINO RTOS

Real-Time Applications on QNX

The QNX Architecture

The QNX Neutrino Micro-Kernel

Threads and Processes

Thread CPU Scheduling

POSIX IPC Services

QNX Neutrino IPC Service

Clocks and Timers

QNX Design Goals

Provide the POSIX API...

- POSIX – Portable Operating System Interface [for uniX]
  A set of IEEE standards on how to interface with the OS.
- API – Application Programming Interface

...to a wide range of hardware...

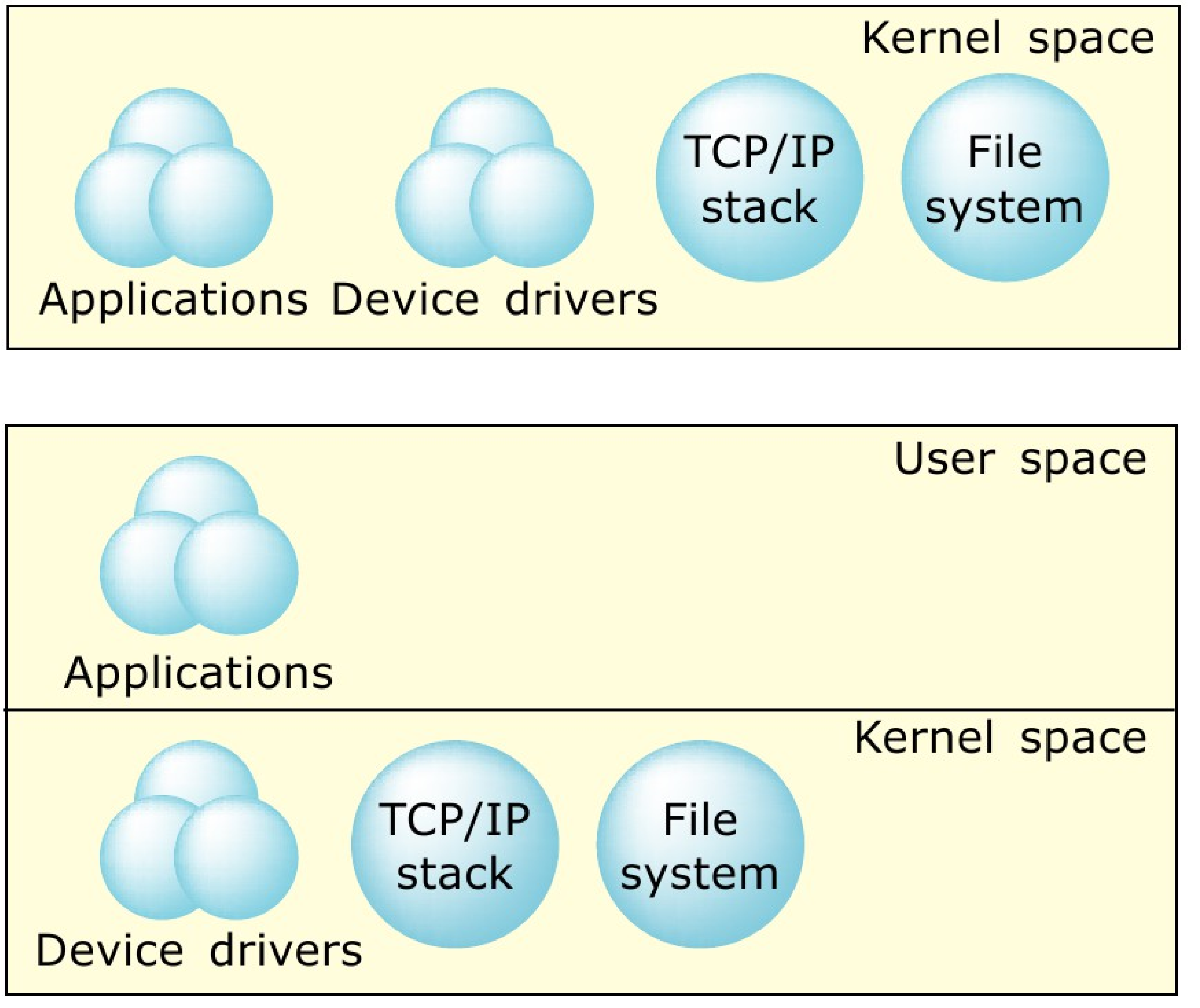
- From resource constrained embedded systems, to large distributed computing environments
- Supported CPU architectures:
  x86, ARM, PowerPC, MIPS, XScale, SH-4

...for Real-Time mission critical applications.

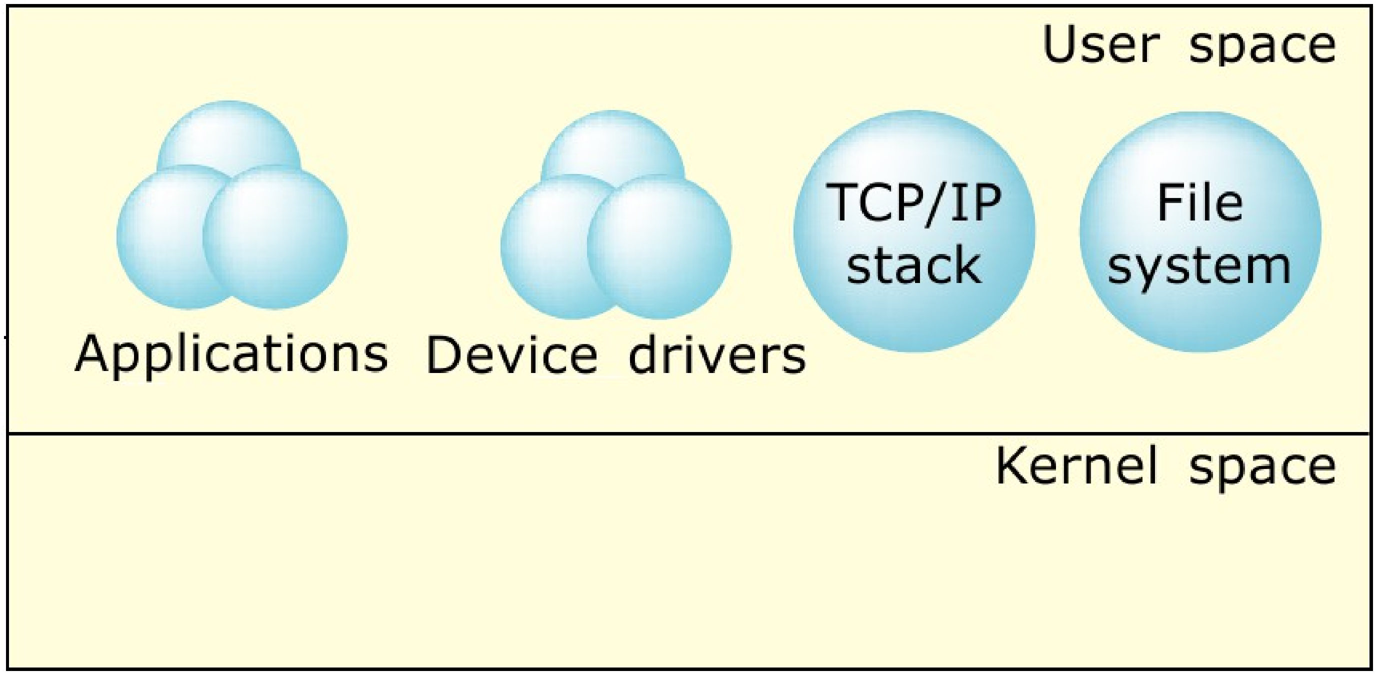
Traditional OS Architectures

Executive

Monolithic Kernel



Micro-kernel OS Architecture

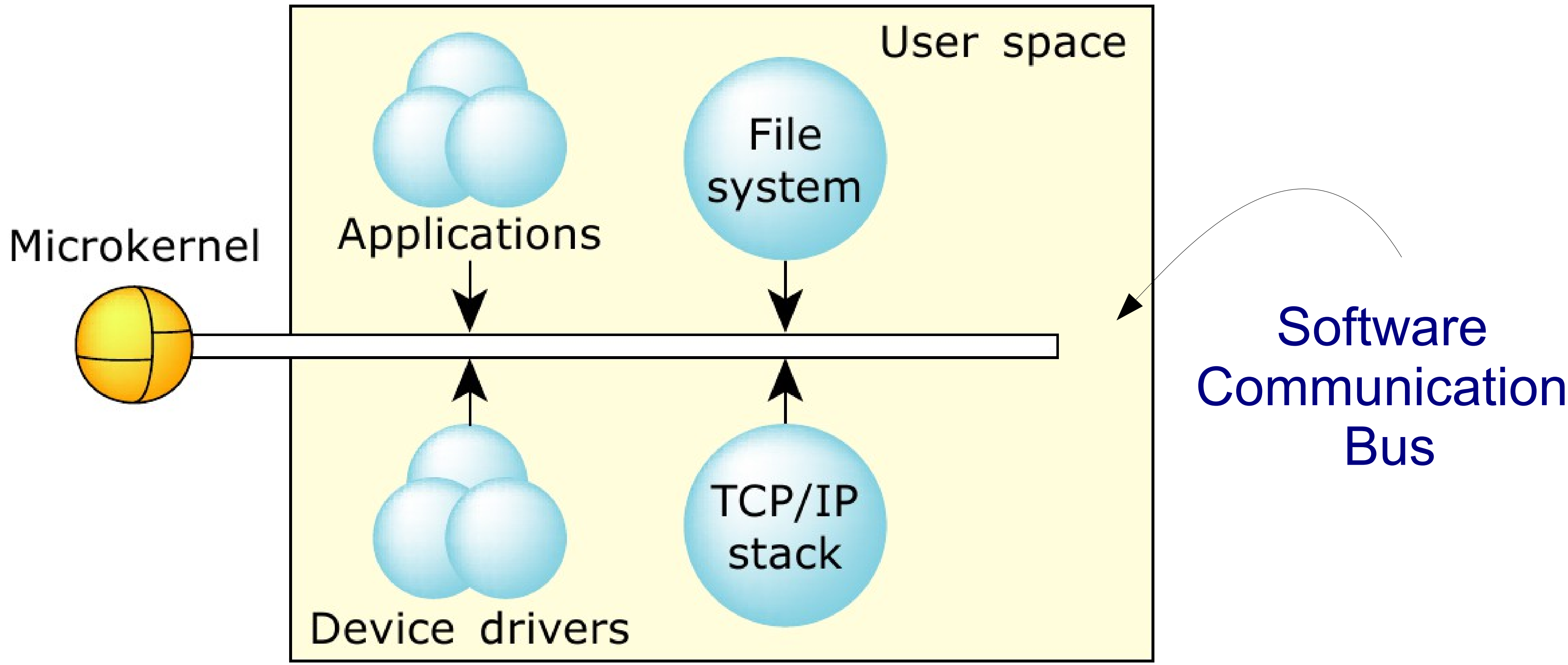


Main identifying characteristic: MODULARITY

Micro-kernel => small kernel (this is just a side-effect of the modularity!)

System processes are essentially indistinguishable from any user-written program — they use the same public API and kernel services available to any (suitably privileged) user process.

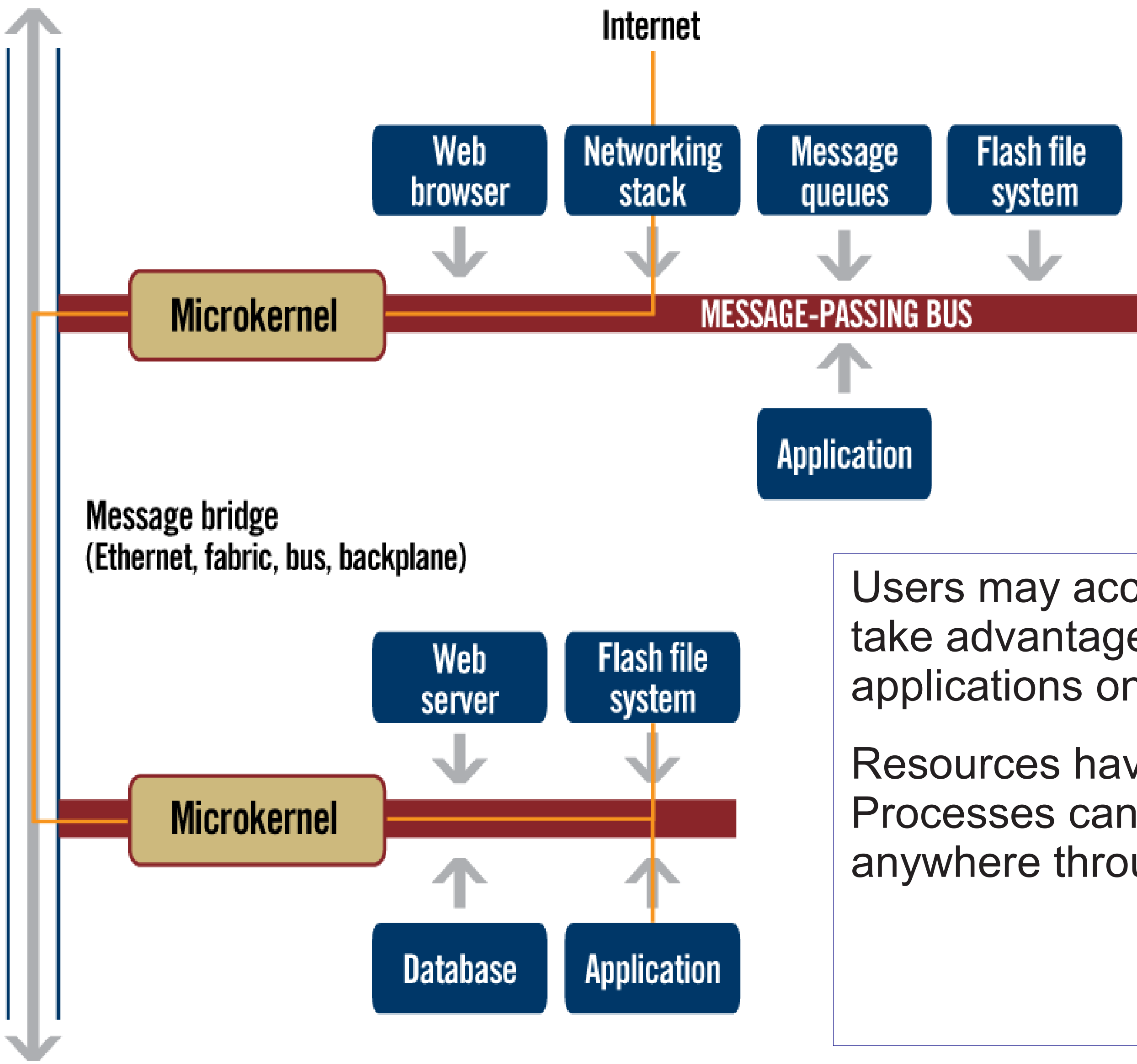
Micro-kernel OS Architecture



A message is a parcel of bytes passed from one process to another. (The OS attaches no special meaning to the content of a message — the data in a message only has meaning for the sender and its receiver)

Synchronizing the execution of several processes is also achieved by exchanging messages.

NEUTRINO RTOS



QNX extends the message passing architecture across virtually any network interconnection technology.

Users may access files anywhere on the network, take advantage of any peripheral device, and run applications on any machine on the network

Resources have true location independence: Processes can communicate in the same manner anywhere throughout the entire network.

Micro-kernel OS Architecture

Advantages:

Commercial

- Modularity allows the exact same kernel to be used on embedded systems, as well as large platforms.
- A range of products may be created that differ only by the modules that are installed on the platform.

Technical

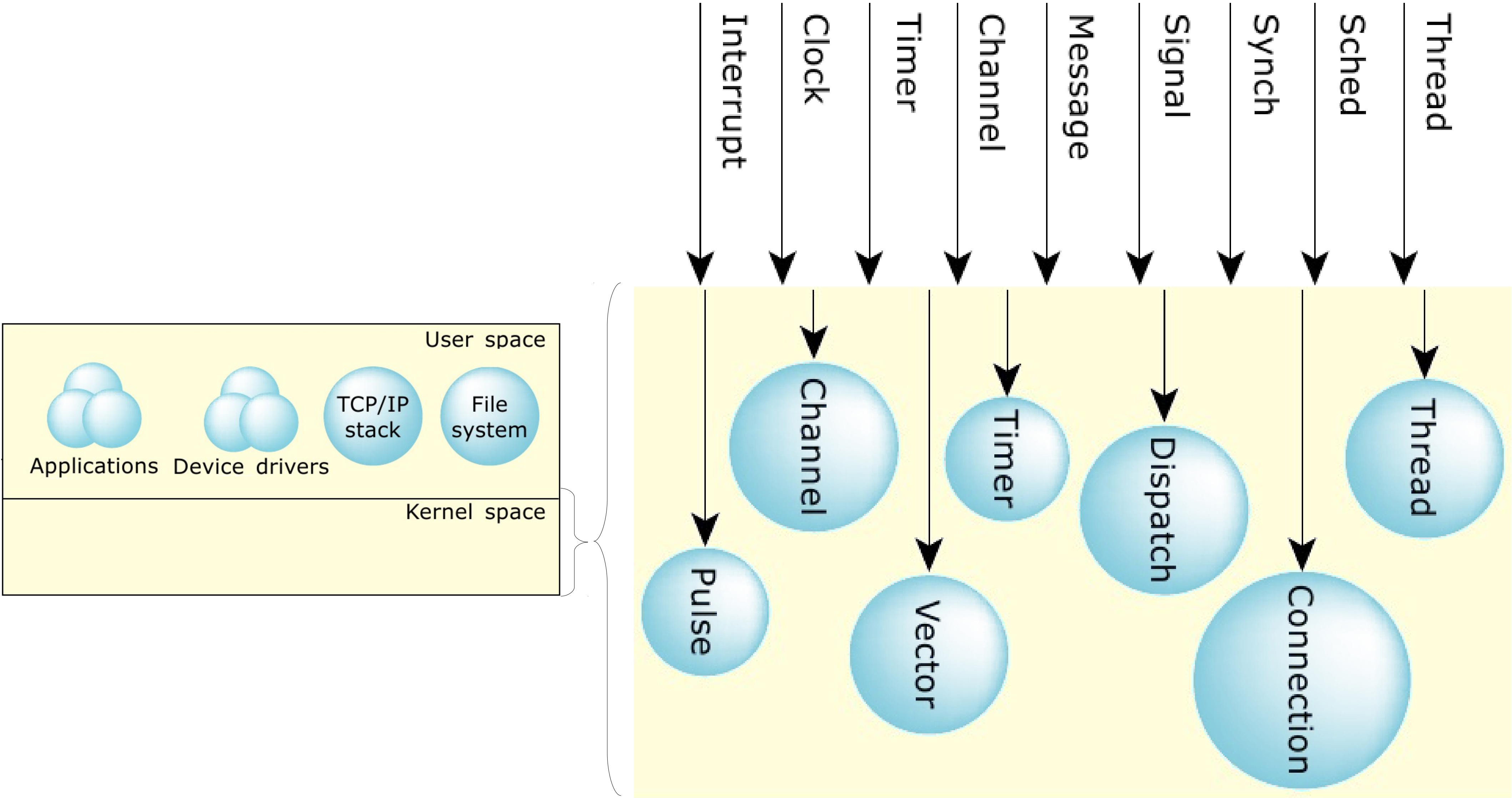
- Memory protection between device drivers means a buggy device driver will not crash the system!
- Easy for the user to extend the OS by writing applications that provide system services (e.g. a device driver).

Real-Time Applications on QNX

NEUTRINO RTOS

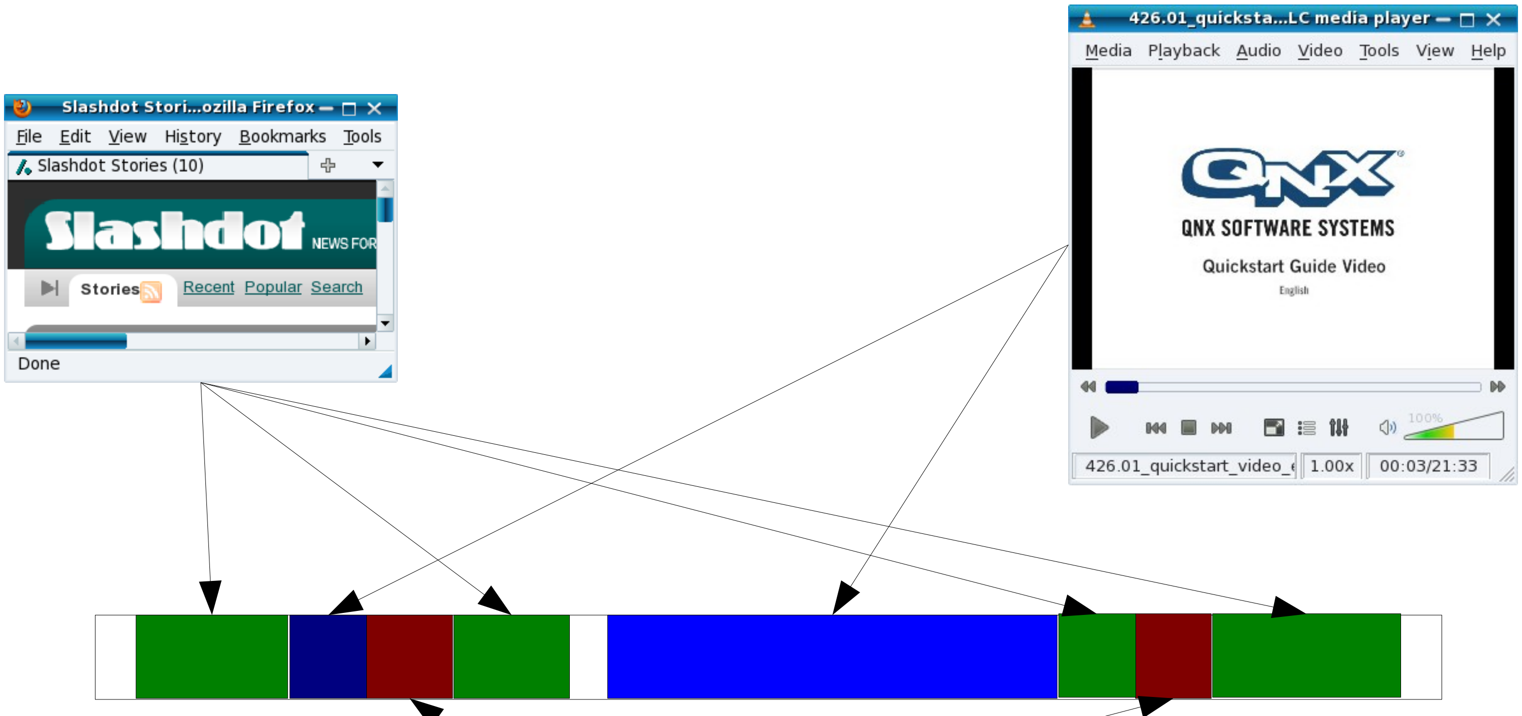
- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Service
- Clocks and Timers

QNX Neutrino Micro-Kernel

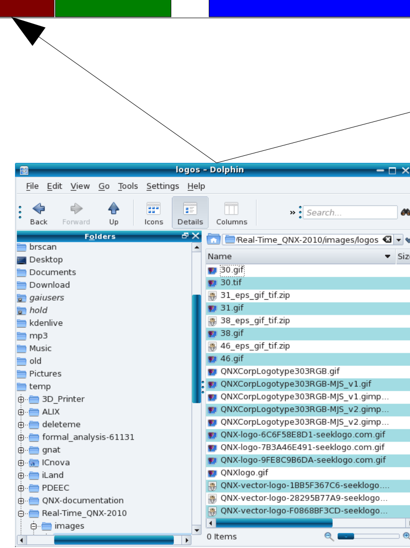


QNX
NEUTRINO RTOS

CPU Time-Sharing...



CPU



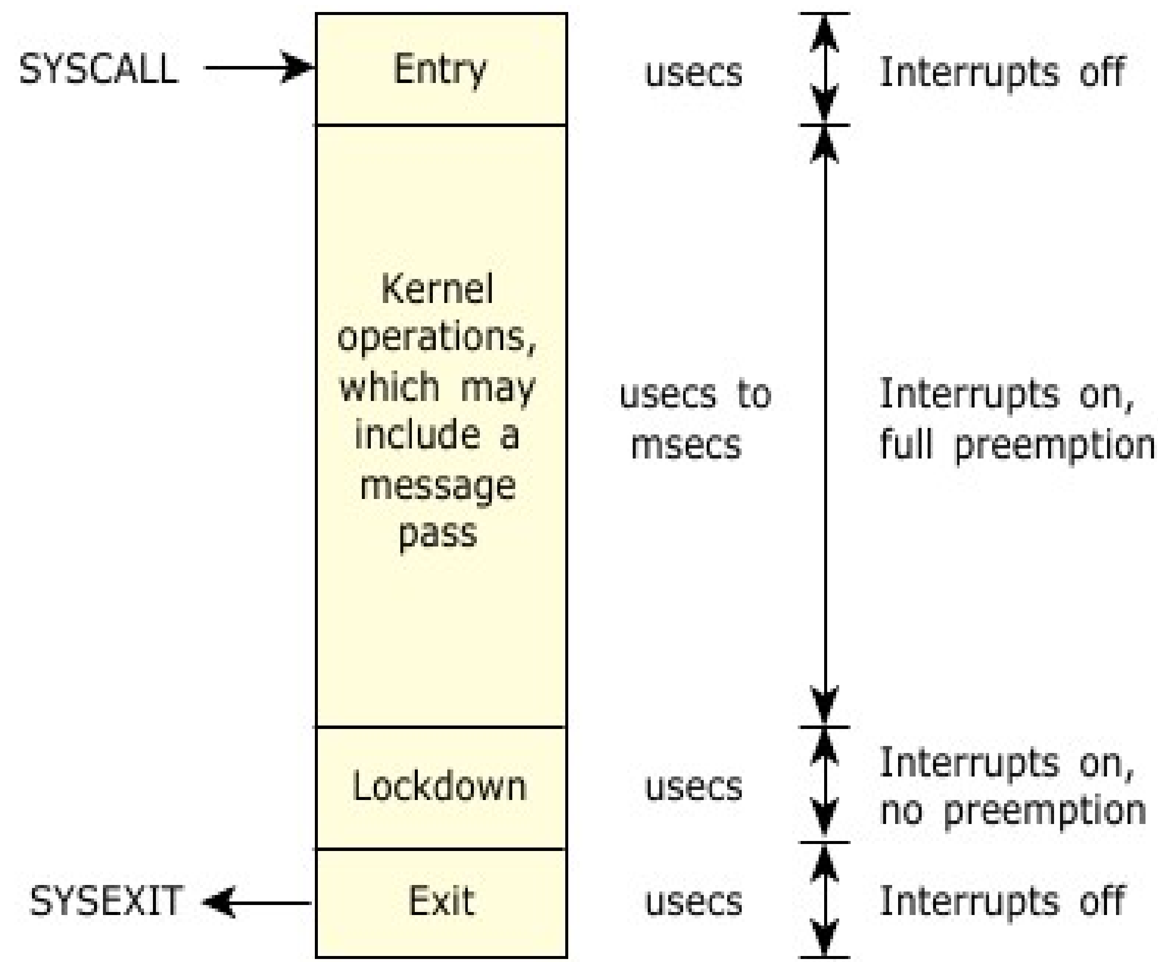
Running many programs concurrently on a single CPU requires the CPU to be shared in time...

Pre-emption
when one program is exchanged for another!
Usually, we want this to be fast, so the video does not stutter!

QNX Neutrino Micro-Kernel

The OS is fully preemptible,
- even when passing messages between processes;
- the simplicity of the microkernel permits short nonpreemptible code sections.

Services in the microkernel are all executed quickly.
- Operations requiring significant work are assigned to external processes/threads,
(i.e. when the context switch time is insignificant in relation to the work done to service the request).



Real-Time Applications on QNX

NEUTRINO RTOS

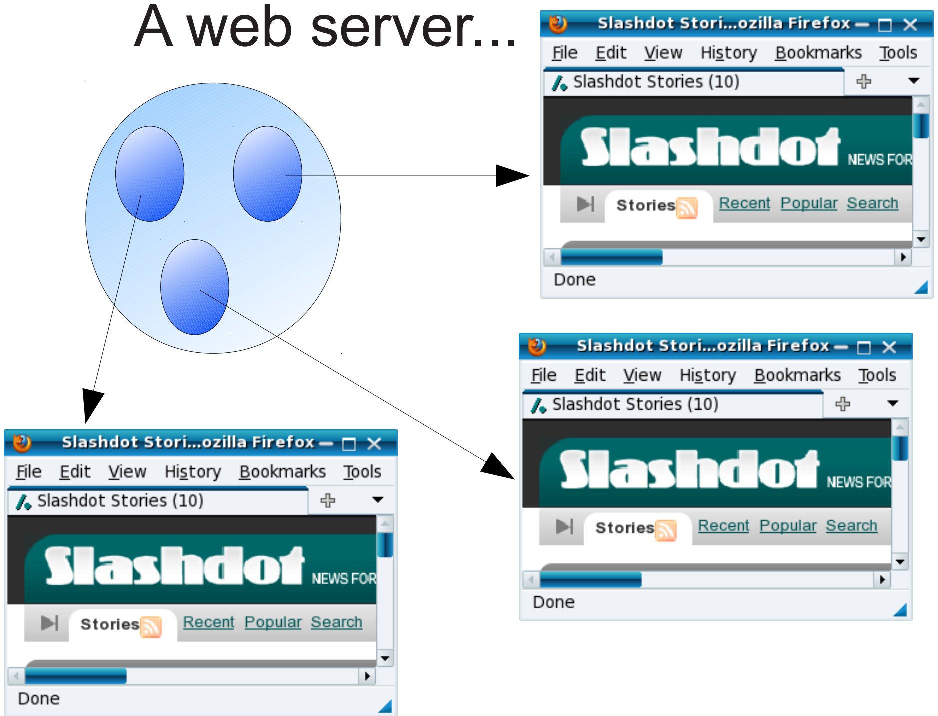
- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Service
- Clocks and Timers

NEUTRINO RTOS

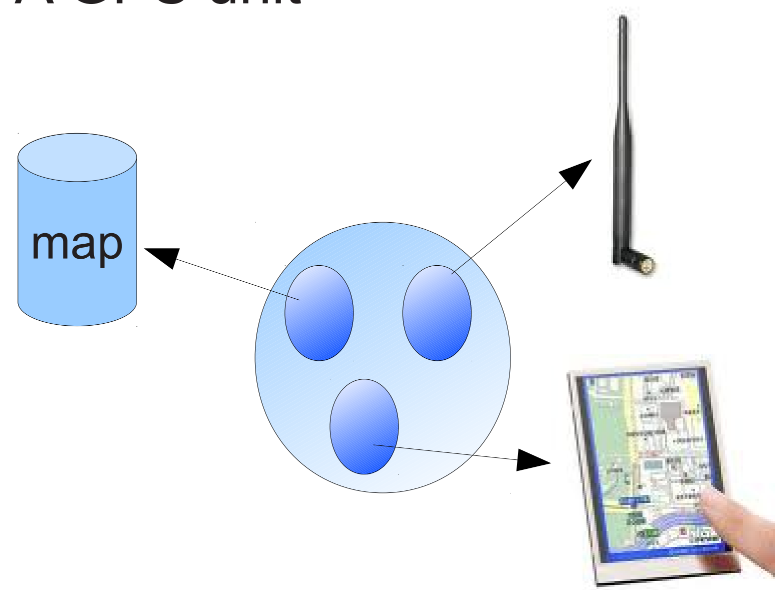
Concurrency

When building an application, sometimes we need it to do several things at the same time (i.e. concurrently).

A web server...



A GPS unit



Concurrency

NEUTRINO RTOS

Using a cyclic executive...

```
int main (void) {
  Hardware_init();

  FuncX_init();
  FuncY_init();
  FuncZ_init();

  while (1) {
    FuncX();
    FuncY();
    FuncZ();
  }
}
```

Using interrupts...

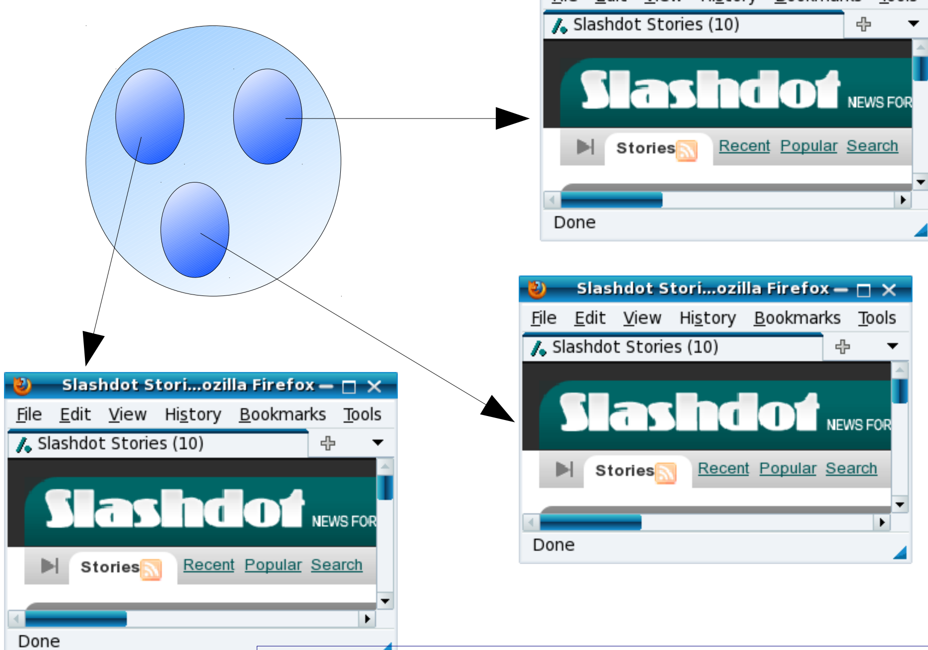
```
int main (void) {
  Hardware_init();
  FuncX_init();
  FuncY_init();
  FuncZ_init();
  while (1);
}

int FuncX_init(void){
 // executed upon arrival
 // of interrupt
 ...
}
```

Processes and Threads

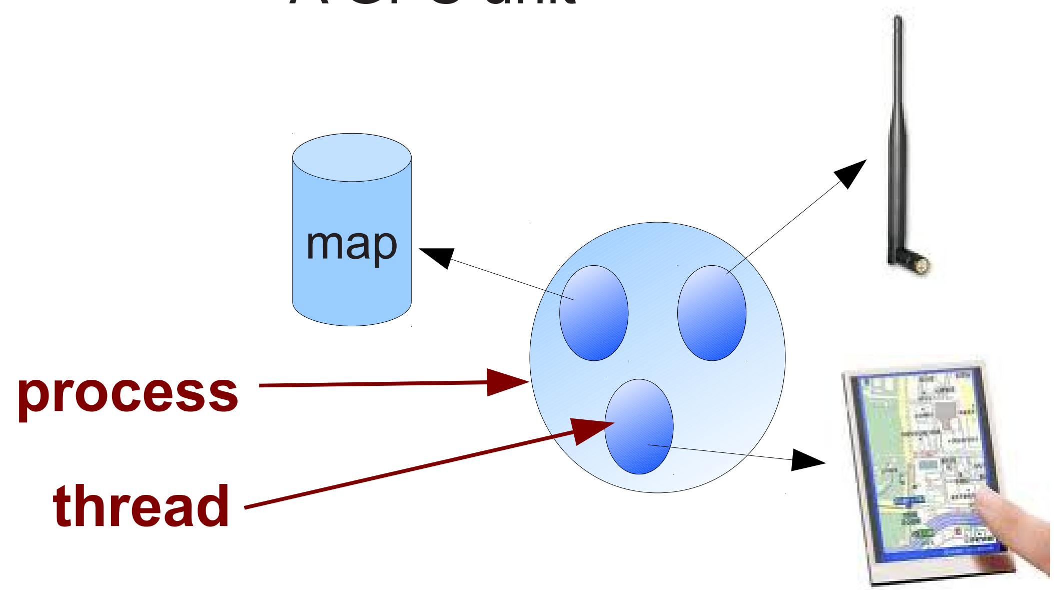
NEUTRINO RTOS

The operating systems manage abstract entities (threads, processes) that simulate concurrent execution, even on a single CPU.

A web server...



A GPS unit



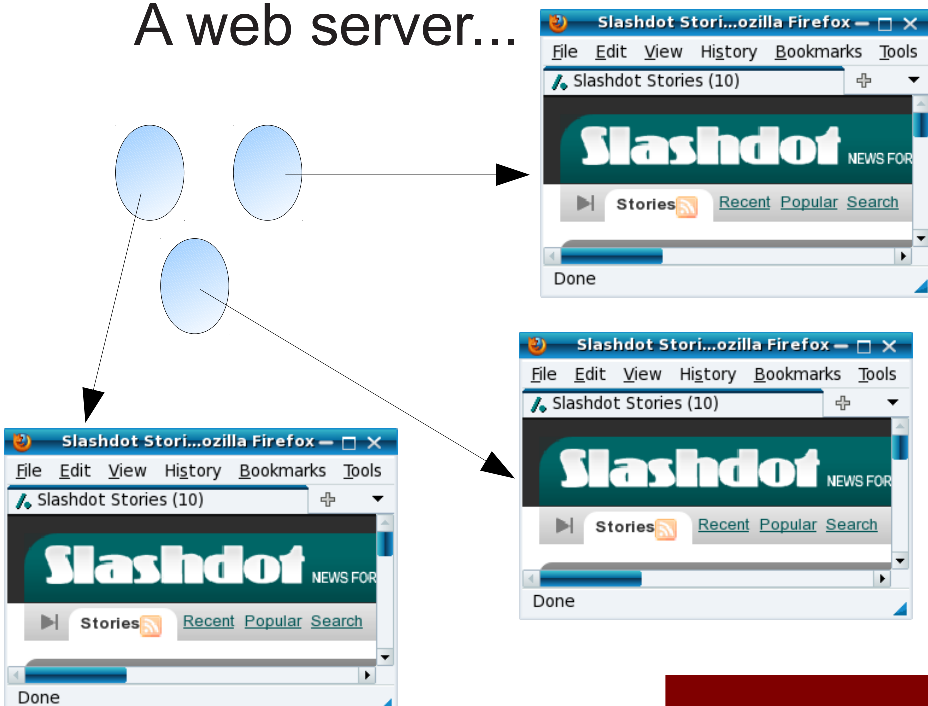
Thread - the minimum “unit of execution,” scheduled by the kernel.

Process - a “container” for threads, (contains at least one thread).

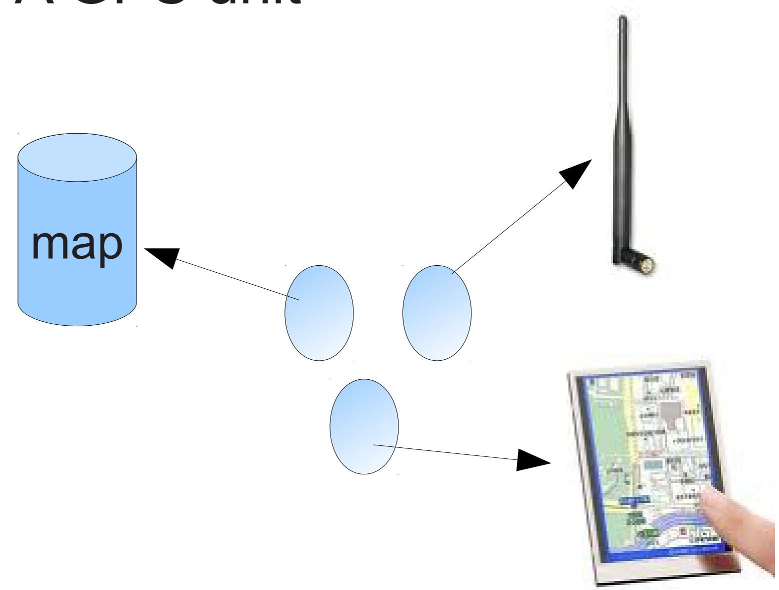
Processes and Threads

When building an application, sometimes we need it to do several things at the same time (i.e. concurrently).

A web server...



A GPS unit



Why not use several concurrent applications (i.e. processes)?

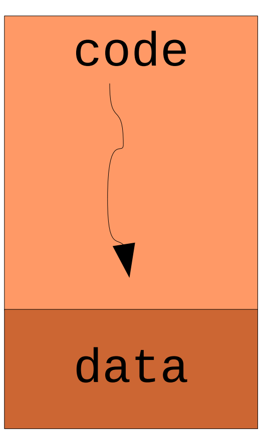
Processes and Threads

The memory used by each process is protected from each other;

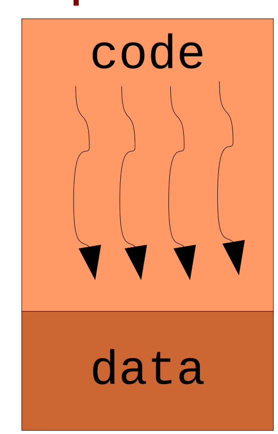
Each process may contain one or more threads that share the process’s memory.

The choice of using threads or processes affects not only the concurrency capabilities of the application, but also the IPC and synchronization services the application may use.

Single threaded process



Multi threaded process



Why not use several concurrent applications (i.e. processes)?

NEUTRINO RTOS

Real-Time Applications on QNX

- **The QNX Architecture**
- **The QNX Neutrino Micro-Kernel**
- **Threads and Processes**
  - **Processes**
  - **Threads**
- **...**

Process Life-Cycle

Creation

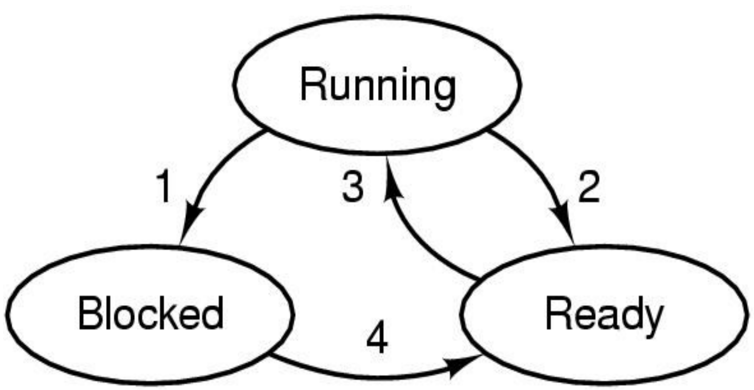
fork()

Termination

void _exit(int status)
(cancel calling process)

int kill(pid_t pid, int sig)
(kill another process)

the OS decides to kill it
(lack of resources,
invalid operation)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process Life-Cycle

```
#include <sys/types.h>
#include <process.h>
pid_t fork(void); /*clones the calling process*/
```

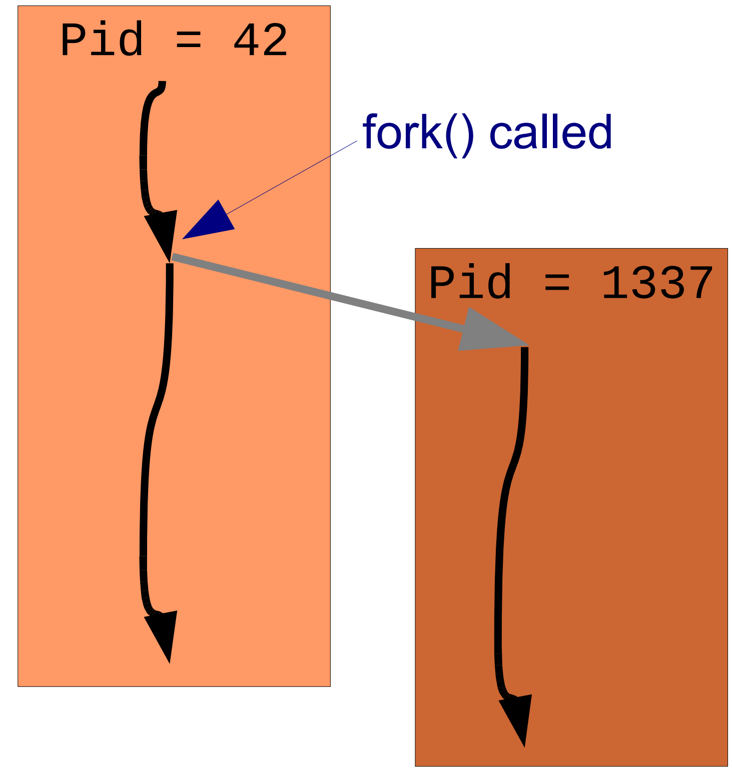
Creates a new process (child process) which is an exact copy of the calling process (parent process), except for the following:

- The child process has its own memory area.
- The child process has a unique process ID, and a different parent process ID.
- The child process has its own copy of the parent's file descriptors.
- File locks previously set by the parent aren't inherited by the child.
- Pending alarms are cleared for the child process.
- The set of signals pending for the child process is initialized to the empty set.

Process Life-Cycle

```
(...)
pid_t pid = fork();
if (pid == 0) {
  child_process_function();
} else  {
  parent_process_function();
}
```

Process Creation



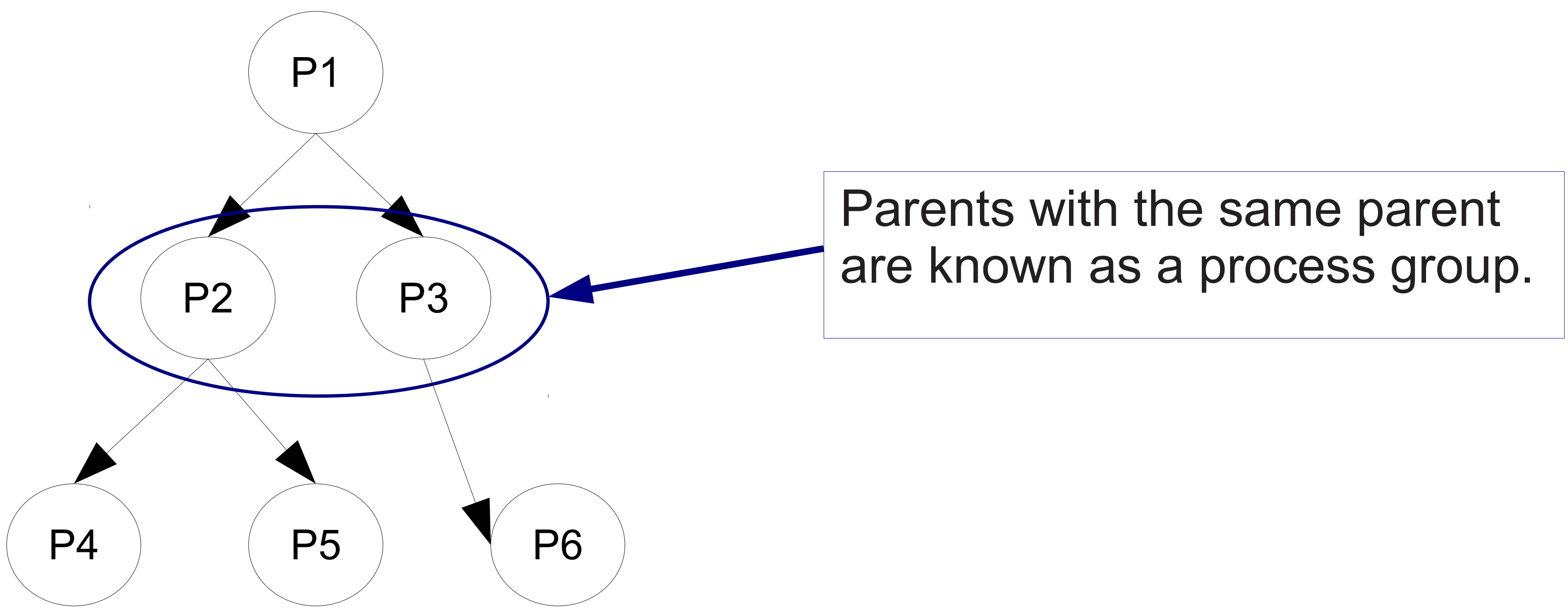
The fork() function returns:

- 0 on the child process.
- the child process ID, in the parent process.

QNX NEUTRINO RTOS

Process Relations

In POSIX, there is a hierarchical family relation between processes...



Launching Another Executable

Problem: How can a process launch a new process that will execute a program stored in a file (e.g. /usr/bin/doom)

Solution: Use fork() followed by execv()

```
#include <process.h>
int execv( const char * path, char * const argv[] );
```

```
(...)
pid_t pid = fork();
if (pid == 0) {
  execv("/usr/bin/doom", NULL);
} else  ...
```

More Process System Calls

```
#include <sys/types.h>
#include <process.h>
 pid_t getpid(void);
 pid_t getppid(void);
```

```
#include <stdlib.h>
 void  exit(int status)
 void _exit(int status)
```

`getpid()` get `pid` of calling process

`getppid()` get `pid` of parent process

`_exit()` terminate calling process

`exit()` C library function...

- First calls all exit functions registered with `at_exit()`
- And then calls `_exit()`

NEUTRINO RTOS

Process Synchronisation

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Suspend process execution until:

Any child process terminates - `wait()`

A specific child process terminates - `waitpid()`

`status` returns the value that the terminating process passed when calling `void _exit(int status)`

`options` controls detailed functioning of `waitpid()`
(e.g.: WNOHANG — return immediately if there are no children to wait for)

Process Synchronisation

```
...
int status;
pid_t child_pid;
...

if ((child_pid = wait(&status)) != -1) {
  if (WIFEXITED(status) != 0)
    printf("Process %d exited with status %d\n",
           pid, WEXITSTATUS(status));
  else
    printf("Process %d exited abnormally\n", pid);
}
```

Processes: Conclusion

Other Inter-Process Synchronisation primitives:

- Pipes
- Sockets
- Shared Memory
- Since each process always has a default thread, use thread synchronisation primitives between the threads of the two processes
  - POSIX thread synchronisation primitives may not always work when threads belong to distinct processes. It depends on the specific OS implementation.
  - QNX Neutrino allows inter-process thread synchronisation!

Real-Time Applications on QNX

The QNX Architecture

The QNX Neutrino Micro-Kernel

Threads and Processes

Processes

Threads

...

Thread Life-Cycle

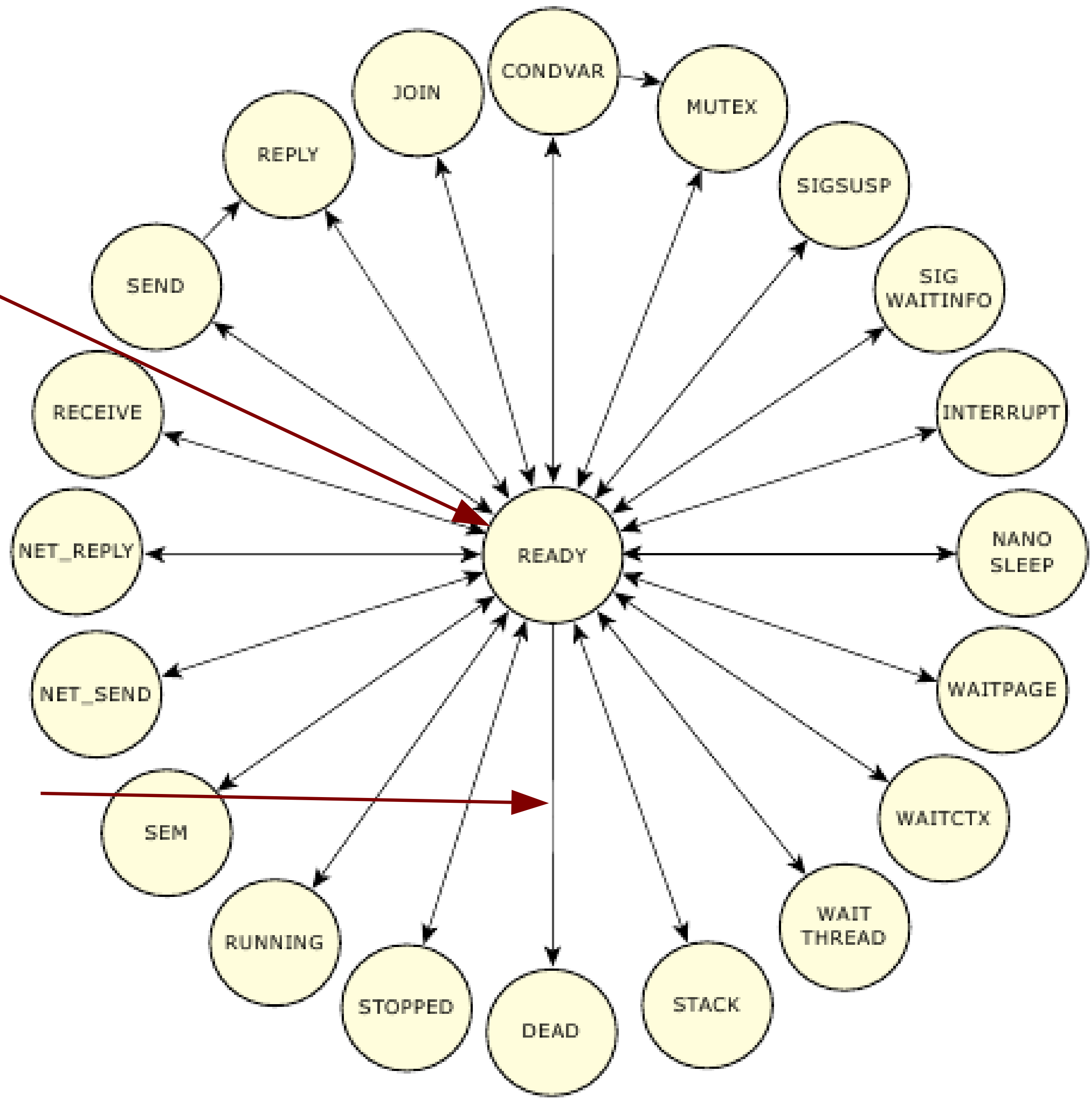
Creation

pthread_create(...)

Termination

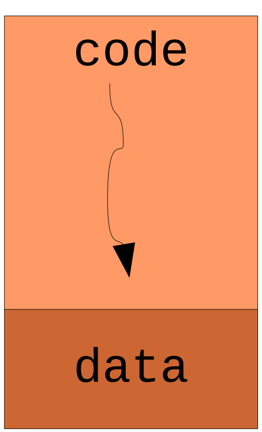
pthread_exit(...)
(cancel calling thread)

pthread_cancel(...)
(cancel another thread)

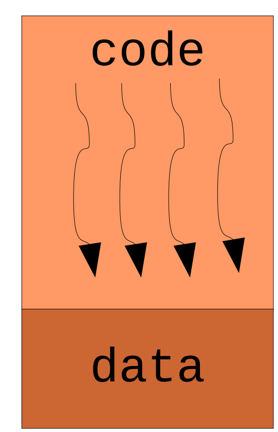


Threads

- Threads within a process share everything within the process's address space (e.g. open files)
- However, each thread still has some "private" data:
  - Its own register set (instruction pointer, stack pointer, ...)
  - Its own stack (used, for e.g., for local variables).
  - tid (a unique number within the process)
  - name (a Neutrino extension to POSIX)
  - Signal mask
  - Thread local storage
  - Cancellation handlers (callback functions that are executed when the thread terminates)



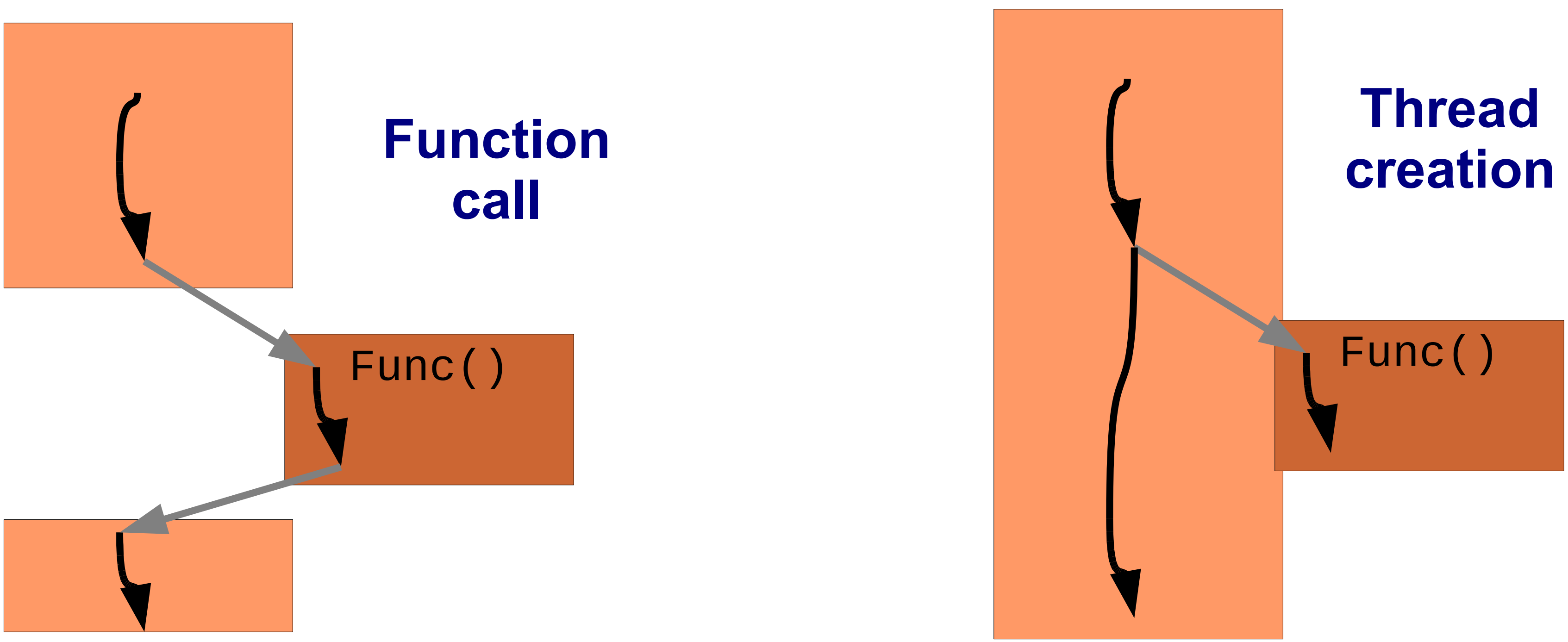
Single threaded process



Multi threaded process

Thread Creation

```
#include <pthread.h>
int pthread_create( pthread_t* thread,
                    const pthread_attr_t* attr,
                    void* (*start_routine)(void* ),
                    void* arg );
```



NEUTRINO RTOS

Thread Creation

```
#include <pthread.h>

void* function(void* arg) {
  int *parm_ptr = (int *)arg;
  printf("This is thread %d, arg=%d\n",
          pthread_self(), *parm_ptr);
  return NULL;
}

int main(void) {
  int parm0=10; int parm1=11; int parm2=12;
  pthread_create(NULL, NULL, function, (void *)&parm0);
  pthread_create(NULL, NULL, function, (void *)&parm1);
  pthread_create(NULL, NULL, function, (void *)&parm2);

  /* Allow threads to run for 60 seconds. */
  sleep(60);
  return EXIT_SUCCESS;
}
```

Thread Creation

```
#include <pthread.h>

void* function(void* arg) {
  int *parm_ptr = (int *)arg;
  printf("This is thread %d, arg=%d\n",
          pthread_self(), *parm_ptr);
  return NULL;
}
...

This is thread 0, arg=10
This is thread 1, arg=11
This is thread 2, arg=12
```

NEUTRINO RTOS

Thread Creation

```
void* function(void* arg) {
  int *parm_ptr = (int *)arg;
  for (i = 0, i < 5, i++) {
    printf("This is thread %d, loop=%d\n", *parm_ptr, i);
    sleep(1);
  }
  pthread_exit(NULL); //another way of terminating thread
}

int main(void) {
  int parm=10;
  pthread_create(NULL, NULL, function, (void *)&parm);
  parm=20;
  pthread_create(NULL, NULL, function, (void *)&parm);
  parm=30;
  pthread_create(NULL, NULL, function, (void *)&parm);

  sleep(60); return EXIT_SUCCESS;
}
```

Thread Creation

```
void* function(void* arg) {
  int *parm_ptr = (int *)arg;
  for (i = 0, i < 5, i++) {
    printf("This is thread %d, loop=%d\n", *parm_ptr, i);
    sleep(1);
  }
  pthread_exit(NULL); //another way of terminating thread
}
```

```
This is thread 10, loop=0
This is thread 20, loop=0
This is thread 30, loop=0
This is thread 30, loop=1
This is thread 30, loop=1
This is thread 30, loop=1
...
```

Thread Creation

```
int main(...) {
     pthread_attr_t attr;
     pthread_t tid;
     int my_arg = 42;
       /* Initialize attr with default values */
     pthread_attr_init(&attr);
       /* create thread... */
     pthread_create(&tid, &attr, my_fun, (void *)&my_arg);
   ...
```

NEUTRINO RTOS

Basic Thread Synchronisation

```
#include <pthread.h>

int pthread_join(pthread_t thread, void** value_ptr);
```

The pthread_join() function blocks the calling thread until the target thread terminates, unless thread has already terminated.

If value_ptr is non-NULL and pthread_join() returns successfully, then the value passed to pthread_exit() by the target thread is placed in value_ptr. If the target thread has been canceled then value_ptr is set to PTHREAD_CANCELED.

The target thread must be joinable. Multiple pthread_join(), calls on the same target thread aren't allowed. When pthread_join() returns successfully, the target thread has been terminated.

Basic Thread Synchronisation

```
void* function(void* arg) {
  for (i = 0, i < 5, i++) {
    printf("This is thread %d, loop=%d\n",
           pthread_self(), i);
    sleep(1);
  }
  return NULL;
}

int main(void) {
  pthread_t tid[3];
  for (i=0, i < 3, i++)
    pthread_create(&tid[i], NULL, function, NULL);

  for (i=0, i < 3, i++)
    pthread_join(tid[i], NULL);

  return EXIT_SUCCESS;
}
```

NEUTRINO RTOS

Thread-Specific Storage

```
#include <pthread.h>
int pthread_key_create( pthread_key_t * key,
                        void (*destructor)(void *));

int pthread_setspecific(pthread_key_t key,
                        const void* value);

void* pthread_getspecific( pthread_key_t key );
```

pthread_key_create() creates a thread-specific data key that's available to all threads in the process and binds an optional destructor function destructor to the key.

Although the same key may be used by different threads, the values bound to the key using pthread_setspecific() are maintained on a per-thread basis.

NEUTRINO RTOS

Real-Time Applications on QNX

- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- **Thread CPU Scheduling**
- POSIX IPC Services
- QNX Neutrino IPC Service
- Clocks and Timers

Thread Scheduling

Thread execution...

- is temporarily suspended whenever the microkernel is entered as the result of
  - a kernel call,
  - exception,
  - hardware interrupt.

A scheduling decision is made whenever the execution state of any thread changes.

- Threads are scheduled globally across all processes.

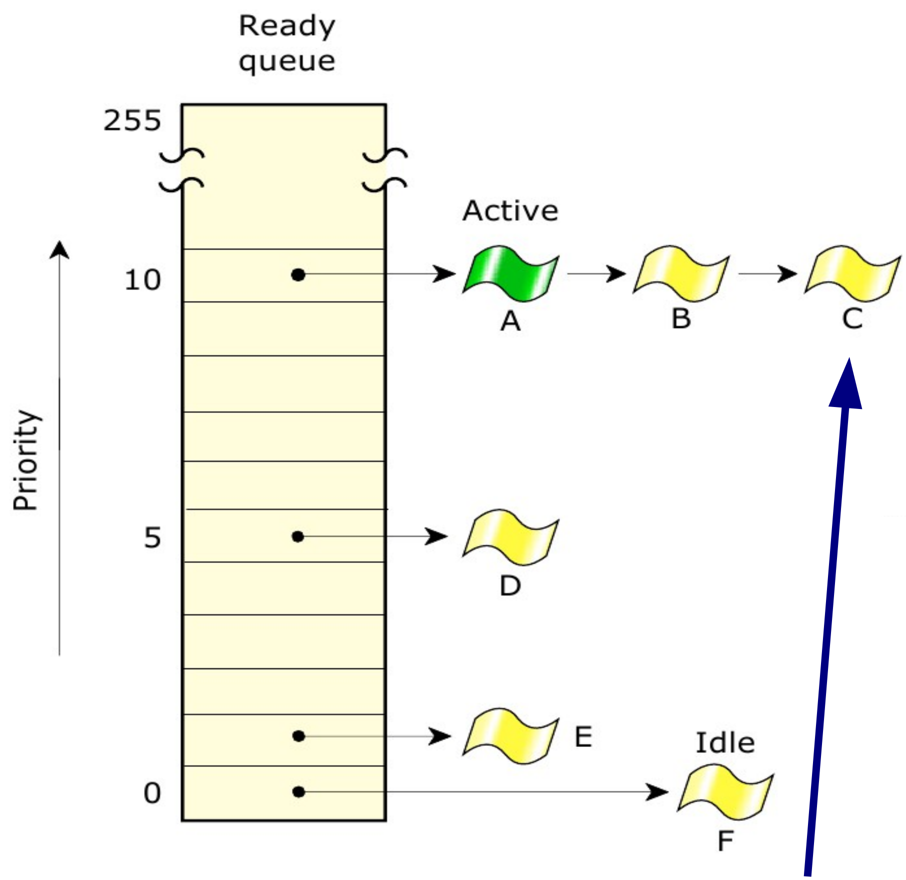
Thread Scheduling

The scheduler will perform a context switch from one thread to another whenever the running thread

- is blocked
  - By calling a blocking system call (e.g. pthread_join())
- is pre-empted
  - Due to an interrupt.
  - A higher priority thread becomes READY.
- yields
  - By calling sched_yield()

```
#include <sched.h>

int sched_yield( void );
```

Thread Priority

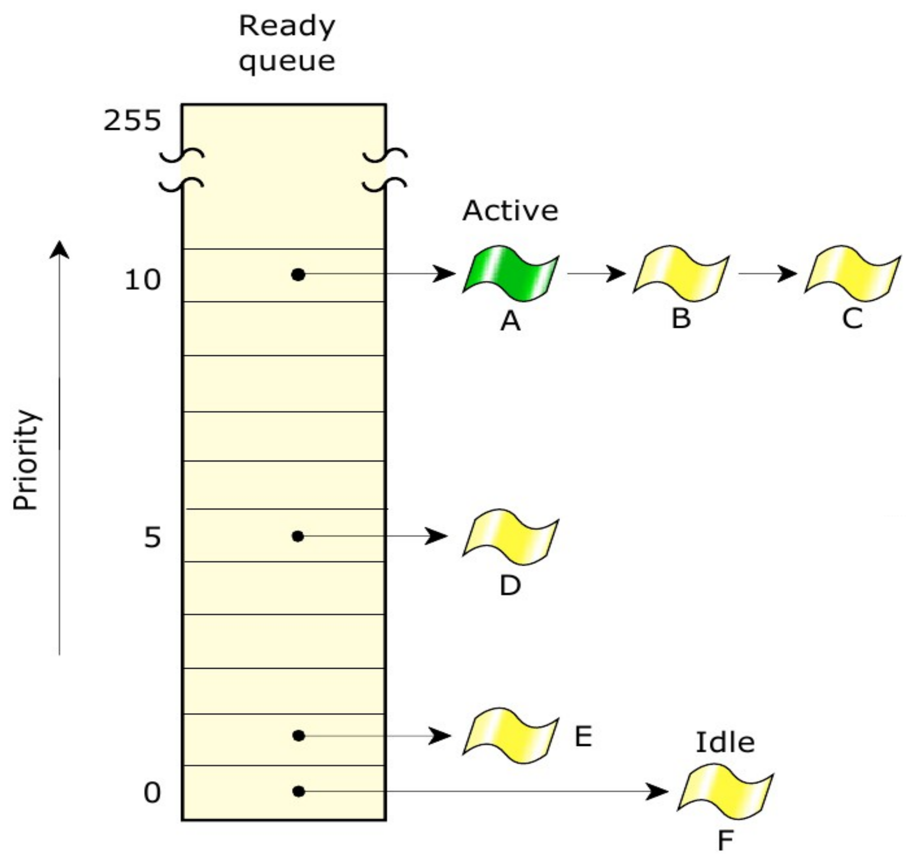


When a thread becomes READY, it is placed at the end of the queue for its assigned priority
(except when a server thread receives a message and comes out of a RECEIVE-Blocked state, in which case it is placed at the head of the queue)

Every thread has an execution priority

- Determines the order by which threads are chosen for execution by the CPU.
- Fixed number from 0 (lowest) to 255 (highest)
  - 0: idle thread
  - 1-63: non-root threads
  - 1-255: root threads
- Multiple threads may have the same priority...

Thread Priority



Note that:

- If a thread never blocks nor yields, then all lower priority threads never get a chance to execute! (starvation)

- Scheduling between threads of the same priority:
  - FIFO scheduling
  - Round-Robin scheduling
  - Sporadic Scheduling

  The algorithm used will be defined by the scheduling algorithm chosen by the running thread.

This is a thread-specific parameter!

NEUTRINO RTOS

Thread Scheduling

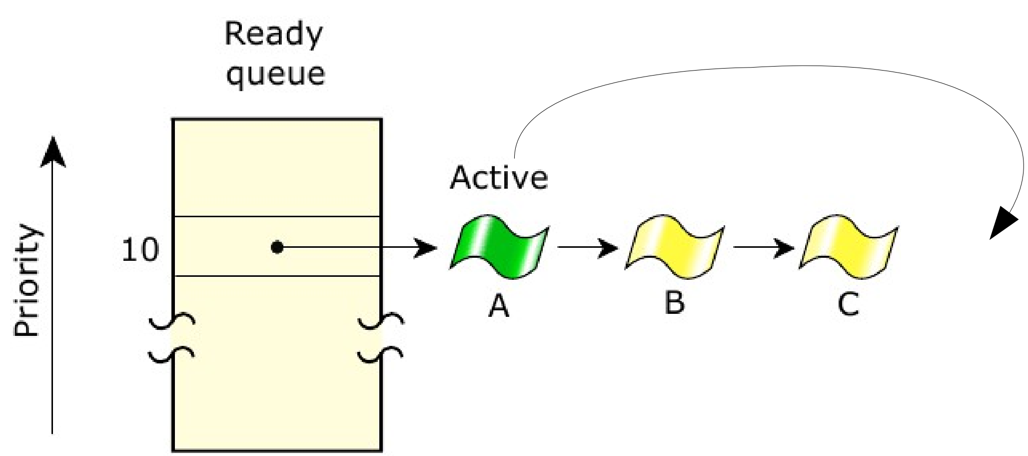
FIFO

A thread runs until it voluntarily relinquishes control of the CPU

(i.e. blocks or yields)

Round-Robin

A thread runs until
- it voluntarily relinquishes control of the CPU (i.e. blocks or yields), or
- it exhausts its time-slice



NOTE: A time-slice is 4x the clock period, which may be changed by ClockPeriod().

Remember: Whatever the chosen algorithm, a thread may always be pre-empted by a higher priority thread!

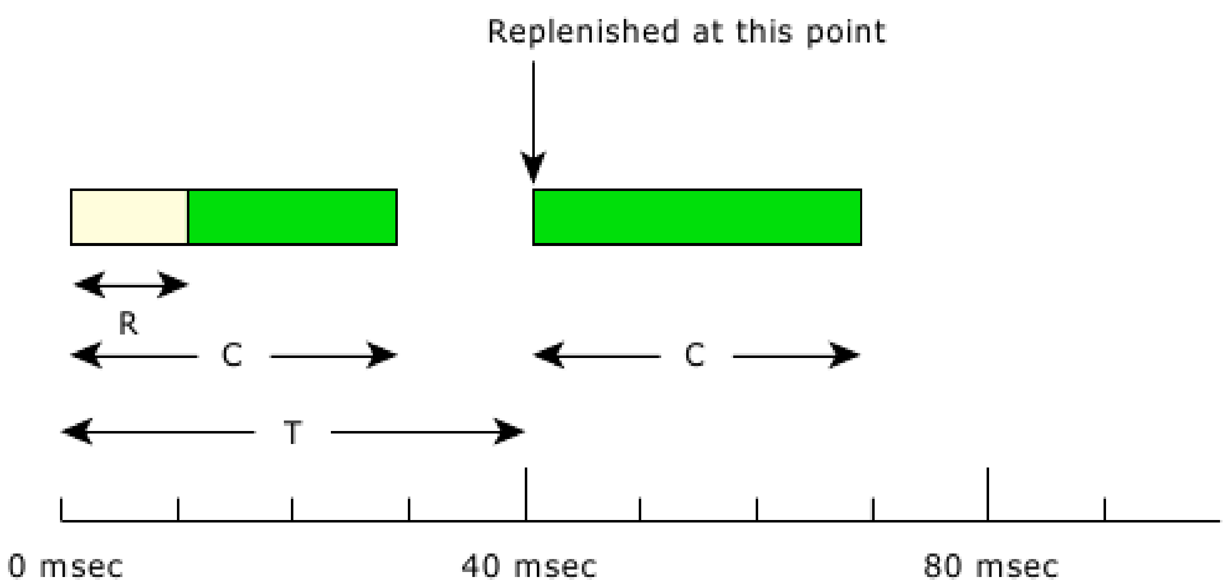
Sporadic Scheduling

A thread has two priorities:

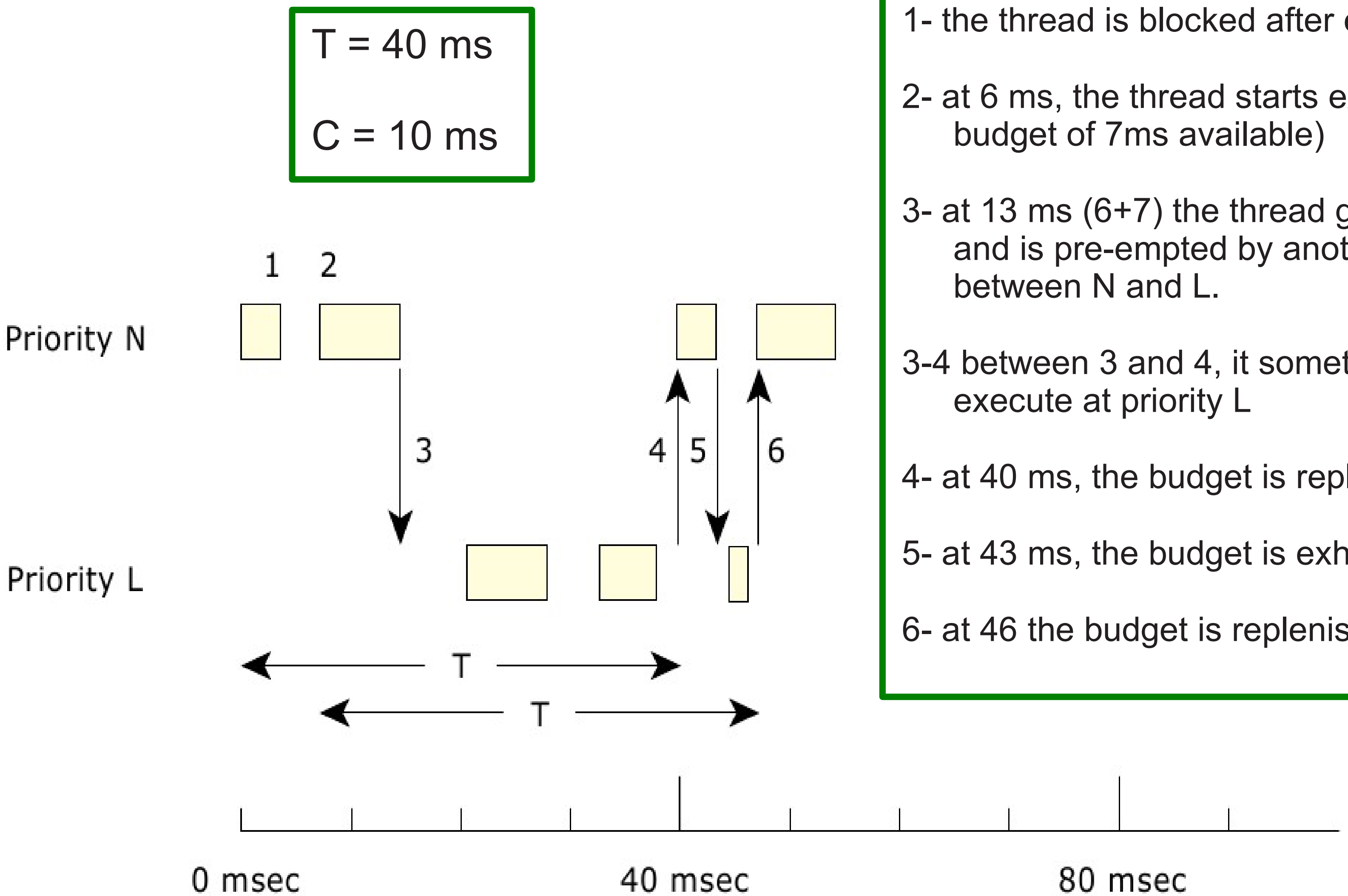
- N: normal priority
- L: low priority

For any sliding time interval T, the thread will execute:

- at N for C time units
- at L if C is exhausted



Sporadic Scheduling



1- the thread is blocked after executing for 3 ms

2- at 6 ms, the thread starts executing again (still has budget of 7ms available)

3- at 13 ms (6+7) the thread goes to reduced priority L, and is pre-empted by another thread with priority between N and L.

3-4 between 3 and 4, it sometimes get an opportunity to execute at priority L

4- at 40 ms, the budget is replenished for 3 ms

5- at 43 ms, the budget is exhausted

6- at 46 the budget is replenished by 7 ms.

Thread Scheduling

```
#include <sched.h>

struct sched_param {
    int32_t sched_priority;
    int32_t sched_curpriority;
    union {
          int32_t reserved[8];
          struct {
                int32_t __ss_low_priority;
                int32_t __ss_max_repl;
                struct timespec __ss_repl_period;
                struct timespec __ss_init_budget;
          } __ss;
    } __ss_un;
}
#define sched_ss_low_priority __ss_un.__ss.__ss_low_priority
#define sched_ss_max_repl     __ss_un.__ss.__ss_max_repl
#define sched_ss_repl_period  __ss_un.__ss.__ss_repl_period
#define sched_ss_init_budget  __ss_un.__ss.__ss_init_budget
```

Thread Scheduling

NEUTRINO RTOS

```
#include <sched.h>
pthread_attr_getschedparam()
pthread_getschedparam()

pthread_attr_setschedparam()
pthread_setschedparam()
```

sched_setscheduler,
sched_setparam → sets the scheduling parameters for thread 1 in the process pid
or the calling thread, if pid is zero
(this detail is not POSIX compliant)

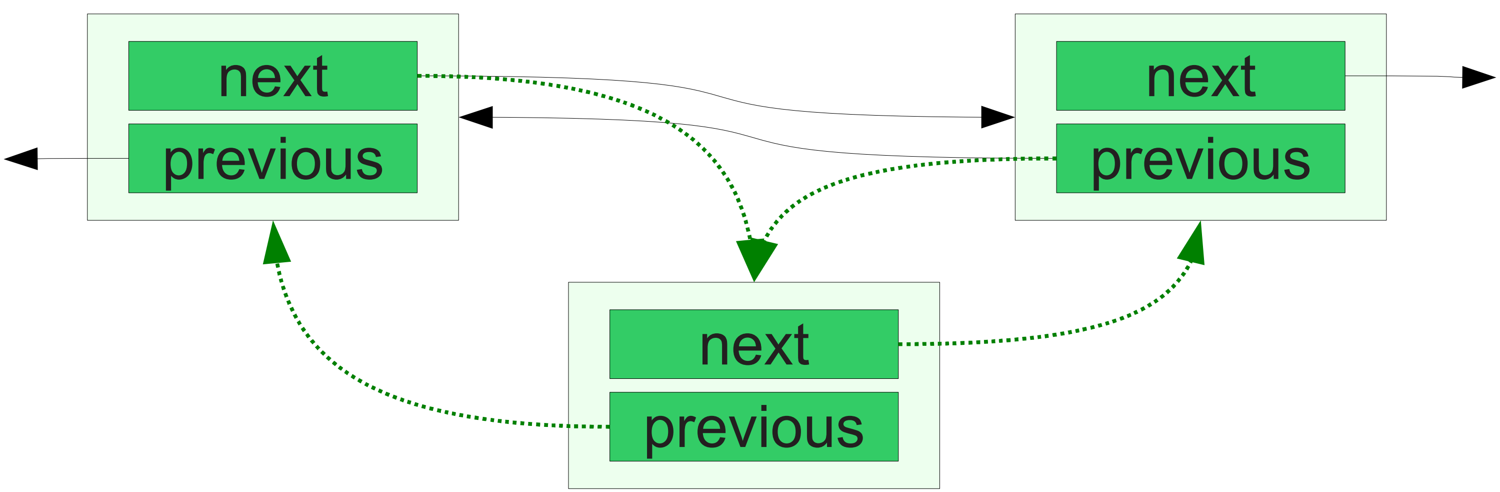
policy → SCHED_FIFO, SCHED_RR, SCHED_OTHER (== SCHED_RR),
SCHED_SPORADIC.

struct sched_param →

Concurrency Issues

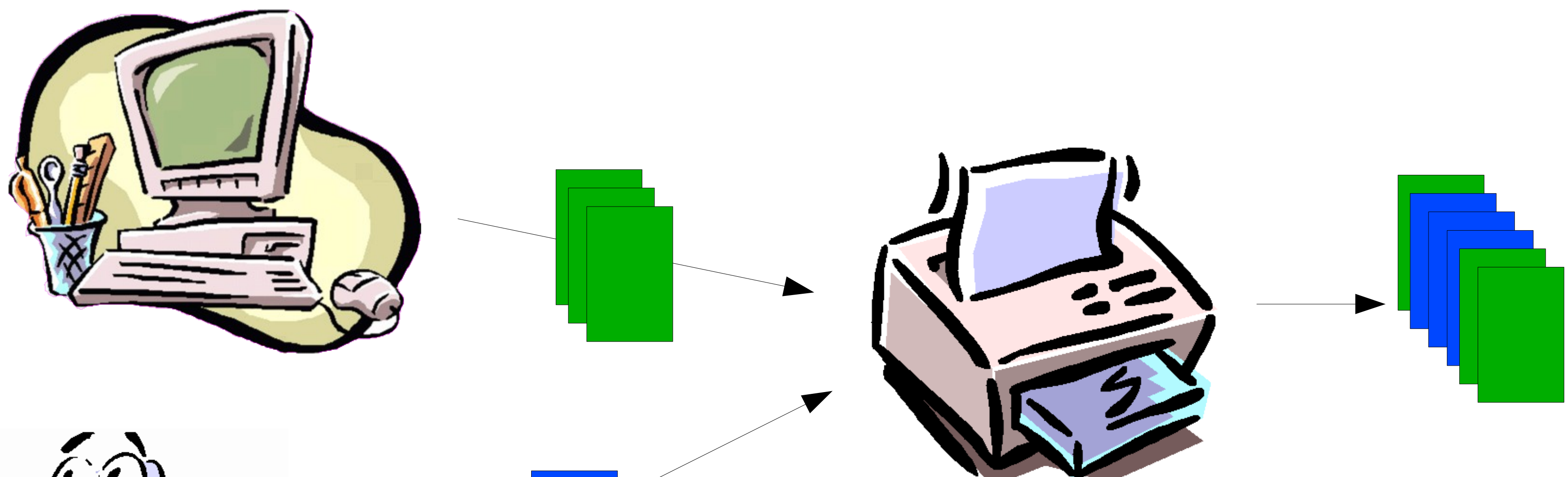
Although the shared memory between threads makes it simple for the threads to share common data (i.e., shared application state), access to this shared data must usually by synchronised...

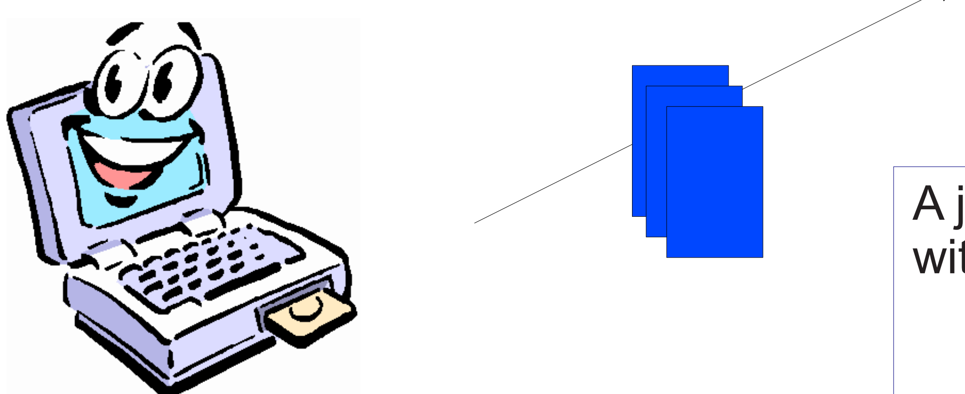
If thread A reads shared state, while thread B is still updating that state, then thread A may read inconsistent data.



Concurrency Issues

A more mundane example...
...printing jobs on a shared printer.



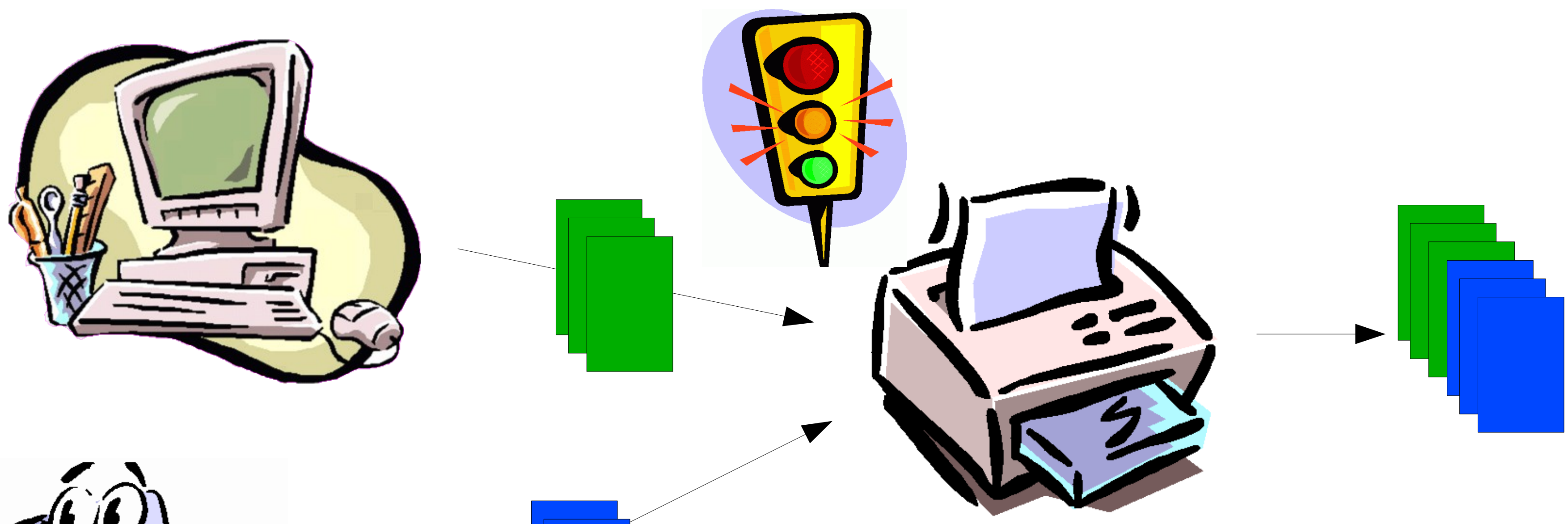


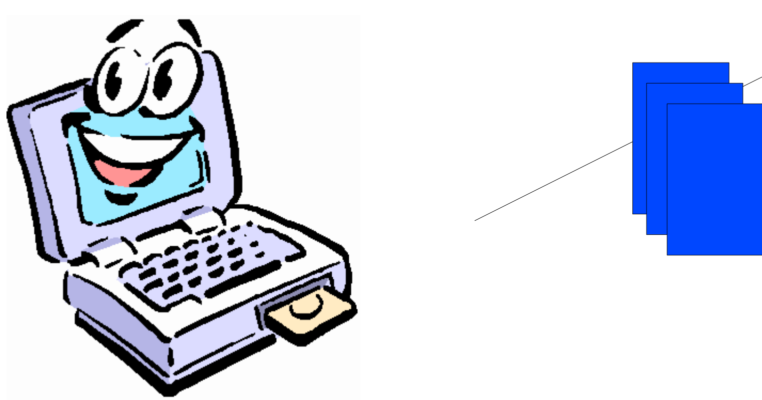
A job (or block of code) that must execute without interruption is called a

critical section

Concurrency Issues

Unless a synchronisation mechanism is used, the program may fail intermittently
→ VERY difficult to debug

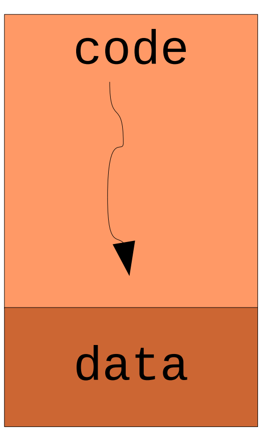




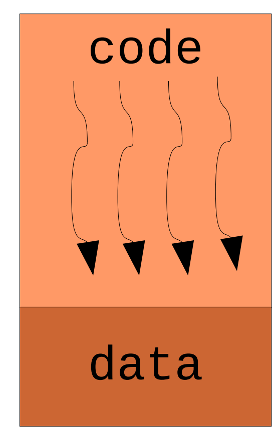
The OS provides abstract entities that implement the functionality of the traffic light...
… semaphores, mutex, condition variable, etc.

Data Sharing Issues

- Threads within a process share everything within the process's address space →
- ...easy to share data among threads in the same process
- How do we share data among threads in distinct processes?
  - Pipes
  - Sockets
  - Shared Memory
  - Message Passing



Single threaded process



Multi threaded process

Threads vs Processes

Traditionally, context switching between processes is relatively slow, so threads were created as a kind of 'light weight process', without memory protection.

In QNX Neutrino, threads and processes provide nearly identical context-switch performance.

The choice to use processes vs. threads will depend only on the type of application being designed.

Threads:

Concurrent applications which share a large amount of data between concurrent execution threads (e.g. filesystem driver).

Processes:

Concurrent applications which share a small amount of data between concurrent execution threads .

QNX NEUTRINO RTOS

Synchronisation Services

Synchronization service	Supported between processes	Supported across a QNX LAN
Mutexes	Yes	No
Condvars	Yes	No
Barriers	No	No
Sleepon locks	No	No
Reader/writer locks	Yes	No
Semaphores	Yes	Yes (named only)
FIFO scheduling	Yes	No
Send/Receive/Reply	Yes	Yes
Atomic operations	Yes	No

NEUTRINO RTOS

Real-Time Applications on QNX

- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- Thread CPU Scheduling
- **POSIX IPC Services**
- QNX Neutrino IPC Service
- Clocks and Timers

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(   pthread_mutex_t* mutex );
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

Only one thread may have the mutex locked at any given time.

Threads attempting to lock an already locked mutex will block until the thread that owns the mutex unlocks it.

When the thread unlocks the mutex, the highest-priority thread waiting to lock the mutex will unblock and become the new owner of the mutex.

Threads will sequence through a critical region in priority-order.

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(   pthread_mutex_t* mutex );
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

Entry to the kernel is done at acquisition time only if the mutex is already held so that the thread can go on a blocked list;

Kernel entry is done on exit only if other threads are waiting to be unblocked on that mutex.

This allows acquisition and release of an uncontested critical section or resource to be very quick.

This is achieved by using compare-and-swap opcode on x86 processors and the load/store conditional opcodes on most RISC processors.

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(   pthread_mutex_t* mutex );
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

Mutex: MUTual EXclusion lock

pthread_mutex_t : the data type of a mutex

- an opaque data type (we never get to use it directly)

pthread_mutex_lock() → lock the mutex

- calling thread blocks until the mutex becomes available

pthread_mutex_unlock() → unlock the mutex

NEUTRINO RTOS

Mutexes

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t* mutex );
```

- pthread_mutex_trylock() → lock the mutex
  - calling thread never blocks.
  - Returns EOK if success (the calling thread locked the mutex)
  - Return EBUSY if the mutex was already locked.

- Useful when the thread has alternative means of achieving objective without the required resource.

Mutexes

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(
                pthread_mutex_t * mutex,
                const struct timespec * abs_timeout );
```

pthread_mutex_timedlock() → lock the mutex

- calling never blocks for a specified maximum time!
- Returns EOK if success
  (the calling thread locked the mutex)
- Return EBUSY if the mutex was already locked.

QNX NEUTRINO RTOS

Mutexes

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t* mutex );
```

pthread_mutex_trylock() → lock the mutex

- calling thread never blocks.
- Returns EOK if success (the calling thread locked the mutex)
- Return EBUSY if the mutex was already locked.

Useful when the thread has alternative means of achieving objective without the required resource.

Mutexes

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init( pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* attr );

int pthread_mutex_destroy( pthread_mutex_t* mutex );
```

A mutex must be initialised before being used.

- **`PTHREAD_MUTEX_INITIALIZER`**: use default attributes
- **`attr`**: use these attributes for the mutex (may ne NULL)

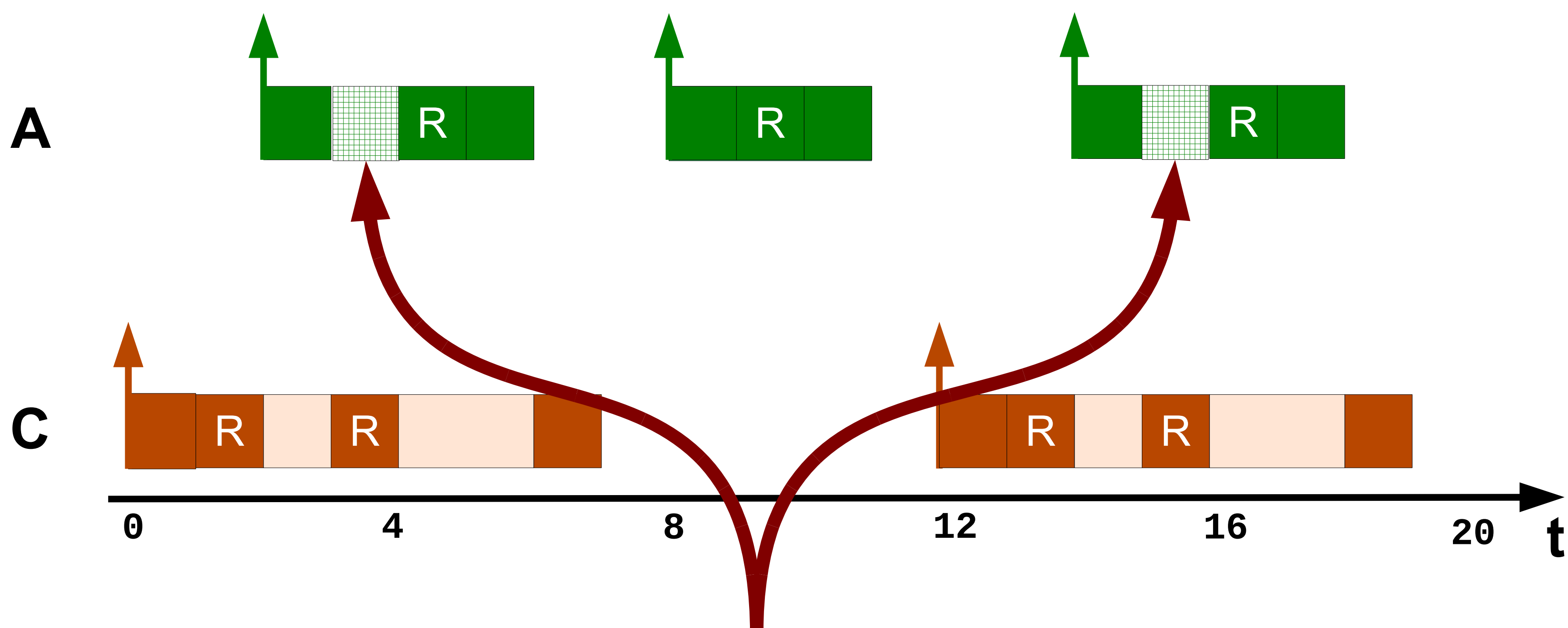
A mutex may be destroyed if no longer required

Priority Inversion

Example:

R- resource shared by A and C

Process	Period (*T*)	Deadline (D)	Execution Time (C)	Execution Sequence	Priority
A	6	6	3	ERE	3
C	12	12	4	ERRE	1



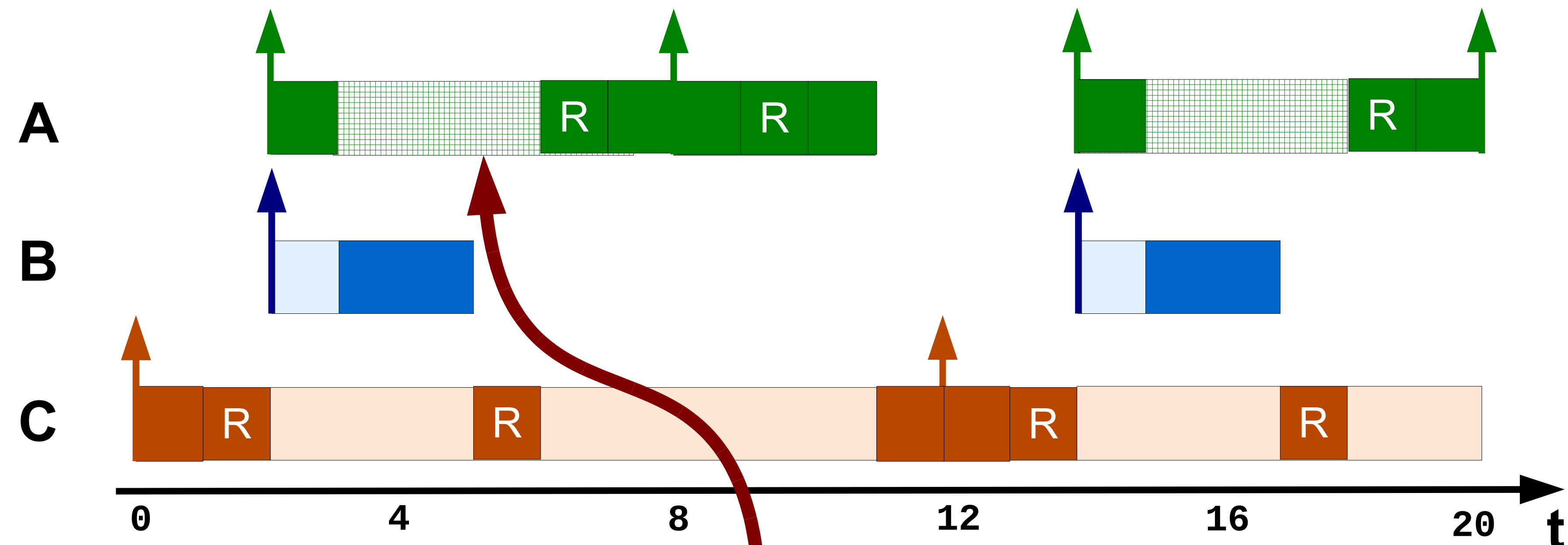
PRIORITY iNVERSION: a high priority thread will not be able to execute because it needs access to a resource which is currently being used by a lower priority thread!

Unbounded Priority Inversion

Example:

R- resource shared by A and C

Process	Period (*T*)	Deadline (D)	Execution Time (C)	Execution Sequence	Priority
A	6	6	3	ERE	3
B	12	12	2	EE	2
C	12	12	4	ERRE	1



UNBOUNDED PRIORITY iNVERSION: the high priority thread may be blocked for a time period larger than the time required to release the resource!

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(   pthread_mutex_t* mutex );
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

In order to BOUND priority inversion, by default mutexes implement the (simple) Priority Inheritance Protocol.

if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex.

The current owner returns to its real priority when it unlocks the mutex.

Mutexes

NEUTRINO RTOS

```
#include <pthread.h>

int pthread_mutex_init( pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* attr );

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

`pthread_mutexattr_init():`

A Initializes mutex attribute variable with default attributes

- protocol → PTHREAD_PRIO_INHERIT
  (simple PIP protocol)
- recursive → PTHREAD_RECURSIVE_DISABLE
  (NOTE: recursive locking is not POSIC compliant)

NEUTRINO RTOS

Mutexes

```
#include <pthread.h>

pthread_mutexattr_setprotocol()
pthread_mutexattr_setprioceiling()
pthread_mutexattr_setpshared()
pthread_mutexattr_setrecursive()
pthread_mutexattr_settype()

pthread_mutexattr_getprioceiling()
...
```

pthread_mutexattr_setprotocol()

- PTHREAD_PRIO_INHERIT → simple Priority Inheritance Protocol
- PTHREAD_PRIO_PROTECT → immediate Priority Ceiling Protocol
  - execute the thread at the highest priority or priority ceilings of all the mutexes owned by the thread and initialized with PTHREAD_PRIO_PROTECT, whether other threads are blocked or not.
- The POSIX protocol of PTHREAD_PRIO_NONE isn't currently supported.

Real-Time Applications on QNX

NEUTRINO RTOS

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Condition Variables

```
#include <pthread.h>

int pthread_cond_wait(...)
int pthread_cond_signal(...)
int pthread_cond_broadcast(...)
```

Used when we want a thread to wait until another thread tells it to proceed.

The condition variable must be used in connection with a mutex.

pthread_cond_wait() → wait for a signal
pthread_cond_signal() → release one waiting thread
(The longest waiting highest priority thread!)
pthread_cond_broadcast() → release all waiting thread
(normal scheduling results in the highest priority thread running first!)

NEUTRINO RTOS

Condition Variables

```
pthread_mutex_lock( &mutex );
while (!arbitrary_condition)
  pthread_cond_wait( &condvar, &mutex );
...
pthread_mutex_unlock( &mutex );
```

We must lock the mutex before calling `pthread_cond_wait()`

The mutex is released while the thread remains blocked on `pthread_cond_wait()`

The mutex is relocked before returning from `pthread_cond_wait()`

The while() loop is necessary because POSIX does not guarantee that false wakeups will not occur (e.g. process received a POSIX signal <> condvar signal!).

NEUTRINO RTOS

Condition Variables

```
#include <pthread.h>

int pthread_cond_signal   ( pthread_cond_t*  cond );
int pthread_cond_broadcast( 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,
                    const struct timespec* abstime );
```

The thread can specify until when it is willing to wait on the condition variable (absolute time!).

The return value will indicate whether it received a signal (EOK) or whether it timed out (ETIMEDOUT)

Priority inheritance protocol used will depend of mutex attributes.

Condition Variables

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init( pthread_cond_t* cond,
                       pthread_condattr_t* attr );

int pthread_condattr_init( pthread_condattr_t* attr);
int pthread_condattr_setclock(...)    getclock(...)
int pthread_condattr_setpshared(...)  getpshared(...)
```

A condition variable must be initialized before first use.

We can specify which clock to use to determine the time out in timedwait.

The condition variable may be set to allow access from threads in distinct processes (pshared).

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

NEUTRINO RTOS

Sleepon Locks

```
#include <pthread.h>
int pthread_sleepon_lock(void)
int pthread_sleepon_unlock(void)

int pthread_sleepon_wait      (const void *addr)
int pthread_sleepon_ timedwait(const void *addr, ...)
int pthread_sleepon_signal    (const void *addr)
int pthread_sleepon_broadcast (const void *addr)
```

Similar to condition variables, but intend to be simpler to use:

- always uses a default mutex,
- and wait on a simple void *ptr.

These are NOT POSIX compliant functions. They are QNX specific.

These functions are implemented as library functions based on the condition variable services.

Sleepon Locks

```
// Thread 1
pthread_sleepon_lock();
while (void_ptr)
  pthread_sleepon_wait(void_ptr);
...
pthread_sleepon_unlock();
```

```
// Thread 2
pthread_sleepon_lock();
void_ptr = 1;
pthread_sleepon_signal(void_ptr);
pthread_sleepon_unlock();
```

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Barriers

```
#include <pthread.h>

int pthread_barrier_init(
                   pthread_barrier_t *barrier,
                   const pthread_barrierattr_t *attr,
                   unsigned int count);

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

A barrier is a synchronization mechanism that allows threads to wait for each other at a rendezvous point. Once a specified number of threads have reached this rendezvous point, all threads are released simultaneously.

A call to pthread_barrier_wait() will block the calling thread. When the correct number of threads have called this function, all those threads will unblock at the same time.

Barriers

```
#include <pthread.h>

int pthread_barrierattr_init(
                       pthread_barrierattr_t * attr );

int pthread_barrierattr_setpshared(...);

int pthread_barrierattr_getpshared(...);
```

Like a condition variable, a barrier may be set to allow access from threads in distinct processes (pshared).

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Reader/Writer Locks

```
#include <pthread.h>

int pthread_rwlock_rdlock( pthread_rwlock_t* rwl );
int pthread_rwlock_wrlock( pthread_rwlock_t* rwl );
int pthread_rwlock_unlock( pthread_rwlock_t* rwl );
```

Commonly known as “Multiple readers, single writer locks,” these locks are used when many threads may simultaneously read the data, and (at most) one thread may write the data.

More expensive than mutexes, but are sometimes handy.

Implemented as library routines, so priority inversion may occur!

Avoids reader and writer starvation by:
- favouring blocked readers over writers after a writer has just released an exclusive lock.

Reader/Writer Locks

```
#include <pthread.h>

int pthread_rwlock_rdlock(...);
int pthread_rwlock_tryrdlock(...);
int pthread_rwlock_timedrdlock(...);

int pthread_rwlock_rdlock(...);
int pthread_rwlock_tryrdlock(...);
int pthread_rwlock_timedrdlock(...);
```

Timeouts in timed versions are Absolute, and not relative!

Reader/Writer Locks

```
#include <pthread.h>

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_init( pthread_rwlock_t * rwl,
                         const pthread_rwlockattr_t * attr );

int pthread_rwlockattr_init(pthread_rwlockattr_t* a);
int pthread_rwlockattr_getpshared(...);
int pthread_rwlockattr_setpshared(...);
```

Like a mutex, a Reader/Writer Lock may be set to allow access from threads in distinct processes (pshared).

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Semaphores

```
#include <semaphore.h>

int sem_post( sem_t * sem );
int sem_wait( sem_t * sem );
int sem_trywait(   sem_t * sem );
int sem_timedwait( sem_t * sem, ... *abs_timeout);
int sem_getvalue ( sem_t * sem, int * value );
```

Similar to mutexes, but with a counter...

sem_post() increments the semaphore

sem_wait() decrements the semaphore.
This call will block if the semaphore is already at zero, and later return after some other thread does a sem_post()

More expensive than mutexes, but are sometimes handy.

NEUTRINO RTOS

Semaphores

```
...

while (sem_wait(&s) && (errno == EINTR)) {};

/* Semaphore was decremented */
call_critical_region();

sem_post(&s)

...
```

Like condition variables, semaphores can legally return a nonzero value because of a false wakeup (e.g. a POSIX signal is received), correct usage requires a loop.

Semaphores

Semaphores are guaranteed to work across processes (unlike mutexes, which may or may not work across processes, depending on the specific POSIX implementation)

Since QNX Neutrino allows mutexes across processes, use of semaphores is sometimes made just for code portability.

Another difference between semaphores and other synchronization primitives is that semaphores are “async safe” and can be manipulated by signal handlers. If the desired effect is to have a signal handler wake a thread, semaphores are the right choice.

Semaphores don’t affect a thread’s effective priority, i.e. do not support Priority Inheritance Protocols.

Semaphores

```
#include <semaphore.h>

// Unnamed semaphores
int sem_init(sem_t *sem, int pshared, unsigned val);
int sem_destroy( sem_t * sem );

// Named semaphores
sem_t * sem_open( const char * sem_name, int oflags,
                  [mode_t mode, unsigned int val] );
int sem_close( sem_t * sem );
int sem_unlink( const char * sem_name );
```

Semaphores may be named or unnamed. DO NOT MIX!

Named semaphores are identified by a name in the filesystem.

Named semaphores are slower than the unnamed variety.

Real-Time Applications on QNX

...

POSIX IPC Services

- Mutexes
- Condition Variables
- Sleepon locks
- Barriers
- Reader/Writer locks
- Semaphores
- Synchronization via scheduling algorithm

Synchronisation via Scheduling Algorithm

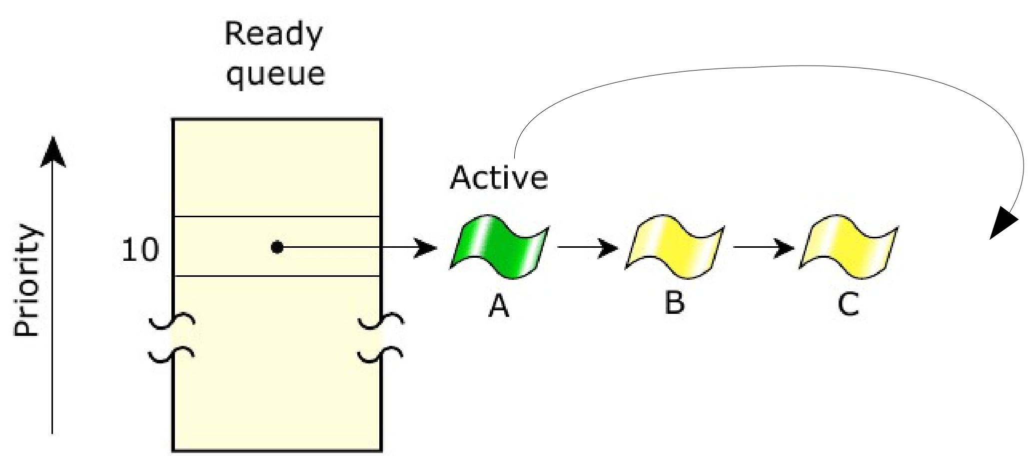
FIFO

A thread runs until it voluntarily relinquishes control of the CPU

(i.e. blocks or yields)

Round-Robin

A thread runs until
- it voluntarily relinquishes control of the CPU (i.e. blocks or yields), or
- it exhausts its time-slice



Remember: Whatever the chosen algorithm, a thread may always be pre-empted by a higher priority thread!

Synchronisation via Scheduling Algorithm

NEUTRINO RTOS

FIFO

A thread runs until it voluntarily relinquishes control of the CPU

(i.e. blocks or yields)

A thread can safely enter a critical section knowing it will never be pre-empted by other threads of the same priority!

The hardware has a single CPU

Remember: Whatever the chosen algorithm, a thread may always be pre-empted by a higher priority thread!

Real-Time Applications on QNX

...

POSIX IPC Services

...

Barriers

Reader/Writer locks

Semaphores

Synchronization via Scheduling Algorithm

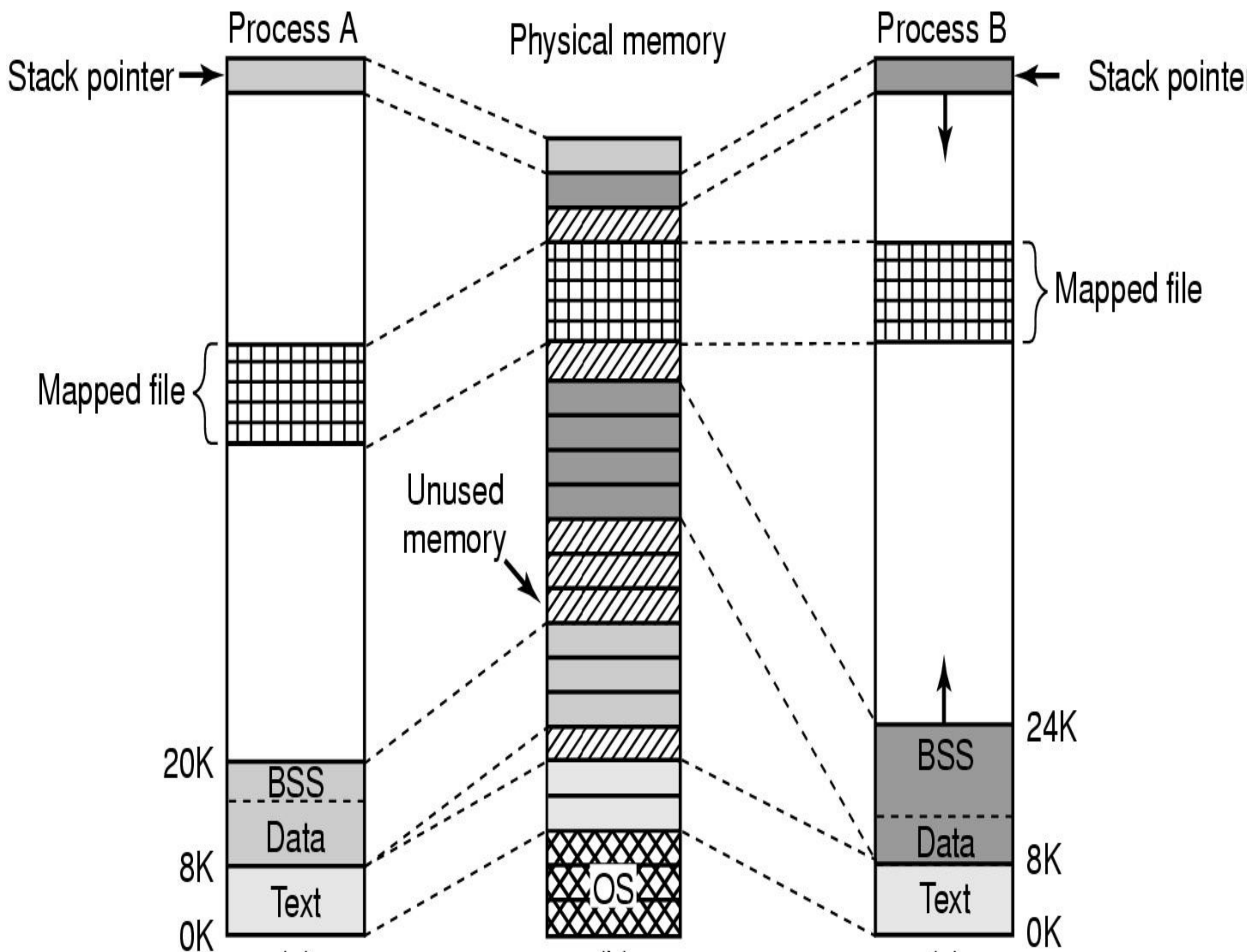
Shared Memory

Shared Memory

Shared memory provides **un-synchronised** data sharing between two or more processes.

Two or more processes obtain pointers to the **same physical memory area**.

Shared memory provides the highest bandwidth IPC available!



NEUTRINO RTOS

Shared Memory

```
#include <fcntl.h>
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode)

void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

Once access to the shared memory area has been established, processes access this memory just like any other memory in its address space (e.g., using pointers...).

Since access is un-synchronised, there is the risk of inconsistent reads and writes → shared memory is usually used in conjunction with another synchronising IPC (e.g. mutexes, semaphores, message passing, etc...).

NEUTRINO RTOS

Shared Memory

```
/* Map in a shared memory region */
fd = shm_open( "/datapoints", O_RDWR|O_CREAT, 0777);
ftruncate(fd, len); // 1st process sets the file size
addr = mmap(NULL, len, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, 0 );

/* To share memory with hardware, eg. video memory */
/* Map in VGA display memory on x86 platform       */
addr = mmap(NULL, 65536, PROT_READ|PROT_WRITE,
            MAP_PHYS|MAP_SHARED, NOFD, 0xa0000);

/* To allocate a DMA buffer for a bus-mastering PCI
 * network card:
 * Allocate a physically contiguous buffer
 */
addr = mmap(NULL, 262144,
            PROT_READ|PROT_WRITE|PROT_NOCACHE,
            MAP_PHYS|MAP_ANON, NOFD, 0);
```

QNX NEUTRINO RTOS

Shared Memory

```
#include <fcntl.h>
#include <sys/mman.h>

shm_open(...);   // Open and/or create shared memory.
Close (...);     // Close access to shared memory.
shm_unlink(...); // Delete shared memory file.

mmap  (...); //map shared memory to local addr. space
Munmap(...); //unmap shared memory from local addr.

mprotect(...); //change protections of shared memory
msync   (...); //synch. memory with physical storage
```

Shared Memory

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

addr : local address on to which we suggest shared memory should be mapped. If NULL, then no suggestion is made.

len : number of bytes to map. On x86, must be a multiple of page size (i.e. 4096).

fildes : file descriptor of file to be mapped to memory, or NOFD when mapping physical memory.

off : map shared memory starting at offset

prot : protections to be applied to memory →

- PROT_EXEC - Memory may be executed.
- PROT_NOCACHE - Memory should not be cached.
- PROT_NONE - No access allowed.
- PROT_READ - Memory may be read.
- PROT_WRITE - Memory may be written.

Shared Memory

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

flags : determine how the memory is mapped →

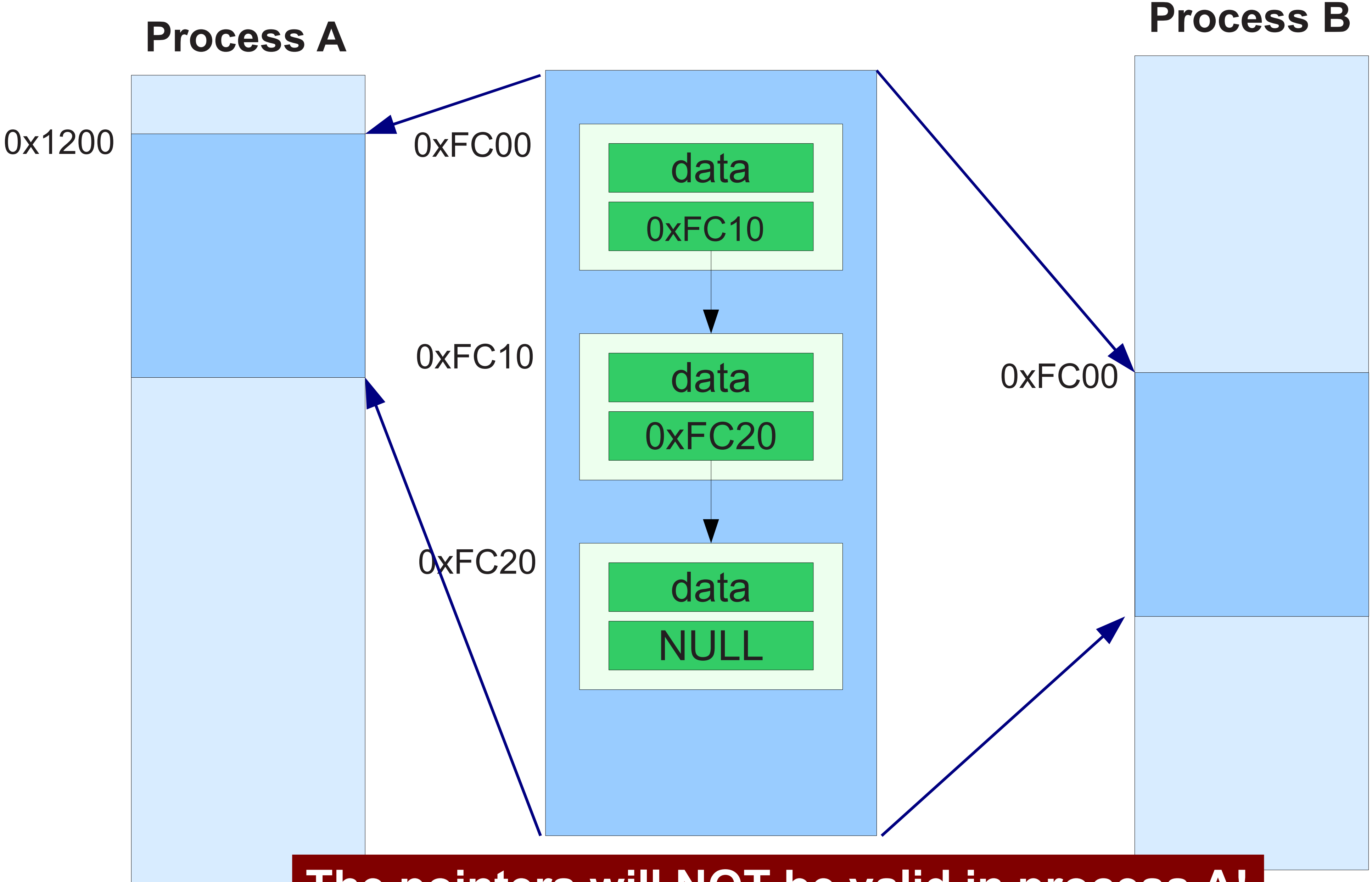
Map type – one of the following must be specified

- MAP_SHARED - The mapping is shared by the calling processes.
- MAP_PRIVATE - The mapping is private to the calling process.
  It allocates system RAM and makes a copy of the object.

Map type – one of the following must be specified

- MAP_ANON - commonly used with MAP_PRIVATE, i.e. not for shared memory; fd must be set to NOFD. May be used as the basis for a page-level memory allocator.
- MAP_FIXED - Map object to the address specified by addr.
- MAP_PHYS - indicates that we wish to deal with physical memory. fd should be set to NOFD. When used with MAP_SHARED, the off specifies the exact physical address to map (e.g. for video frame buffers). If used with MAP_ANON then physically contiguous memory is allocated (e.g. for a DMA buffer).

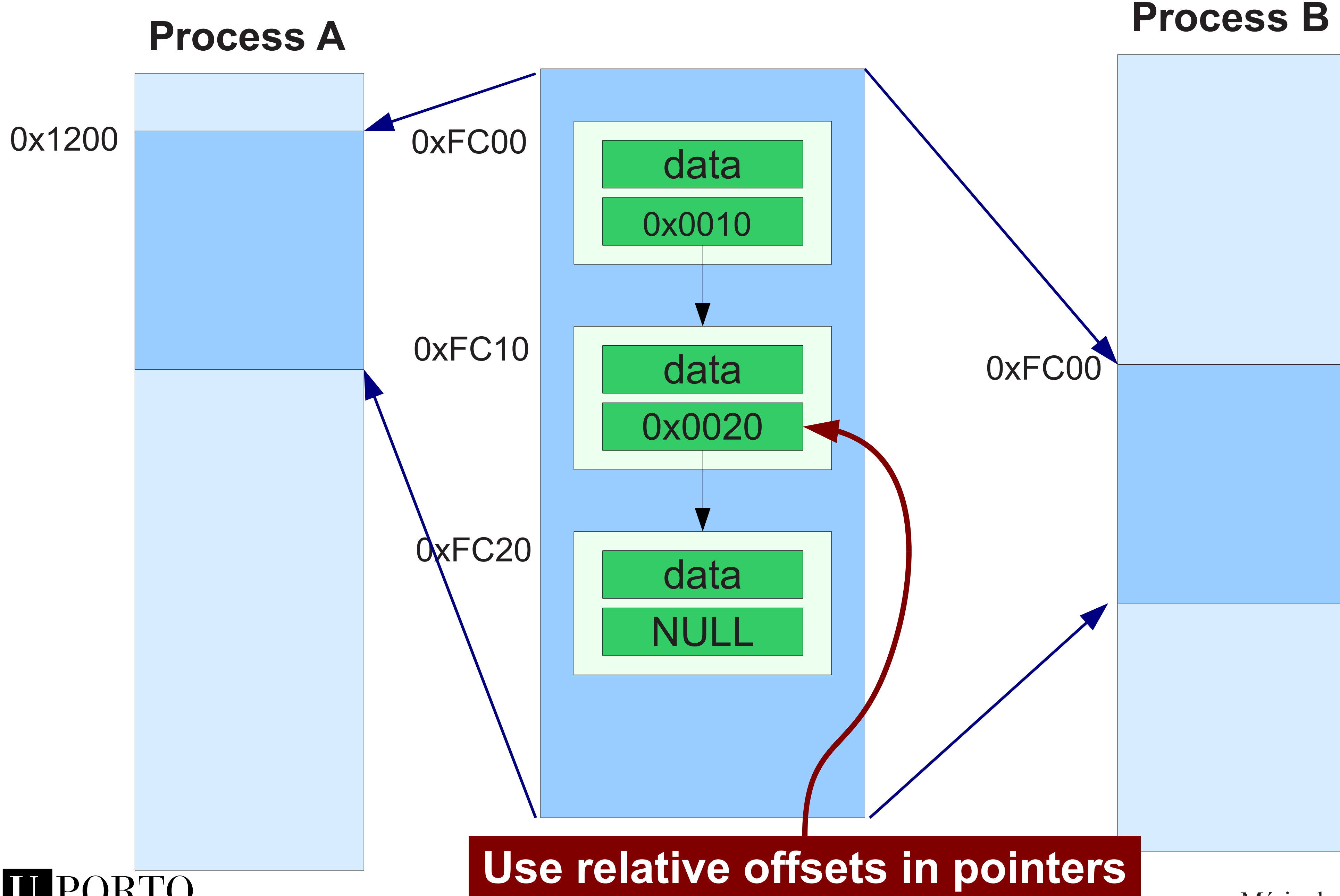
Shared Memory



The pointers will NOT be valid in process A!

NEUTRINO RTOS

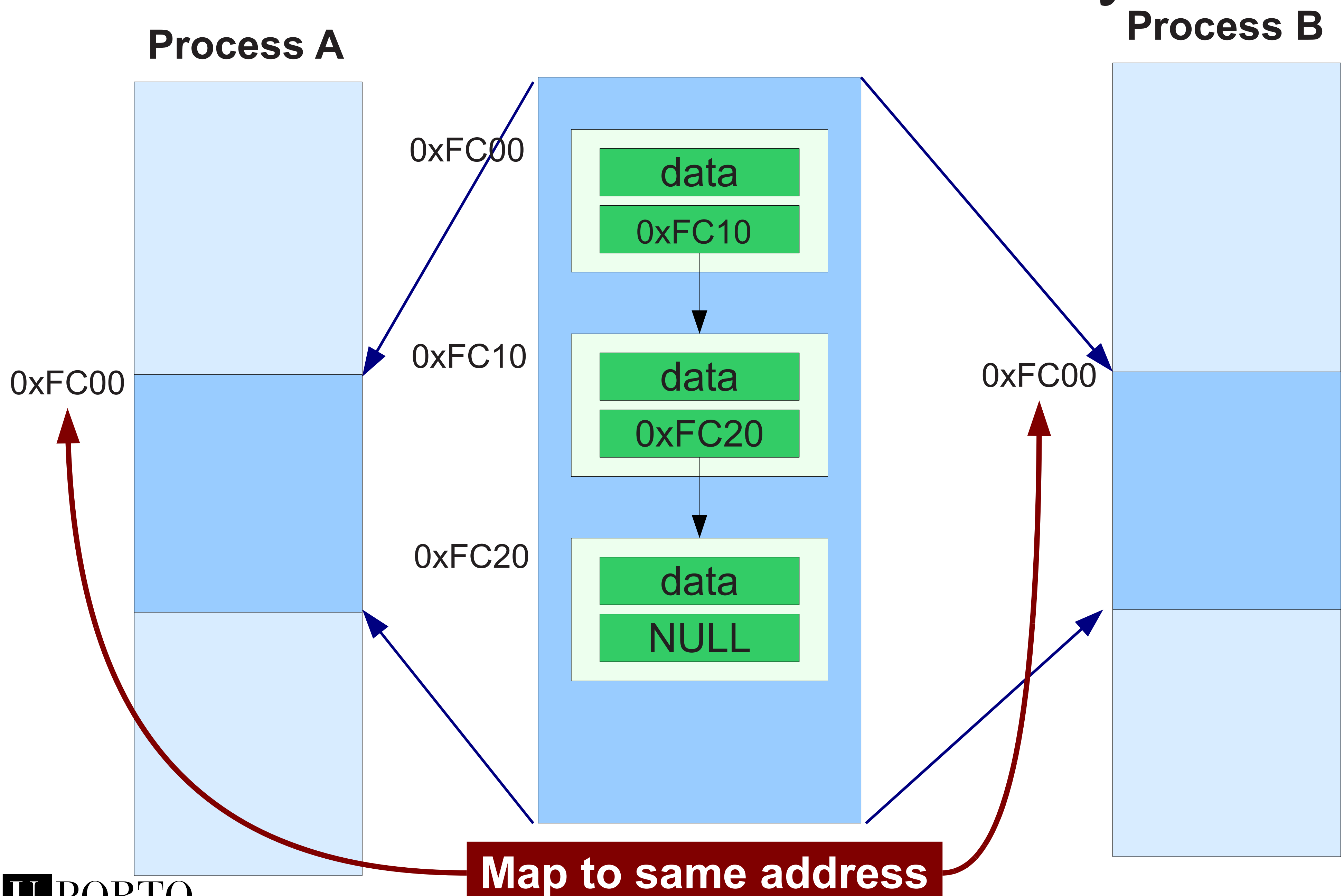
Shared Memory



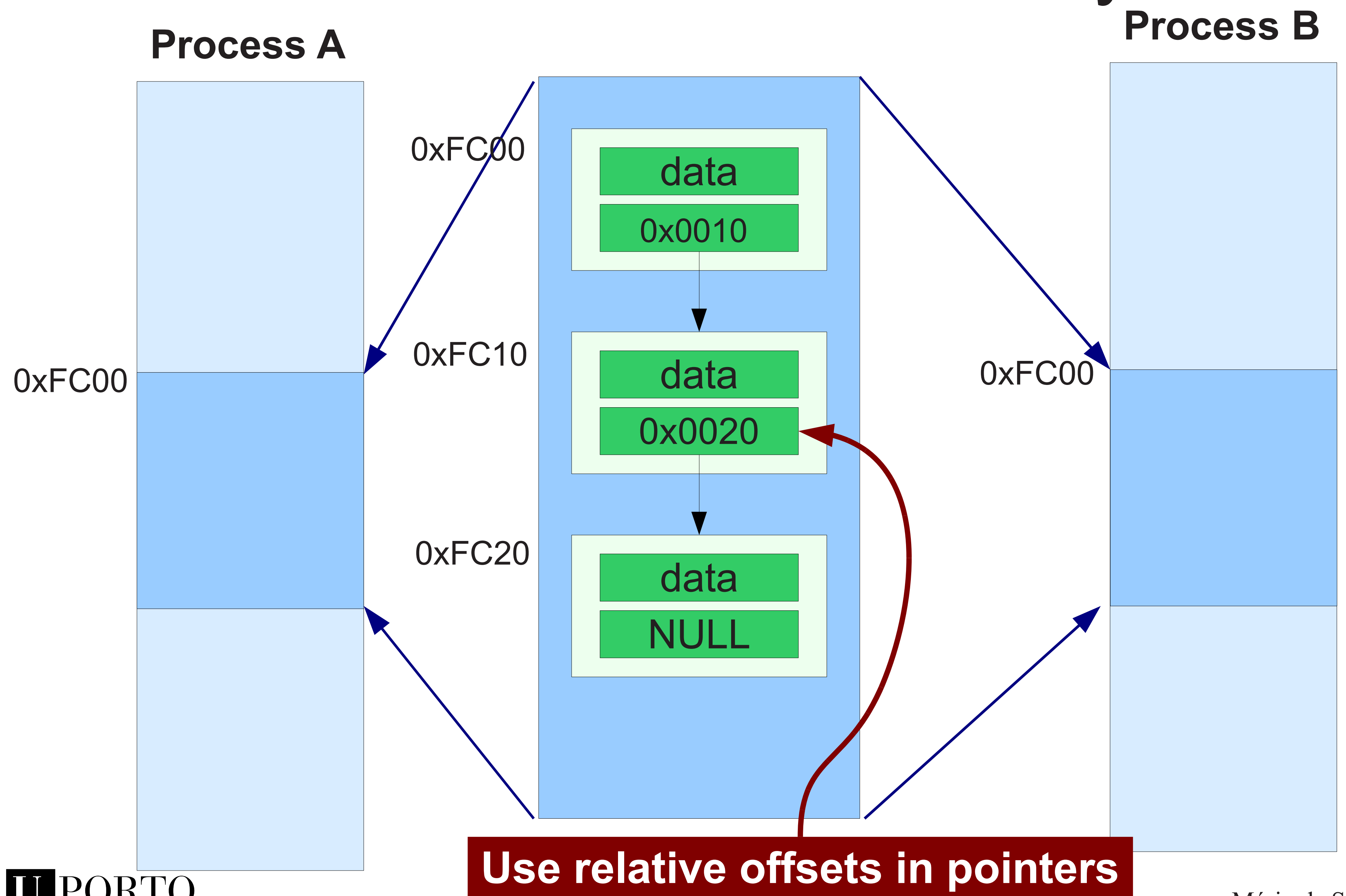
U.PORTO

Mário de Sousa

Shared Memory



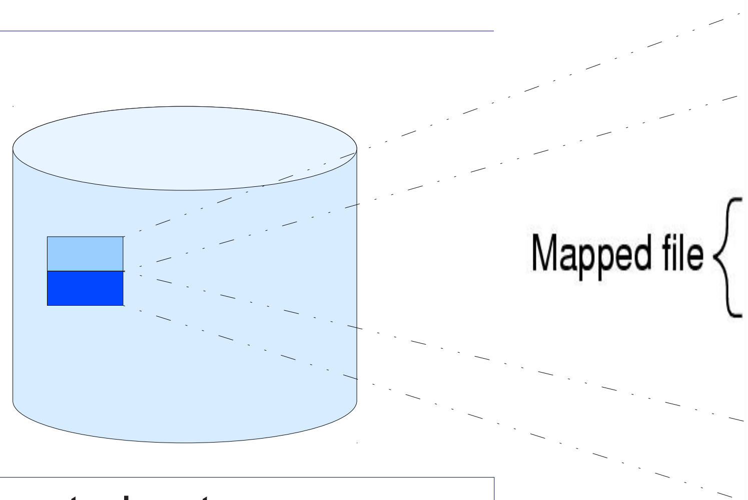
Shared Memory

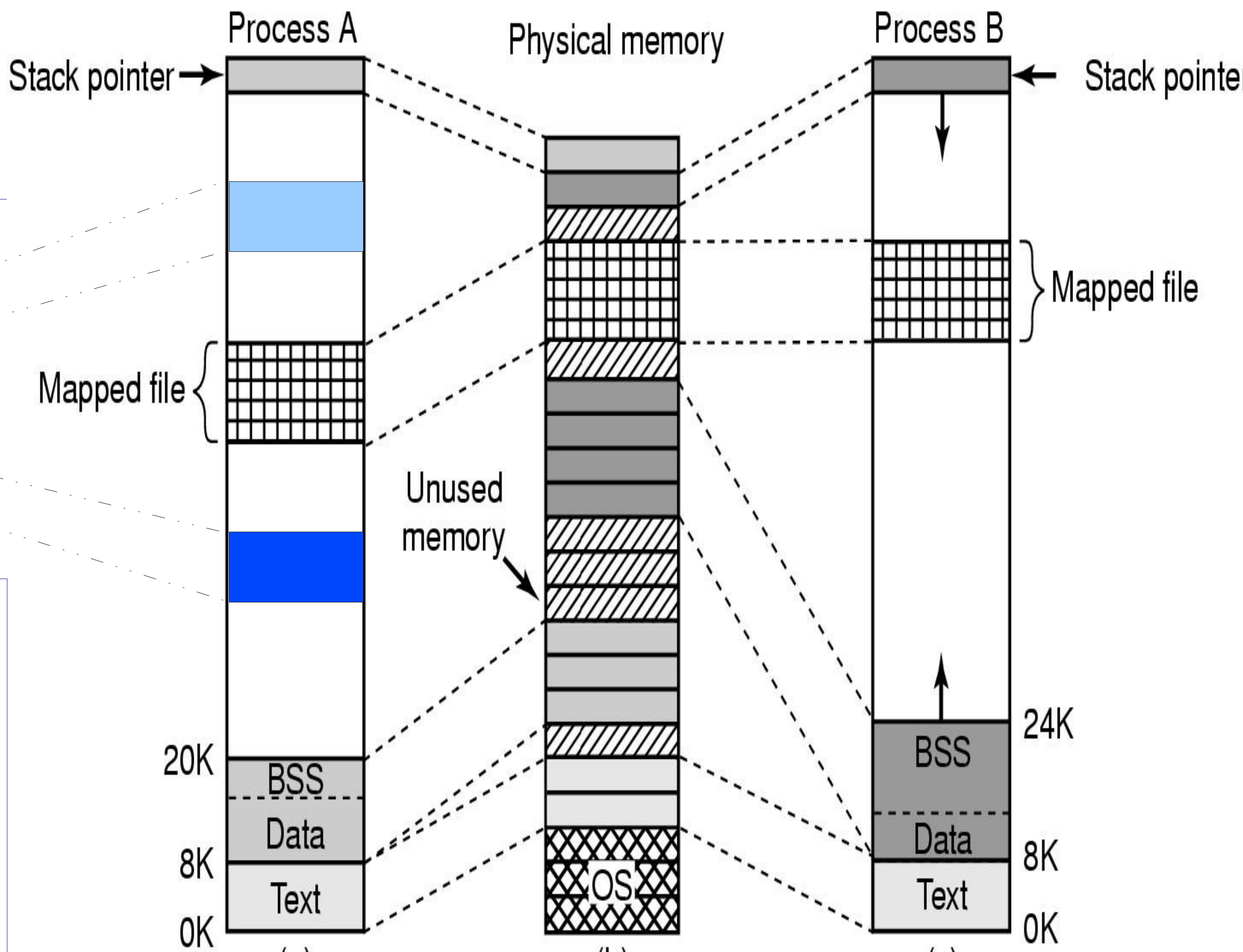


QNX NEUTRINO RTOS

Memory & Paging
(small detour...)

On some systems, a swap area is used on disk to augment the physical RAM.





When trying to access memory that is currently on disk, it must first be copied to RAM – **This takes time!!**

Not a good idea for Real-Time applications!

NEUTRINO RTOS

Memory & Paging

(small detour...)

```
#include <sys/mman.h>

int mlock(const void * addr, size_t len);
int mlockall(int flags);
```

calling mlock() and mlockall() lock part or all of the calling process's virtual address space into RAM, preventing that memory from being paged to the swap area.

The addr must be a multiple of PAGESIZE, which depends on the target platform.

The calling process needs superuser capabilities to mlock().

Flags :

MCL_CURRENT lock all pages which are currently mapped into the address space of the process.

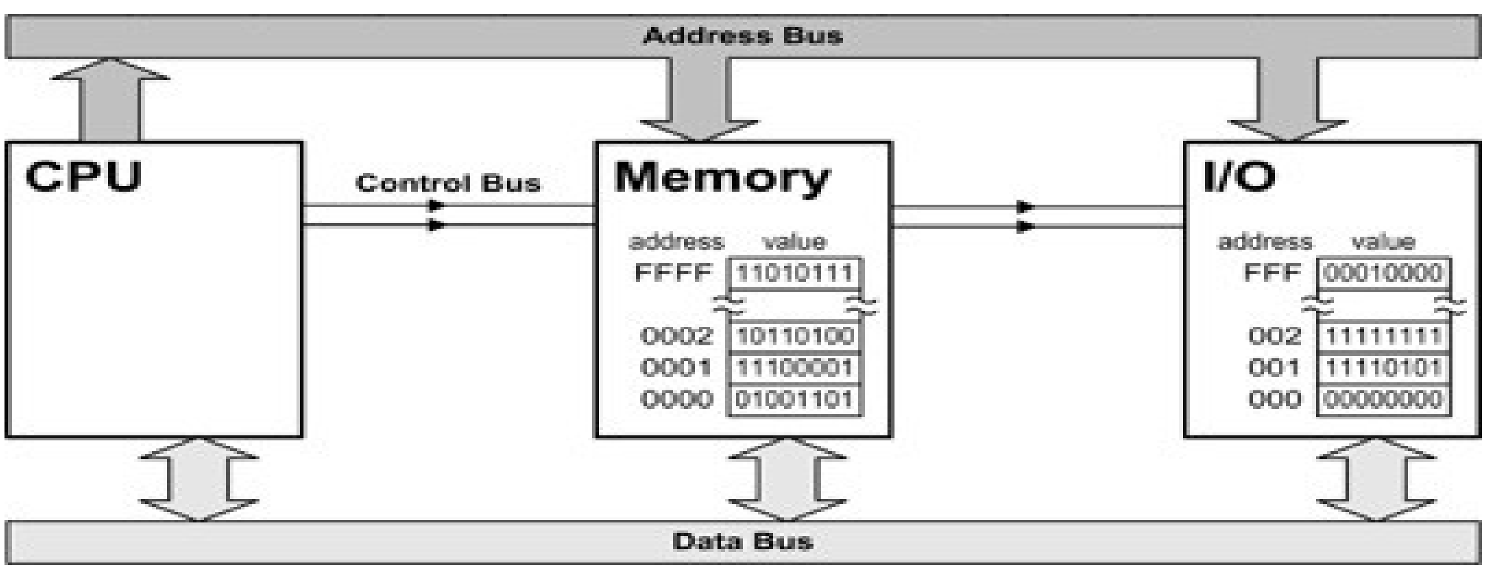
MCL_FUTURE Lock all pages which will become mapped into the address space of the process in the future.

Memory & I/O Space
(another small detour...)

```
#include <stdint.h>
#include <sys/mman.h>

uintptr_t mmap_device_io( size_t len, uint64_t io );
```

QNX Specific!

The mmap_device_io() function maps len bytes of device I/O memory at io and makes it accessible via the in*() and out*() functions in <hw/inout.h>.



Real-Time Applications on QNX

...

POSIX IPC Services

...

Reader/Writer locks

Semaphores

Synchronization via Scheduling Algorithm

Shared Memory

POSIX Signals

QNX NEUTRINO RTOS

POSIX Signals

```
#include <sys/types.h>            #include <signal.h>
  // setup a signal handler
... signal(int   sig, void(*func)(int));
  // send a signal
int raise (int   sig);
int kill  (pid_t pid, int sig);
  // wait for a signal
int pause (void);
int sigpause( int sig );
```

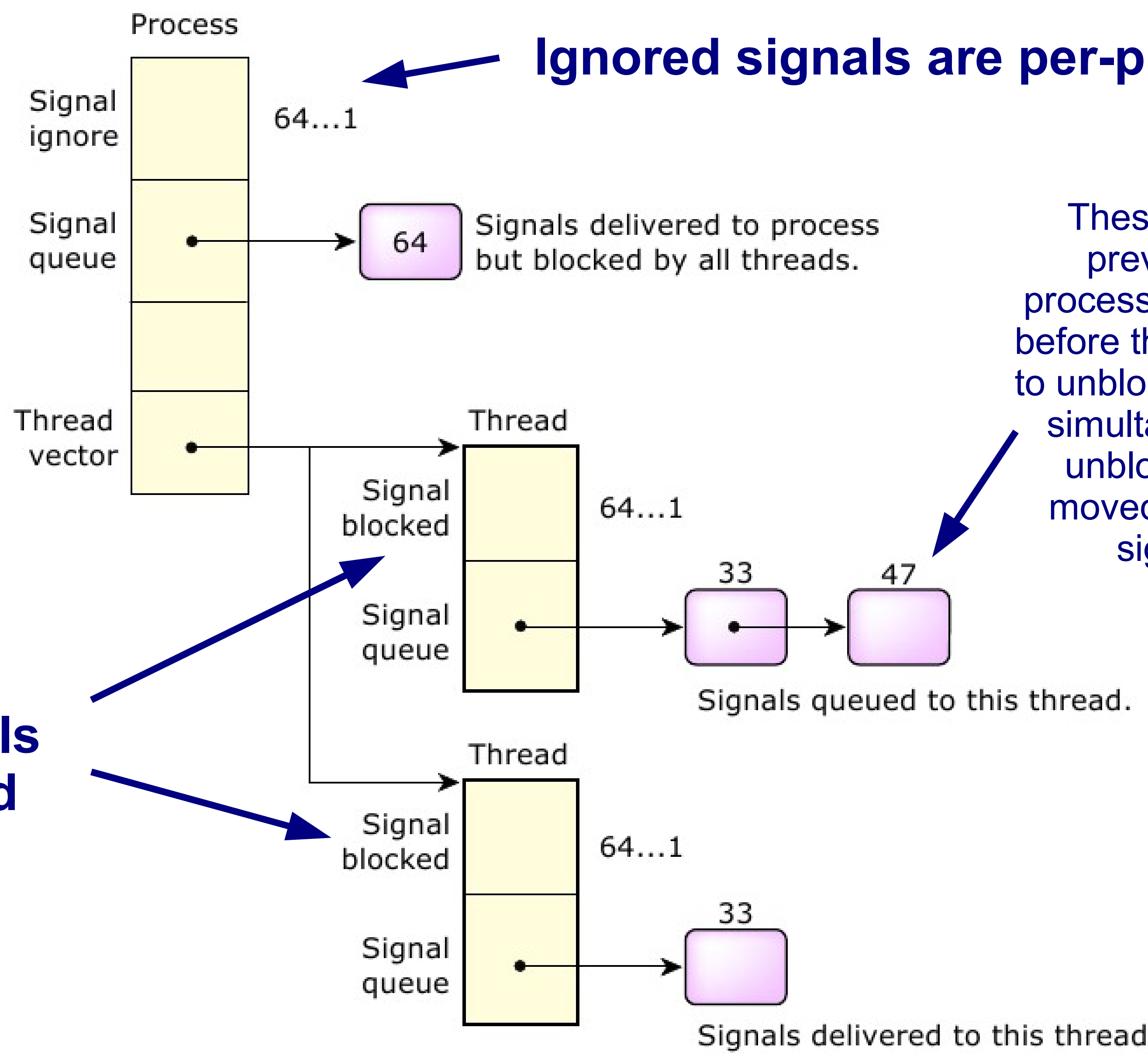
Send **asynchronous** signals between **processes**.

When a signal is received, whatever code the process is running is interrupted, and a signal handling function is executed. After returning from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.

The signal catching function must be: void func(int sig_no) {...}

QNX NEUTRINO RTOS

POSIX Signals



Ignored signals are per-process

These signals were previously on the processes' signal queue, before the thread decided to unblock several signals simultaneously. When unblocked, they are moved to the thread's signal queue.

Blocked signals are per-thread

POSIX Signals

```
#include <sys/types.h>
#include <signal.h>

int sigprocmask( int how,
                 const sigset_t *set, sigset_t *oset)

int pthread_sigmask( int how,
                 const sigset_t* set, sigset_t* oset)
```

Each sent/received signal is identified by the signal type (an int).

Each process may ignore some signals types.
Each thread in a process may block one or more signal types.
ignored signals are simply discarded
blocked signals are queued
(until they are either ignored or unblocked).

POSIX Signals

In multi-threaded applications:

The signal actions are maintained at the process level.
If a thread ignores or catches a signal, it affects all threads within the process.

The signal mask is maintained at the thread level.
If a thread blocks a signal, it affects only that thread.

An un-ignored signal targeted at a thread will be delivered to that thread alone (pthread_kill()).

An un-ignored signal targeted at a process is delivered to the first thread that doesn't have the signal blocked.
If all threads have the signal blocked, the signal will be queued on the process until any thread ignores or unblocks the signal. If ignored, the signal on the process will be removed. If unblocked, the signal will be moved from the process to the thread that unblocked it.

POSIX Signals

```
#include <signal.h>

int pthread_kill( pthread_t thread, int sig );
```

It is possible to send a signal to a specific thread, but only from another thread in the same process.

This is very seldom used... as there are many other inter-thread synchronisation mechanisms available.

NEUTRINO RTOS

POSIX Signals

```
   // setup a signal handler
... signal(int sig, void(*func)(int));
int sigaction(int sig, const struct sigaction *act,
                             struct sigaction *oact);
   // send a signal
int raise (int   sig);
int kill  (pid_t pid, int sig);
int sigqueue(pid_t pid, int signo,union sigval value)

   // wait for a signal
int pause    (void);
int sigpause( int sig );
int sigsuspend  (sigset_t *sigmask );
int sigwait     (sigset_t *set, int *sig );
int sigwaitinfo (sigset_t *set, siginfo_t *info );int
sigtimedwait    (sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout );
```

A 32 bit value may be sent together with the signal

QNX NEUTRINO RTOS

POSIX Signals

Signal range	Description
1 ... 57	57 POSIX signals (including traditional UNIX signals)
41 ... 56	16 POSIX realtime signals (SIGRTMIN to SIGRTMAX)
57 ... 64	Eight special-purpose QNX Neutrino signals

The eight special signals cannot be ignored or caught.

Calling signal(), sigaction() or SignalAction() to change them will fail with an error of EINVAL.

These signals are always blocked and have signal queuing enabled.

An attempt to unblock these signals via the sigprocmask() function or SignalProcmask() kernel call will be quietly ignored.

Designed to be received synchronously (sigwait(), ...)

QNX NEUTRINO RTOS

POSIX Signals

Signal	Description
SIGABRT	Abnormal termination signal such as issued by the abort() function.
SIGALRM	Time-out signal such as issued by the alarm()function.
SIGCHLD	Child process terminated. The default action is to ignore the signal.
SIGEMT	EMT instruction (emulator trap).
SIGFPE	Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow.
SIGHUP	Death of session leader, or hangup detected on controlling terminal.
SIGILL	Detection of an invalid hardware instruction.
SIGINT	Interactive attention signal (Break).
SIGKILL	Termination signal — should be used only for emergency situations. **This signal cannot be caught or ignored.**
SIGPIPE	Attempt to write on a pipe with no readers.
SIGPOLL	Pollable event occurred.
SIGQUIT	Interactive termination signal.

QNX NEUTRINO RTOS

POSIX Signals

Signal	Description
SIGSEGV	Detection of an invalid memory reference. If a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGSTOP	Stop process (the default). **This signal cannot be caught or ignored.**
SIGSYS	Bad argument to system call.
SIGTERM	Termination signal.
SIGTRAP	Unsupported software interrupt.
SIGTSTP	Stop signal generated from keyboard.
SIGTTIN	Background read attempted from control terminal.
SIGTTOU	Background write attempted to control terminal.
SIGURG	Urgent condition present on socket.
SIGUSR1	Reserved as application-defined signal 1.
SIGUSR2	Reserved as application-defined signal 2.

POSIX Signals – Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void ) {
  /* set an alarm to go off in 5 seconds */
  alarm( 5 );
  /*
   * Wait until we receive a SIGALRM signal. However,
   * since we don't have a signal handler, any signal
   * will kill us.
   */
  printf( "Waiting to die in 5 seconds...\n" );
  pause();
  return EXIT_SUCCESS;
}
```

NEUTRINO RTOS

POSIX Signals – Example 2

```
volatile sig_atomic_t signal_count, signal_number;
void alarm_handler( int signum ) {
  ++signal_count;
  signal_number = signum;
}

int main( void ) {
  unsigned long i;    signal_count = 0;   signal_number = 0;
  signal( SIGINT, alarm_handler );
  for( i = 0; i < 100000; ++i ) {
    if( i == 19999 ) raise( SIGINT );
    if( signal_count > 0 ) break;
  }
  if( i == 100000 ) {
    printf( "\nNo signal was raised.\n" );
  } else if( i == 19999 ) {
    printf( "\nSignal %d was raise()'d.\n", signal_number );
  } else {
    printf( "\nUser raised signal #%d.\n",  signal_number );
  }
  return EXIT_SUCCESS;
}
```

Real-Time Applications on QNX

- ...
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Services
  - Synchronization via atomic operations
  - Message Passing
  - Pulses
  - Events

NEUTRINO RTOS

Atomic Operations

Some operations are atomic (not interruptible) since they are implemented by a single CPU instruction!

QNX Neutrino gives us portable direct access to these functions (Not POSIX compliant!)

```
#include <atomic.h>
```

Supported Functions:

- adding a value
- subtracting a value
- clearing bits
- setting bits
- toggling (complementing) bits.

Real-Time Applications on QNX

...

Thread CPU Scheduling

POSIX IPC Services

QNX Neutrino IPC Services

- Synchronization via atomic operations
- Message Passing
- Pulses
- Events

NEUTRINO RTOS

Message Passing

```
#include <sys/neutrino.h>

int MsgSend();      // Send a message
int MsgReceive();   // Receive a message
int MsgReply();     // Reply to a message
int MsgError();     // Reply with an error status
```

Message Passing: a QNX specific IPC mechanism

Simple mechanism, optimised for QNX, so very efficient!

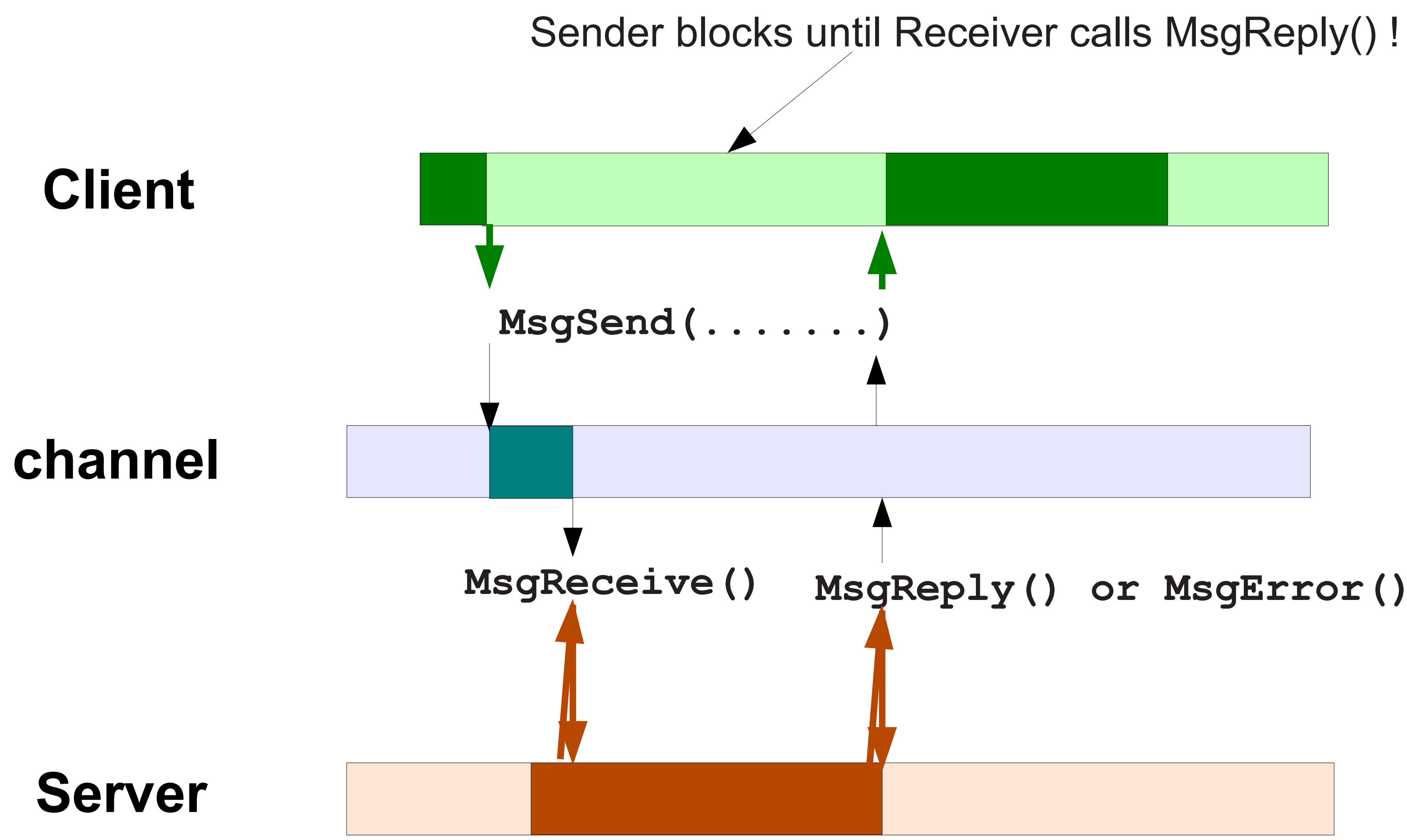
Many other POSIX services (e.g. named semaphores) are built over the Message Passing IPC service.

Synchronous → allows synchronisation between threads

Messages → allows data passing between threads

Messages are sent over a 'channel', which must be created first!

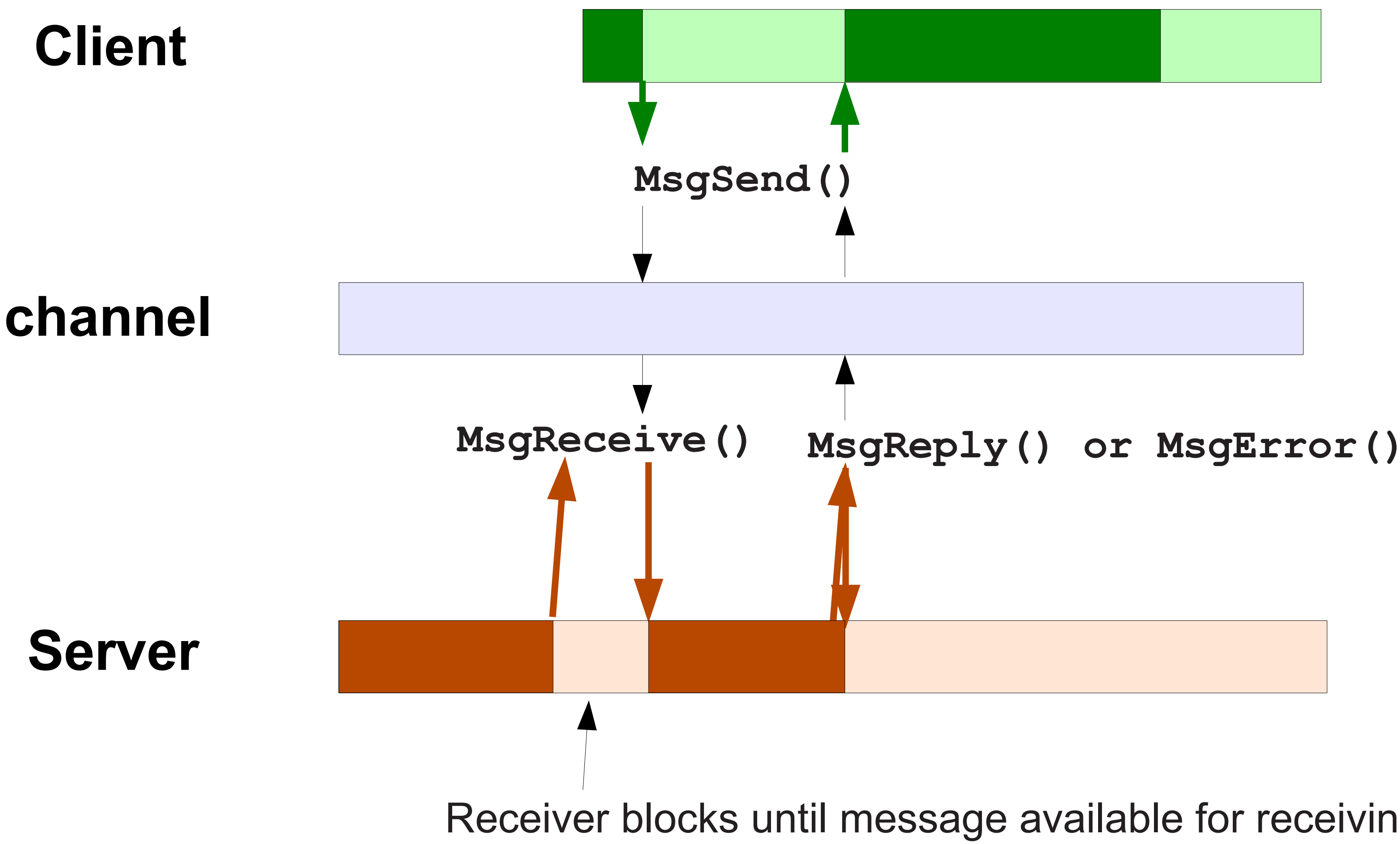
Message Passing



NOTE
Exact execution sequence will depend on process priorities. Here we assume the same priority for both processes.

NOTE
Multiple messages may be queued in priority order if no thread is willing to receive them.

Message Passing



NOTE
Exact execution sequence will depend on process priorities. Here we assume the same priority for both processes.

Message Passing

```
#include <sys/neutrino.h>

ChannelCreate()   // Create a channel to receive messages on.
ChannelDestroy()  // Destroy a channel.
ConnectAttach()   // Create a connection to send messages on.
ConnectDetach()   // Detach a connection.
```

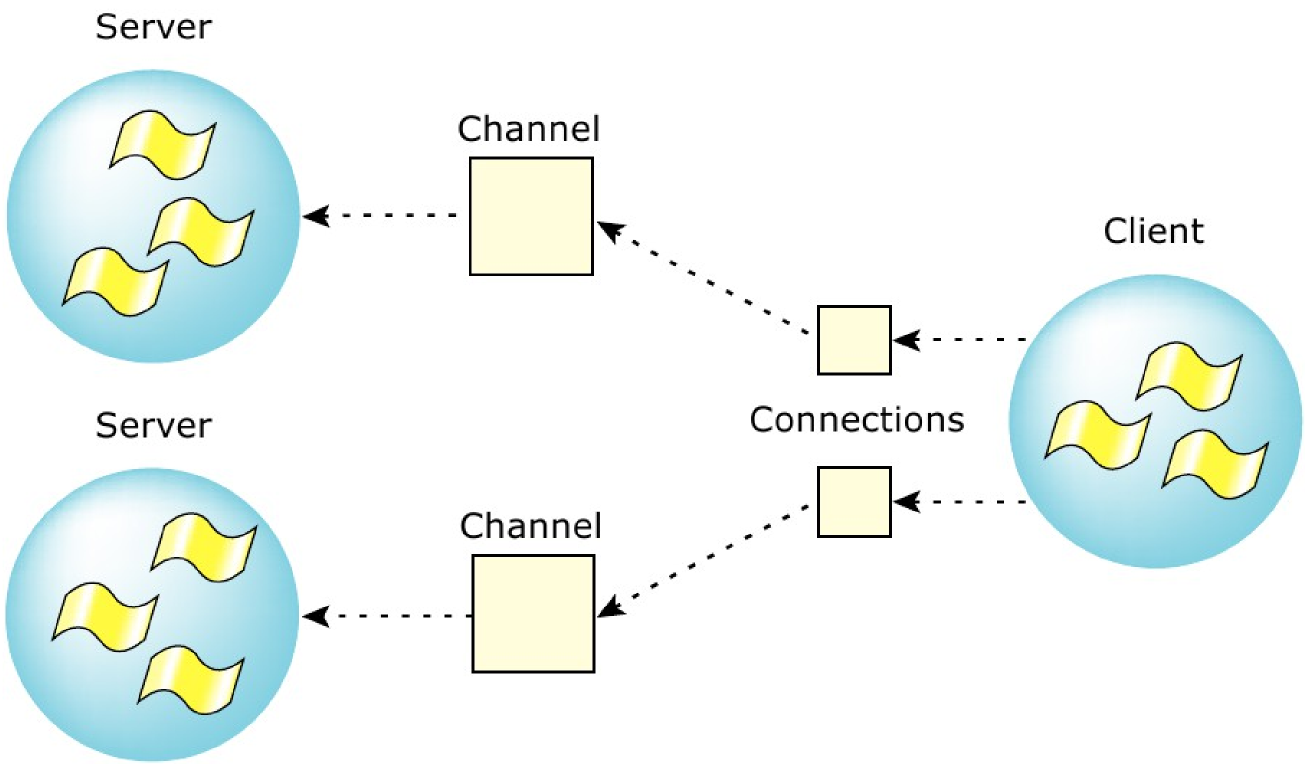
Messages are sent over a 'channel'.

A thread that wishes to receive messages creates a channel;

A thread that wishes to send a message to that thread must first make a connection by “attaching” to that channel.
(client connections map directly into FD – file descriptors)

Message Passing

```
#include <sys/neutrino.h>

ChannelCreate()   // Create a channel to receive messages on.
ChannelDestroy()  // Destroy a channel.
ConnectAttach()   // Create a connection to send messages on.
ConnectDetach()   // Detach a connection.
```



NEUTRINO RTOS

Message Passing

```
// Server code...

chid = ChannelCreate(flags);
for(;;) {
   rcv_id = MsgReceive(chid, &msg, msg_size, &info);
   /* Perform message processing here */
   MsgReply( rcv_id, status, &msg, msg_size );
  }
```

```
// Client code...

coid = ConnectAttach(ND_LOCAL_NODE, 0, chid,
                     _NTO_SIDE_CHANNEL, 0);

MsgSend(coid, &smsg, smsg_size, &rmsg, rmsg_size);
...
```

QNX NEUTRINO RTOS

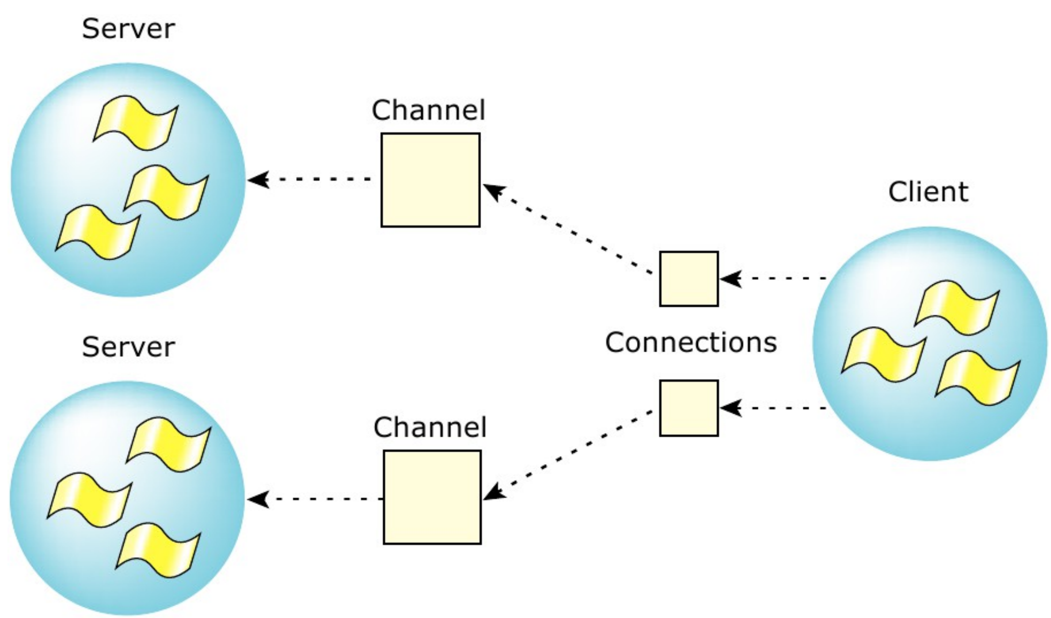
Message Passing

The channel has several lists of messages associated with it:

Receive: A LIFO queue of threads waiting for messages.

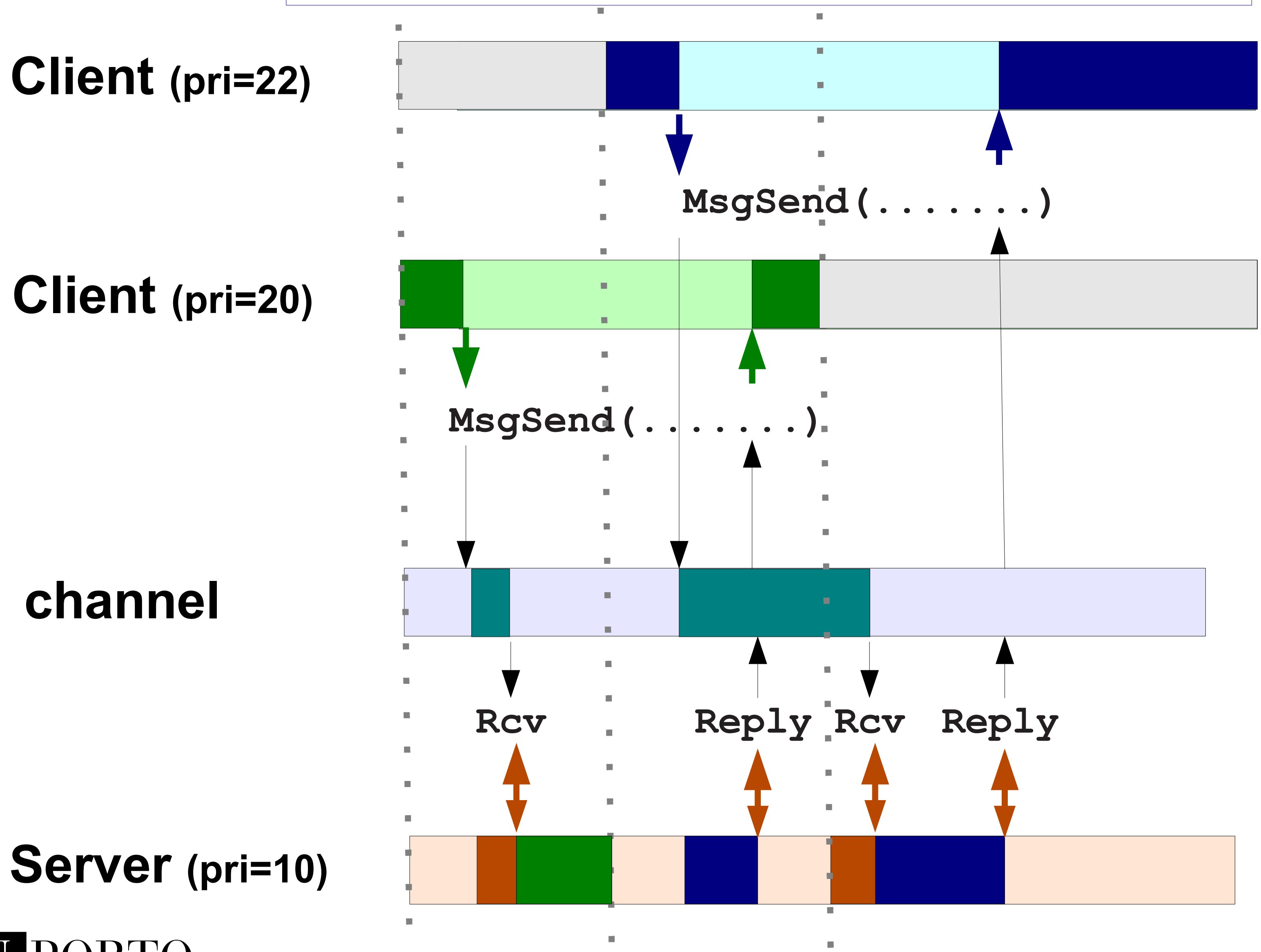
Send: A priority FIFO queue of threads that have sent messages that haven't yet been received.

Reply: An unordered list of threads that have sent messages that have been received, but not yet replied to.



Message Passing

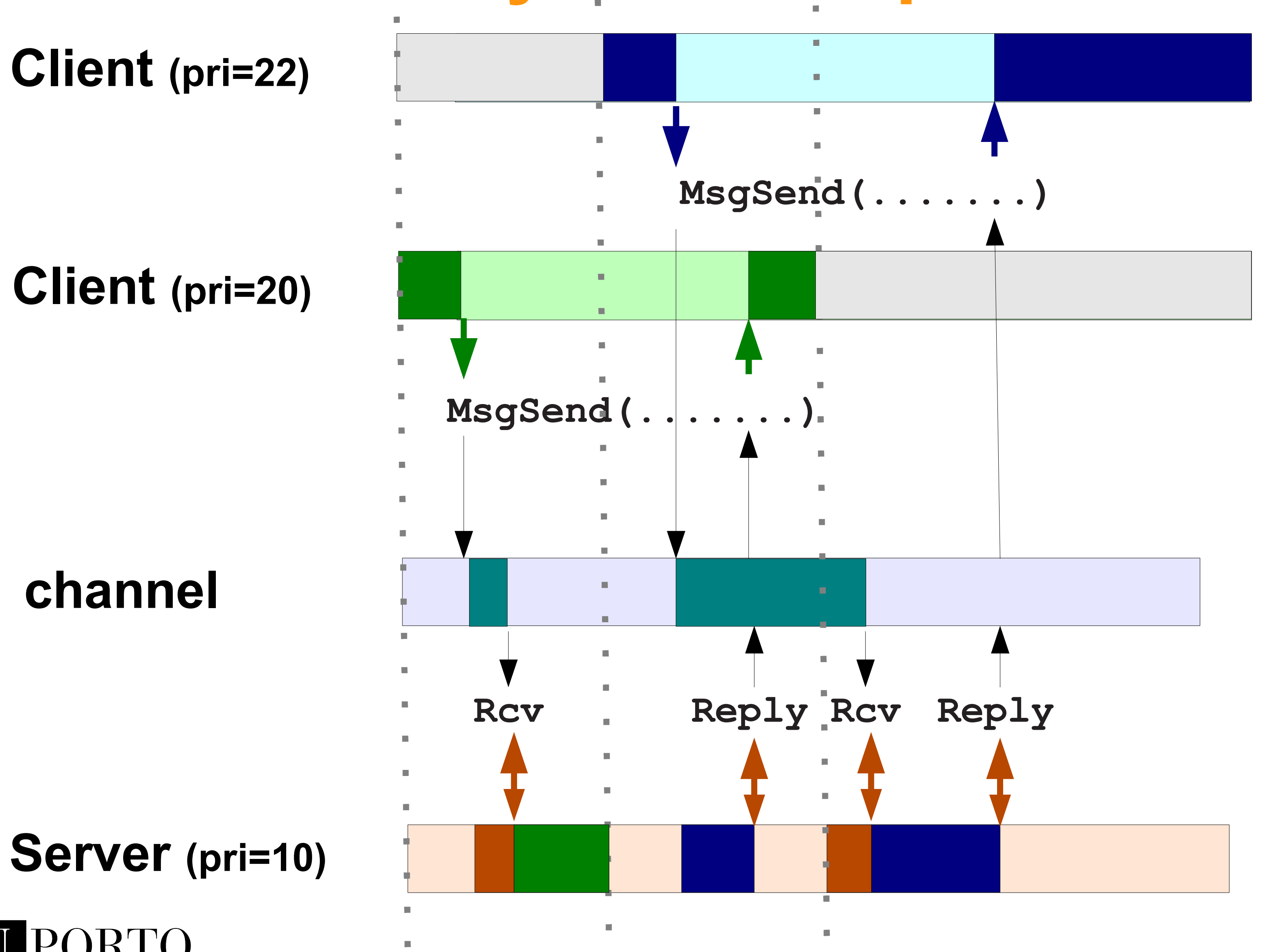
Channels implement Priority Inheritance mechanism



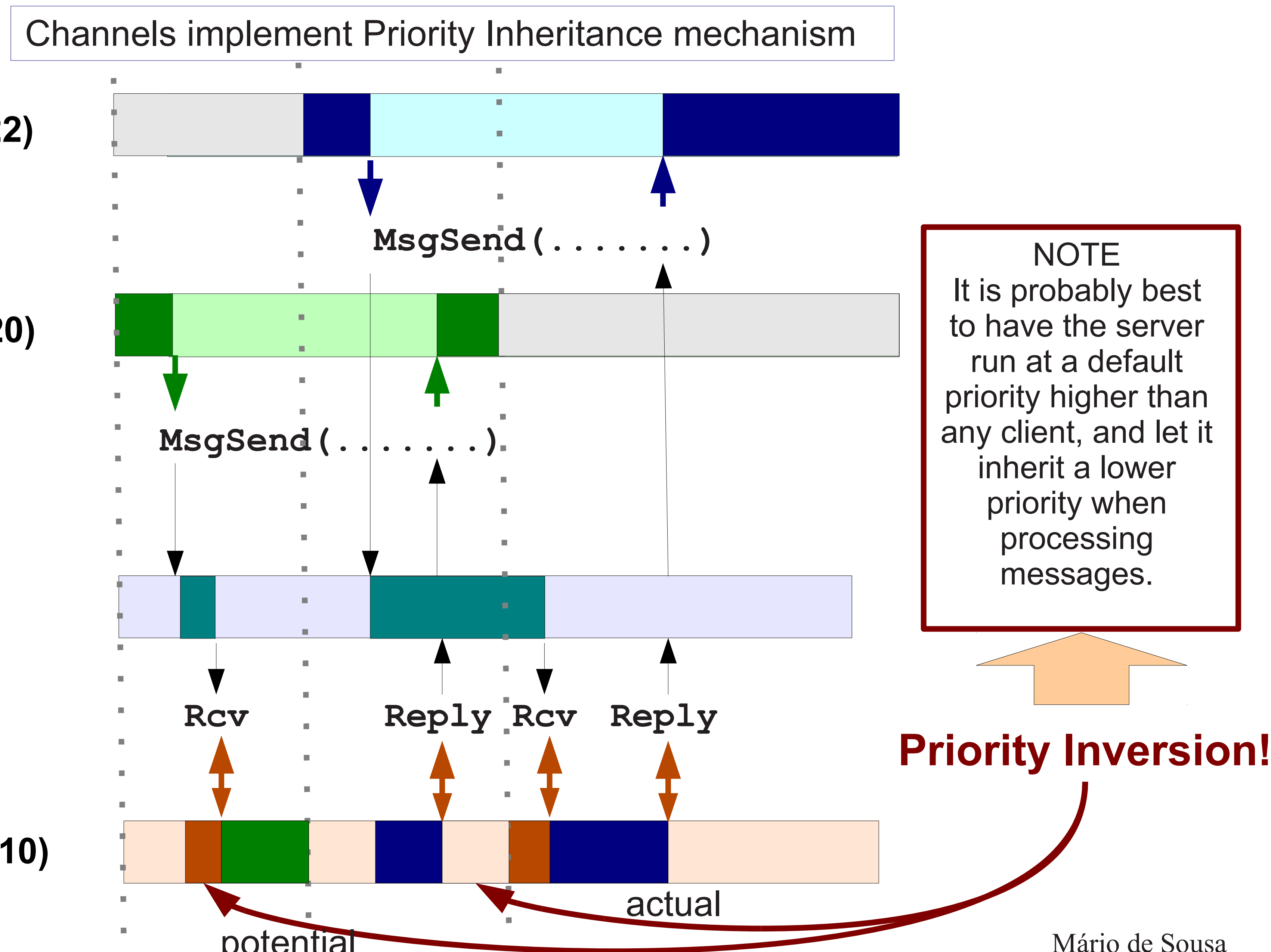
NOTE
It is possible to turn off priority inheritance by specifying the _NTO_CHF_FIXED_PRIORITY flag when calling ChannelCreate()

Message Passing

Can you find a problem here?

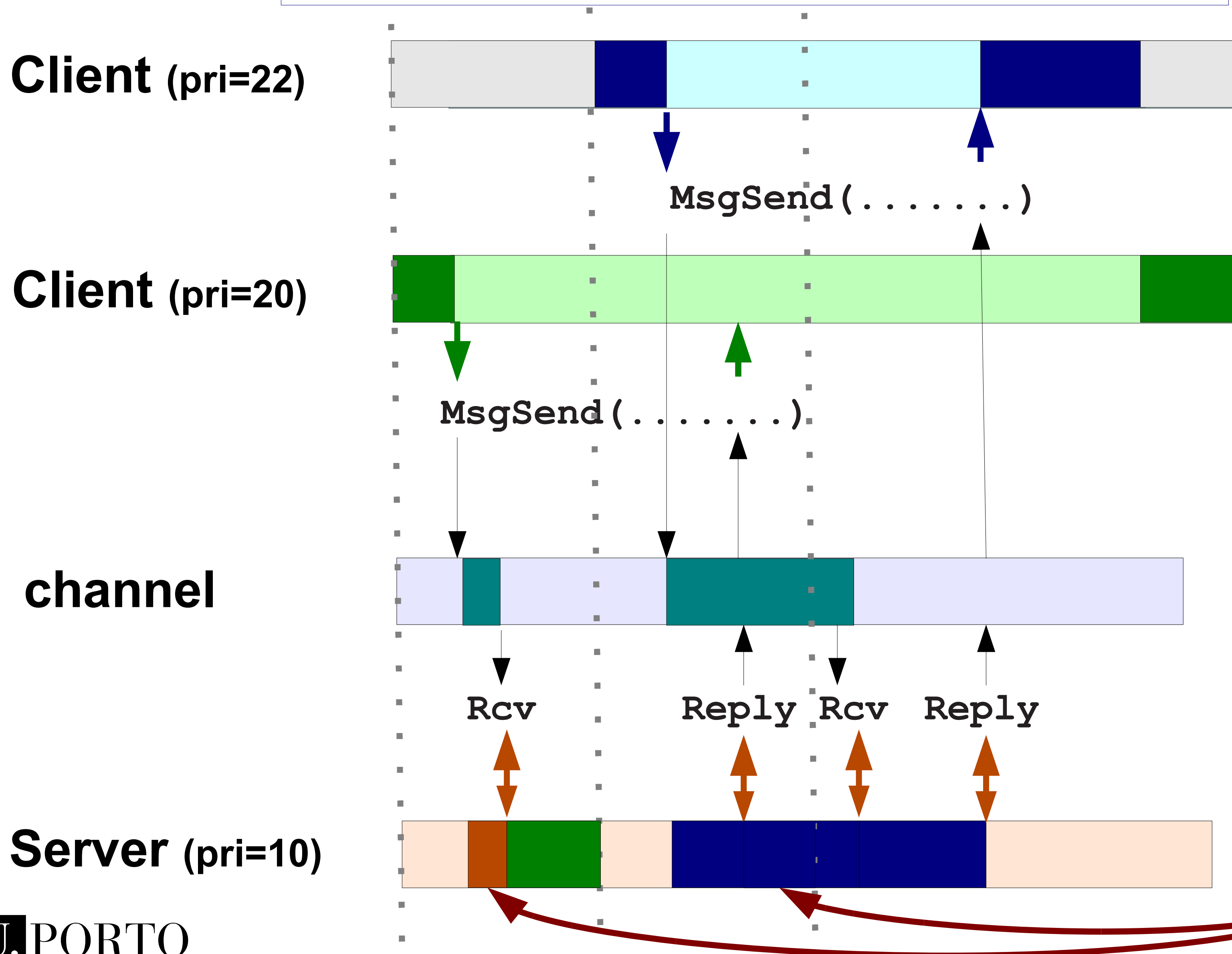


Message Passing



Message Passing

Channels implement Priority Inheritance mechanism



To solve this issue, QNX Neutrino also boosts the priority as long as there is at least 1 message in the channel waiting to be processed.

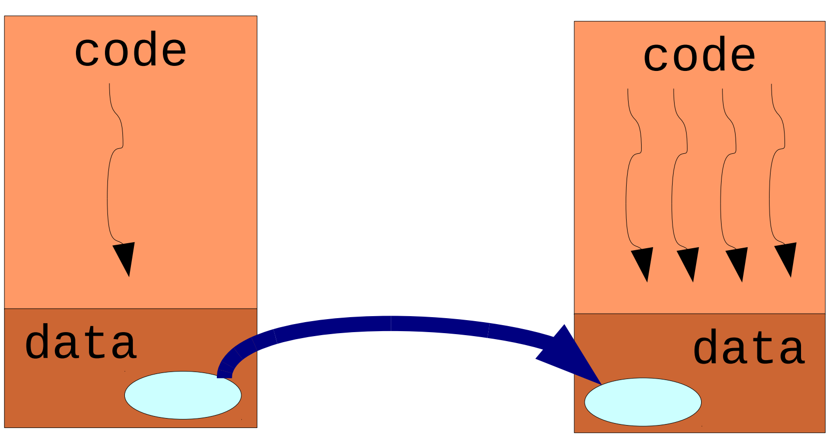
But boost the priority of which thread?

Of all threads that have have previously received messages from that channel!!! (this may not be what we want!)

Message Passing

Messages are copied directly from the address space of one thread to another without intermediate buffering →
the message-delivery performance approaches the memory bandwidth of the underlying hardware.

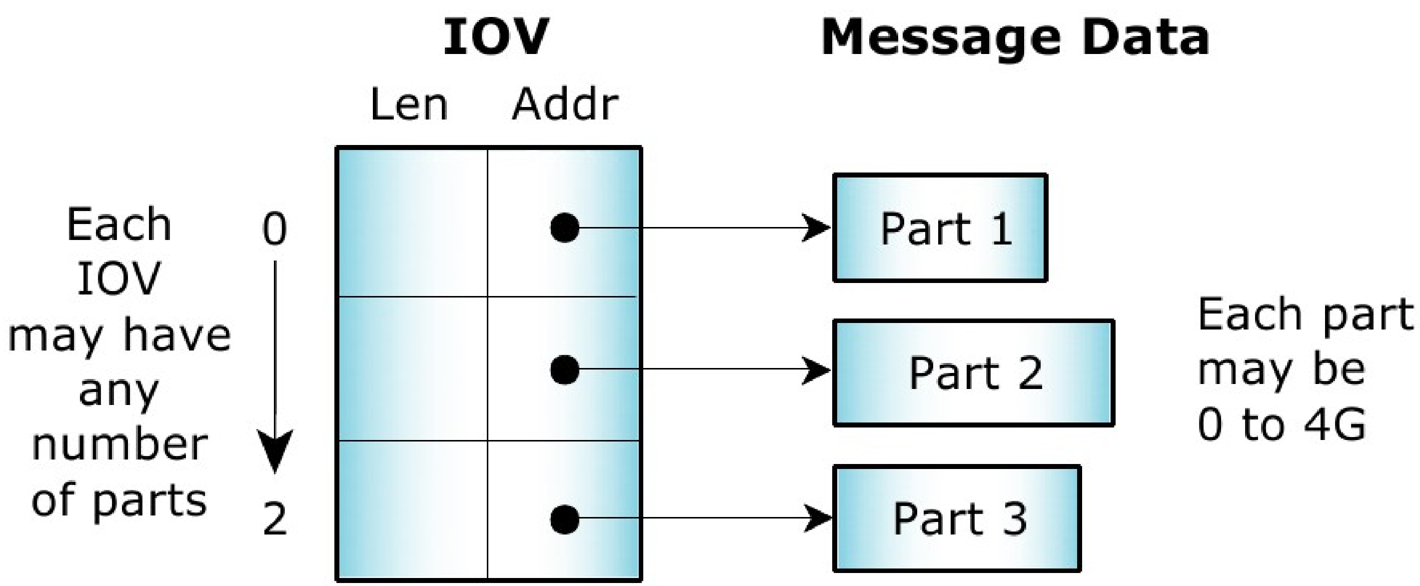
However, the kernel detects large message transfers and uses “page flipping” for those cases. (Since most messages passed are quite tiny, copying messages is often faster than manipulating MMU page tables)



For bulk data transfer, shared memory between processes (with message-passing or another synchronization primitive for notification) is also a viable option.

Message Passing

The messaging primitives support multipart transfers, so that a message delivered from the address space of one thread to another needn't pre-exist in a single, contiguous buffer.

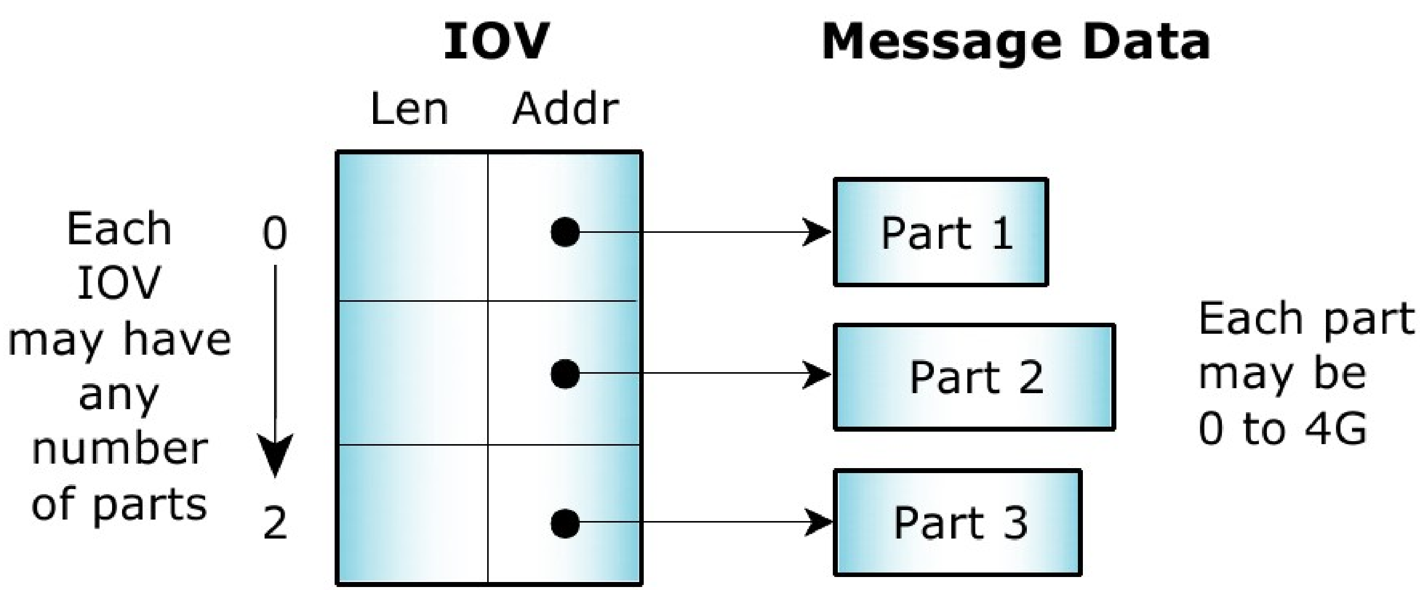


NEUTRINO RTOS

Message Passing

Function	Send message	Reply message
MsgSend()	Simple	Simple
MsgSendsv()	Simple	IOV
MsgSendvs()	IOV	Simple
MsgSendv()	IOV	IOV

IOV	Simple direct
MsgReceivev()	*MsgReceive()*
MsgReceivePulsev()	*MsgReceivePulse()*
MsgReplyv()	*MsgReply()*
MsgReadv()	*MsgRead()*
MsgWritev()	*MsgWrite()*



NEUTRINO RTOS

Message Passing

```
#include <sys/neutrino.h>

int ChannelCreate( unsigned flags );

int ConnectAttach( uint32_t nd, pid_t pid, int chid,
                   unsigned index, int flags );

int MsgSend( int coid, const void* smsg, int sbytes,
                               void* rmsg, int rbytes);

int MsgReceive( int chid, void * msg, int bytes,
                struct _msg_info * info );

int MsgReply( int rcvid, int status,
              const void* msg, int size );
int MsgError( int rcvid, int error );
```

NEUTRINO RTOS

Message Passing

```
#include <sys/neutrino.h>

int MsgRead( int rcvid, void* msg, int bytes,
             int offset );
int MsgWrite(int rcvid, const void* msg, int size,
             int offset );
int MsgInfo( int rcvid, struct _msg_info* info );
```

MsgRead() →

Read data from an already MsgReceive()'d message.

Any thread in the receiving process may read this data.

MsgWrite() →

Write data to the reply buffer of a thread identified by rcvid.

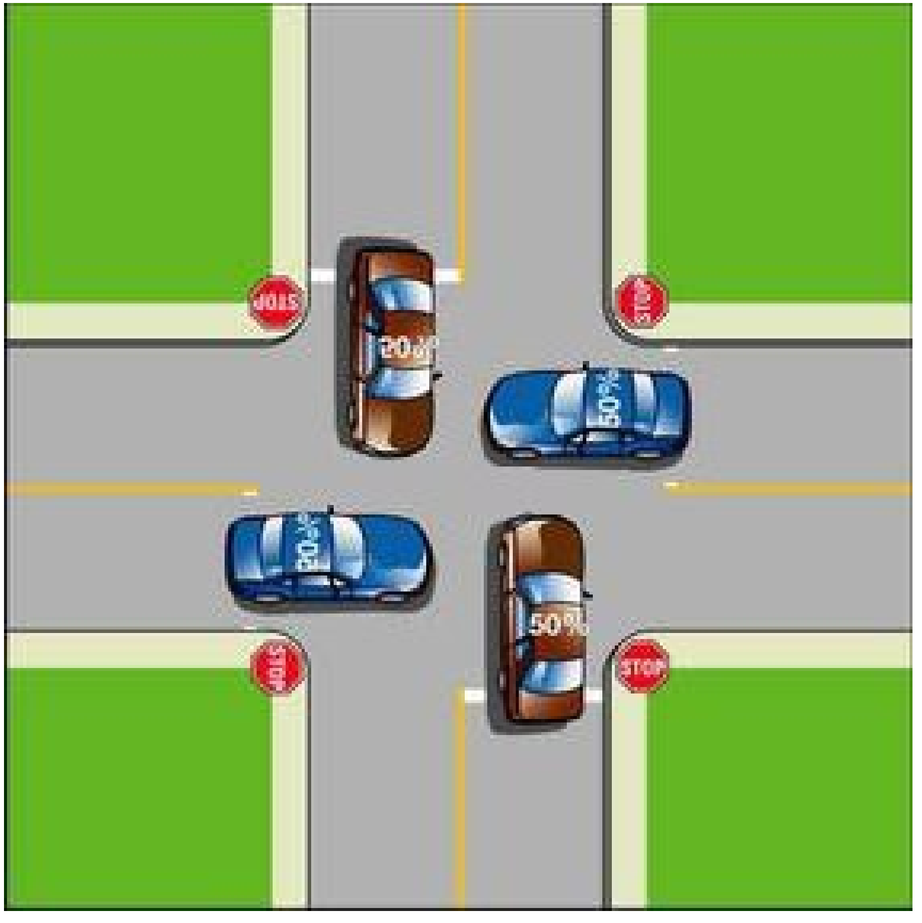
Any thread in the receiving process may write to the reply msg.

MsgInfo() →

Get additional information about a received message.

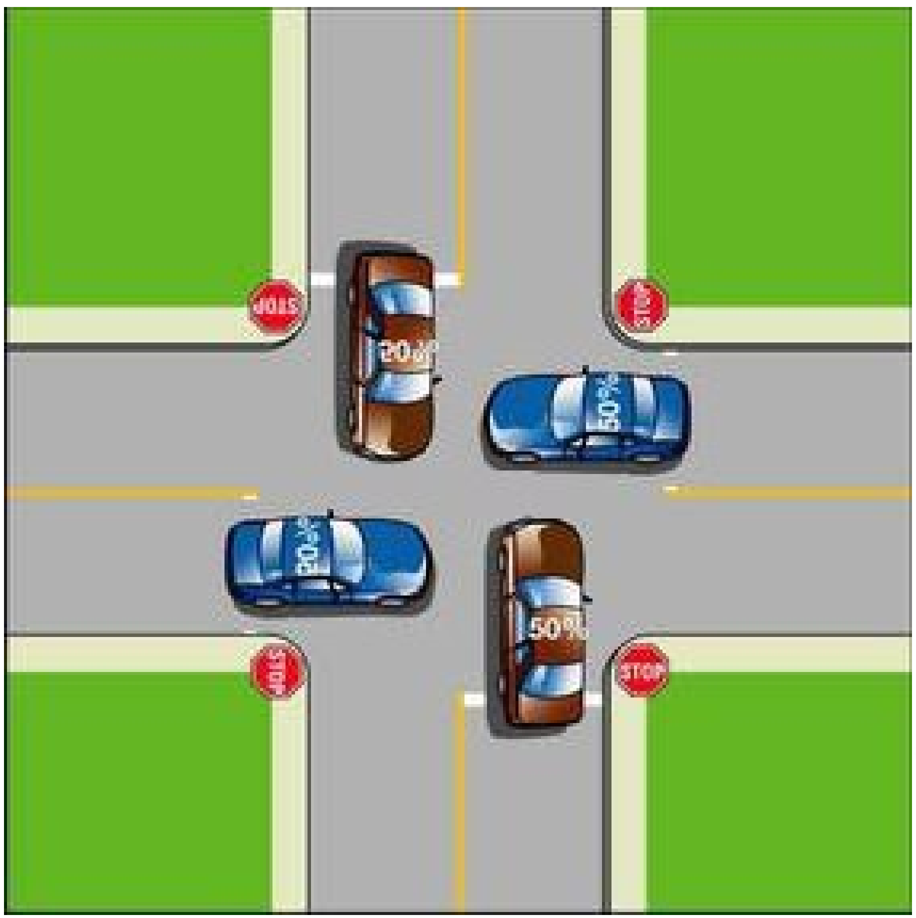
Message Passing

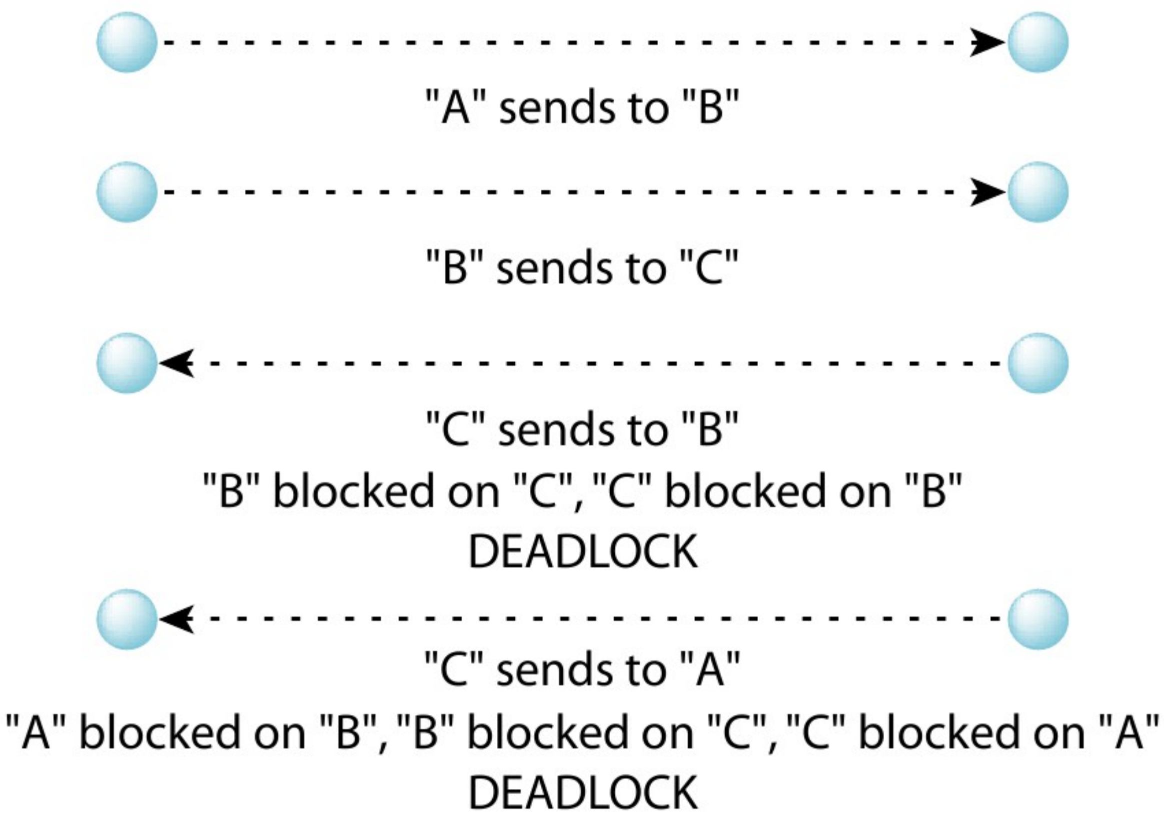
Deadlock may occur when we have more than one resource being shared by two or more entities...



Message Passing

Deadlock may occur when we have more than one resource being shared by two or more entities...



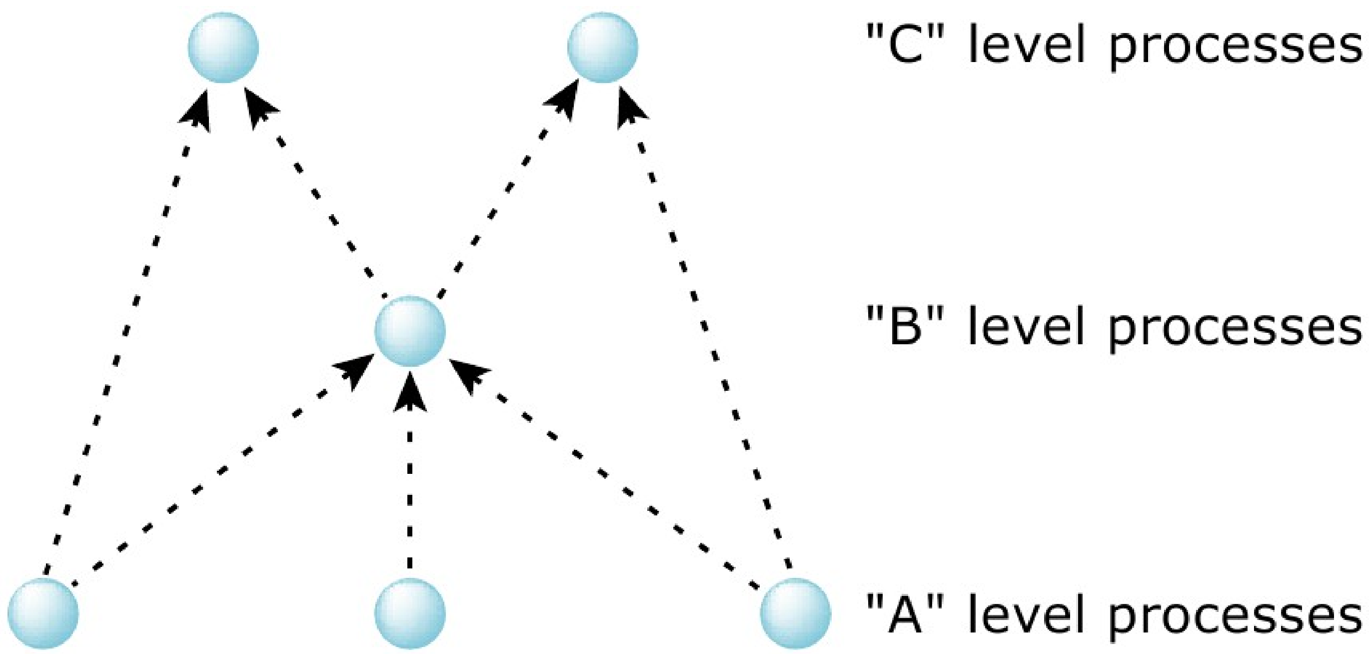


Message Passing

Using message passing, it is possible to guarantee we never reach deadlock as long as we:

Never have two threads sending (MsgSend()) to each other;

Threads are organized in an hierarchy, with sends going up the tree.



NEUTRINO RTOS

Real-Time Applications on QNX

...

Thread CPU Scheduling

POSIX IPC Services

QNX Neutrino IPC Services

- Synchronization via atomic operations
- Message Passing
- Pulses
- Events

Pulses

```
#include <sys/neutrino.h>

int MsgSendPulse( int coid, int priority, int code,
                  int value );

int MsgReceivePulse(int chid, void * pulse,
                int bytes, struct _msg_info * info );
```

Fixed-size, non-blocking (**asynchronous**) messages
(four bytes of data plus a single byte code).

Often used as a notification mechanism within interrupt handlers

Pulses, like Messages, are sent across channels...

When a pulse is read via the MsgReceive() call, the rcvid returned is zero
indicating a MsgReply() mustn't be sent.

When no pulse is available, MsgReceivePulse() will block!

Real-Time Applications on QNX

- ...
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Services
  - Synchronization via atomic operations
  - Message Passing
  - Pulses
  - Events

QNX
NEUTRINO RTOS

Events

There are several different (asynchronous) event types:

QNX Neutrino pulses,

interrupts,

various forms of signals (POSIX, UNIX),

and Forced “unblock” events.

“Unblock” is a means by which a thread can be released from a deliberately blocked state without any explicit event actually being delivered.

The events encountered by an executing thread can come from any of three sources:

a MsgDeliverEvent() kernel call invoked by a thread

A hardware interrupt

the expiry of a timer

A cllient being notified of the event may want to choose the event type it wants to receive...

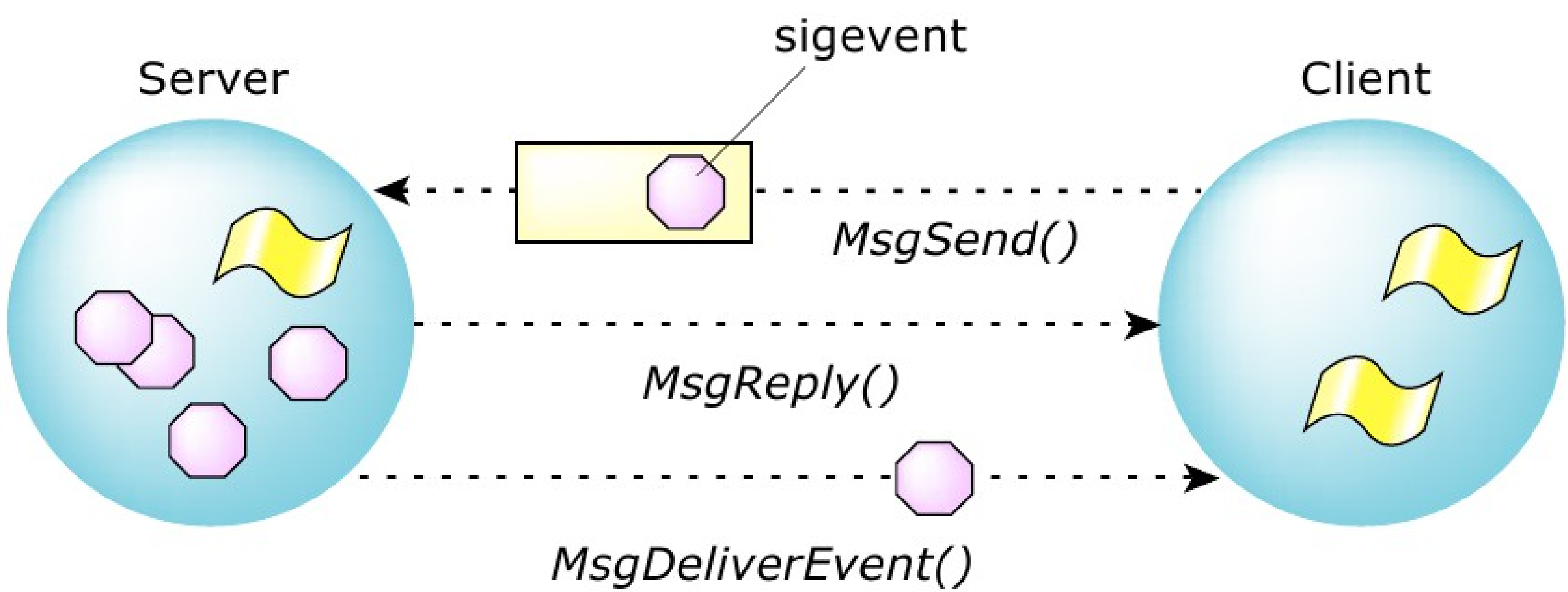
So a delivery mechanism capable of sending any type of event was created... MsgDeliverEvent() !

Events

```
#include <sys/neutrino.h>

int MsgDeliverEvent( int rcvid,
                     const struct sigevent* event );
```

Deliver any type of event, over a channel...

struct sigevent → defines the event type to send!
(typically received from the client itself)



NEUTRINO RTOS

Events - Example

```
// my_hdr.h

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

struct my_msg {
  short type;
  struct sigevent event;
};

#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL+5
#define MSG_GIVE_PULSE _IO_MAX+4
#define MY_SERV "my_server_name"
```

Events - Example

```
// client.c
#include "my_hdr.h"

int main( int argc, char **argv) {
  int chid, coid, srv_coid, rcvid;
  struct my_msg msg;     struct _pulse pulse;

  chid = ChannelCreate(0);
  coid = ConnectAttach(0, 0, chid, _NTO_SIDE_CHANNEL, 0);
  /* fill in the event structure for a pulse */
  SIGEV_PULSE_INIT(&msg.event, coid,
                   SIGEV_PULSE_PRIO_INHERIT, MY_PULSE_CODE, 0);
  msg.type = MSG_GIVE_PULSE;
  /* find the server */
  if ((srv_coid = name_open(MY_SERV, 0)) == -1) {exit(1);}
  /* give the pulse event for later delivery */
  MsgSend( srv_coid, &msg, sizeof(msg), NULL, 0);
  /* wait for the pulse from the server */
  Rcvid = MsgReceivePulse(chid, &pulse, sizeof(pulse), NULL);
  printf("got pulse %d,expecting %d\n",pulse.code,MY_PULSE_CODE);
  return 0;
}
```

NEUTRINO RTOS

Events - Example

```
// server.c
#include "my_hdr.h"
int main( int argc, char **argv) {
  int rcvid;   struct my_msg msg;
  name_attach_t *attach;

  /* attach the name the client will use to find us */
  if ((attach = name_attach(NULL, MY_SERV, 0))== NULL) {exit(1);}
  /* wait for the message from the client */
  rcvid = MsgReceive( attach->chid, &msg, sizeof( msg ), NULL );
  MsgReply(rcvid, 0, NULL, 0);
  if ( msg.type == MSG_GIVE_PULSE ) {
    /* wait until it is time to notify the client */
    sleep(2);
    /* deliver notification to client that client requested */
    MsgDeliverEvent( rcvid, &msg.event );
    printf("server:delivered event\n");
  } else {
    printf("server: unexpected message \n");
  }
  return 0;
}
```

NEUTRINO RTOS

Real-Time Applications on QNX

- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Service
- Clocks and Timers

NEUTRINO RTOS

Clocks

```
#include <sys/types.h>
#include <time.h>

int clock_gettime  (clockid_t clock_id,
                    struct timespec * tp );
int clock_settime  (clockid_t id,
                    const struct timespec * tp );
```

The operating system maintains several distinct clocks, from which we may read the time:

- CLOCK_REALTIME → Real-time clock that maintains system time
- CLOCK_MONOTONIC → A clock that always increases at a constant rate and can't be adjusted.
- CLOCK_SOFTTIME → Same as CLOCK_REALTIME, but if the CPU is in powerdown mode, the clock stops running.

NEUTRINO RTOS

Clocks

```
/* This program sets the clock forward 1 day. */
#include<stdio.h>                  #include<stdlib.h>
#include<unistd.h>                 #include<time.h>

int main( void ) {
  struct timespec stime;
  if(clock_gettime(CLOCK_REALTIME, &stime) == -1) {
    perror( "getclock" );
    return EXIT_FAILURE;
  }
  stime.tv_sec += (60*60)*24L; /* Add one day */
  stime.tv_nsec = 0;
  if(clock_settime(CLOCK_REALTIME, &stime) == -1 ) {
    perror( "setclock" );
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

NEUTRINO RTOS

Clocks

```
#include <time.h>
int clock_getres(clockid_t clock_id,
                 struct timespec * res );

#include <sys/neutrino.h>
int ClockPeriod( clockid_t id,
                 const struct _clockperiod * new,
                       struct _clockperiod * old,
                 int reserved );
```

It is possible to get the resolution of a specific clock.

How clocks are maintained is OS specific. QNX Neutrino uses a standard periodic interrupt timer for each clock tick.

QNX Neutrino also allows us to change the clock tick time dynamically with ClockPeriod() → Not POSIX compliant.

QNX NEUTRINO RTOS

Clocks

```
#include <sys/neutrino.h>
#include <inttypes.h>
uint64_t ClockCycles( void );

// The current CPU frequency!
#include <sys/syspage.h>
SYSPAGE_ENTRY(qtime)->cycles_per_sec
```

Most CPU architectures have a clock cycle counter, that may be used as a very high precision clock.

On x86, this can be read using the rdtsc assembly instruction.

QNX Neutrino provides an architecture independent method of reading this counter, with ClockCycles().

Since the rate at which this clock increases is dependent on CPU frequency, we may need to get the current CPU frequency.

QNX NEUTRINO RTOS

Clocks

```
uint64_t ClockCycles( void );
```

!! WARNING !!

Depending on CPU architecture, this function returns a value from a register that’s unique to each CPU in an SMP system.
(including multicore CPUs!!)

These registers aren’t synchronized between the CPUs. So if you call ClockCycles(), and then the thread migrates to another CPU and you call ClockCycles() again, you can’t subtract the two values to get a meaningful time duration.

You may want to call ThreadCtl(_NTO_TCTL_RUNMASK, …) to lock a thread onto a single CPU.

Be careful with OSs that scale the CPU frequency dynamically!

Clocks

```
#include <sys/types.h>
#include <time.h>
int clock_getcpuclockid(   pid_t pid,
                           clockid_t *clock_id)
#include <pthread.h>
int pthread_getcpuclockid( pthread_t id,
                           clockid_t* clock_id);

clock_t clock( void );
```

Each process and thread has its own private CPU time clock.

To read these clocks with clock_gettime(), you must first get their id with xxx_getcpuclockid().

You can also read the process' CPU time clock more simply with clock().

Clocks

This is all well a good, but how do we specify a fixed timed delay?

```
#include <sys/types.h>          #include <time.h>
int nanosleep(         const struct timespec* rqtp,
                             struct timespec* rmtp);
int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *rqtp,
                          struct timespec *rmtp);
int nanospin(          const struct timespec *when);
```

xxx_nanosleep() functions block the calling thread for
- a fixed time interval (flags with TIMER_ABSTIME not set),
- until a specific absolute time (flags with TIMER_ABSTIME set)

The function may return before the time elapses, if the calling thread needs to catch a signal.

The nanosleep() function uses CLOCK_REALTIME.
The nanospin() function implements busy waiting.

Timers

NEUTRINO RTOS

How about a fixed timed delay, with asynchronous notification?

```
#include <signal.h>
#include <time.h>
int timer_create( clockid_t clock_id,
                  struct sigevent *evp,
                  timer_t * timerid );

int timer_settime( timer_t timerid, int flags,
                   struct itimerspec *value,
                   struct itimerspec *ovalue );
```

timer_create() → Create a timer for asynchronous notification. The notification may be sent by a POSIX signal, a QNX pulse...

timer_settime() → Set a timer for later expiration.

flags: TIMER_ABSTIME specifies an absolute time, otherwise a relative offset time is considered.

Timers

How about a fixed timed delay, with asynchronous notification?

```
#include <signal.h>
#include <time.h>
int timer_create( clockid_t clock_id,
                  struct sigevent *evp,
                  timer_t * timerid );

int timer_settime( timer_t timerid, int flags,
                   struct itimerspec *value,
                   struct itimerspec *ovalue );
```

timer_settime() → Set a timer for later expiration.

value.it_value → specifies the delay/time for first notification.

value.it_interval → specifies re-arming interval for periodic timer notification.

Timers

What if we want to delay until a specific time in the future?

```
#include <pthread.h>
#include <time.h>

int pthread_cond_timedwait( pthread_cond_t* cond,
                            pthread_mutex_t* mutex,
                      const struct timespec* abstime );
```

POSIX does not have a timer for a specific future date.

However, we can do this if we use the timed wait function of condition variables!

We might have to create a mutex and a condition variable for nothing...

NEUTRINO RTOS

Timers

What if we want to delay until a specific time in the future?

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

int main(int argc, char* argv[]) {
  struct timespec to;
  int retval;
  // we'll wait for five seconds FROM NOW when we call
  //  pthread_cond_timedwait()
  memset(&to, 0, sizeof to);
  to.tv_sec = time(0) + 5;
  to.tv_nsec = 0;
  if (retval = pthread_mutex_lock(&m)) {exit(EXIT_FAILURE);}
  if (retval = pthread_cond_timedwait(&c, &m, &to)) {
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

NEUTRINO RTOS

Timers

Periodic Timers solves a very significant issue:

How do we get a piece of code to execute periodically?

```
#include <signal.h>
#include <time.h>

main() {
Ti = timer_create();

for (;;) {
   // use relative delay
   timer_settime( +10ms );
   // wait for timer
   pause();
   // do work
   ...
}
```

This only works if the 'work' to be done always takes the exact same time...

.. but what if this thread gets pre-empted in the middle of its work?

QNX NEUTRINO RTOS

Timers

Periodic Timers solves a very significant issue:

How do we get a piece of code to execute periodically?

```
#include <signal.h>
#include <time.h>

main() {
Ti = timer_create();
Next = clock_gettime() + 10ms
...

for (;;) {
   // use absolute delay
   pthread_cond_wait( Next );
   // do work
   ...
   Next := Next + 10ms;
}
```

This will work, but not very 'elegant'...

Clocks

```
/* This program executes a loop periodically  */
struct sigevent event;      /* Timer event.          */
struct itimerspec itime;    /* Timer specification. */
timer_t timer_id;           /* Timer object.         */
struct _pulse pulse;        /* Asynch. msg pulse.    */
int chid, coid;        /* Channel and Connection id. */

if ((chid = ChannelCreate(0)) == -1) {
    exit(EXIT_FAILURE);
}
coid = ConnectAttach( ND_LOCAL_NODE, 0, chid,
                      _NTO_SIDE_CHANNEL, 0);
/* Setup timer event. */
event.sigev_notify = SIGEV_PULSE;
event.sigev_coid = coid;
event.sigev_priority = getprio(0);
event.sigev_code = _PULSE_CODE_MINAVAIL;
timer_create(CLOCK_REALTIME, &event, &timer_id);
```

Clocks

```
/* Setup and start timer. */
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = 1;
itime.it_interval.tv_sec = 0;
itime.it_interval.tv_nsec = usec_interval * 1000;
timer_settime(timer_id, 0, &itime, NULL);

/* Handle timer pulses. */
while (1) {
  if (MsgReceive(chid,&pulse,sizeof(pulse), NULL) == 0){
    if (pulse.code == _PULSE_CODE_MINAVAIL) {
       /* Insert periodic code here... */
    }
  }
}
```

Clocks

```
#include <time.h>
int timer_gettime( timer_t timerid,
                   struct itimerspec *value );

#include <signal.h>
#include <time.h>
int timer_getoverrun( timer_t timerid );
```

timer_gettime() gets the amount of time left before the specified timer is to expire, along with the timer's reload value.

timer_getoverrun() returns the timer expiration overrun count for the timer specified by timerid (max = DELAYTIMER_MAX)

Only a single signal is queued to the process for a given timer at any point in time. When a timer that has a signal pending expires, no signal is queued and a timer overrun occurs.

NEUTRINO RTOS

Clocks & Timers

For when we do not need to be exact with timings...

```
#include <unistd.h>
unsigned int alarm( unsigned int seconds );

unsigned int sleep( unsigned int seconds );
unsigned int delay( unsigned int duration );
```

sleep() → suspend execution for X seconds.

delay() → suspend execution for X milliseconds.

alarm() → causes the system to send the calling process a SIGALRM signal after a specified number of realtime seconds have elapsed. If 0 is passed, previous alarms are canceled.

NEUTRINO RTOS

Neutrino Specific System Call Timers

An often-needed timeout service provided by the OS is the ability to **specify the maximum time** the application is prepared to wait **for any given kernel call or request** to complete.

Using generic OS timer services does not work:

in the interval between the specification of the timeout and the request for the service, the thread may be preempted long enough that the specified timeout will have expired before the service is even requested.

The application will then end up requesting the service with an already lapsed timeout in effect (i.e. no timeout).

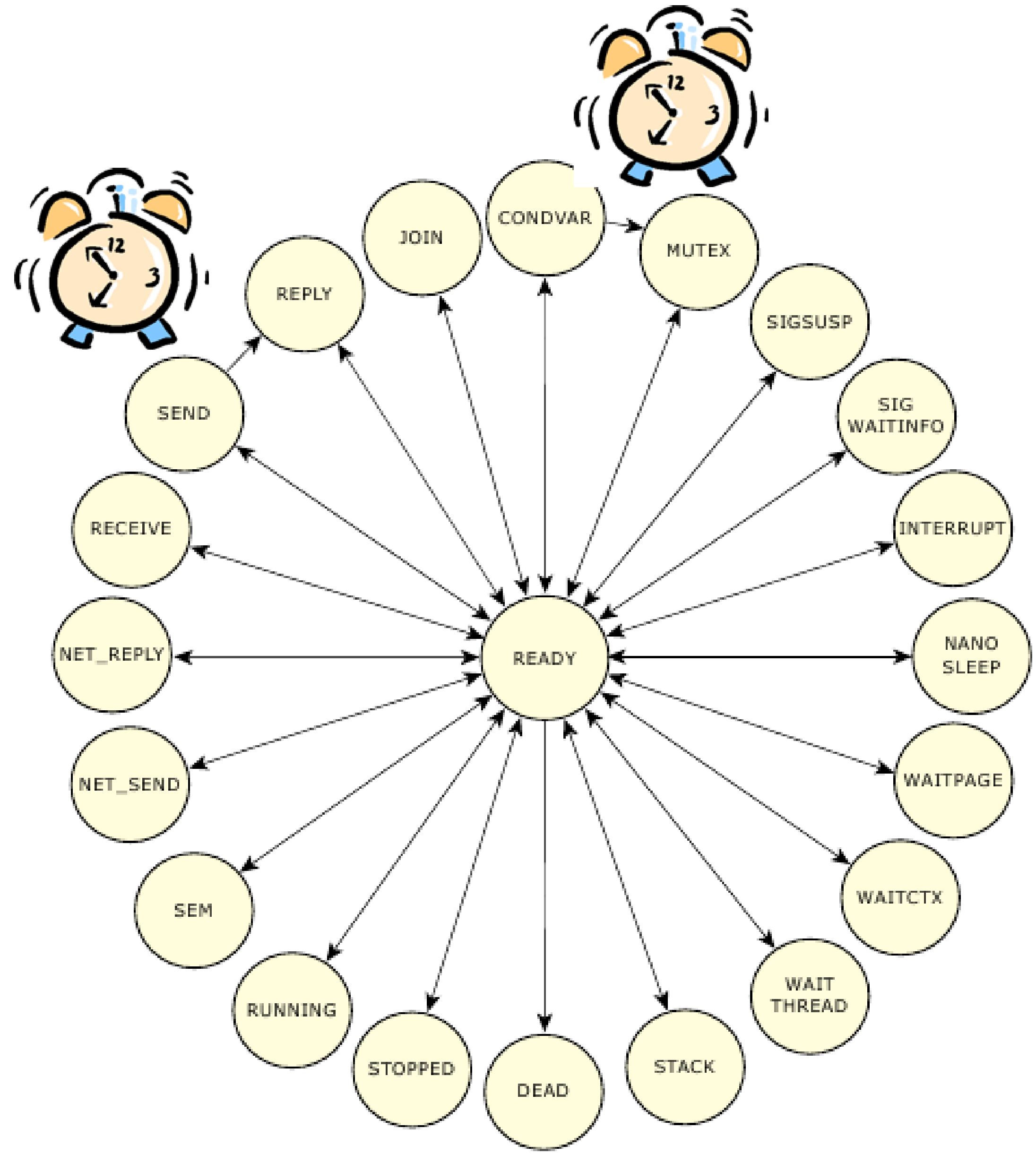
```
alarm();
//The thread gets preempted, and the alarm fires here
Blocking_OS_call();
```

Neutrino Specific System Call Timers

Instead of adding specific timed variants for each OS kernel service request, QNX Neutrino adds an automatic 'timer' for each **blocking state** a thread may be in.

A thread may define if this timer is to be used, with a single call TimerTimeout(), for each **blocking state**.

Every time a thread becomes blocked in this state the timer is started automatically!



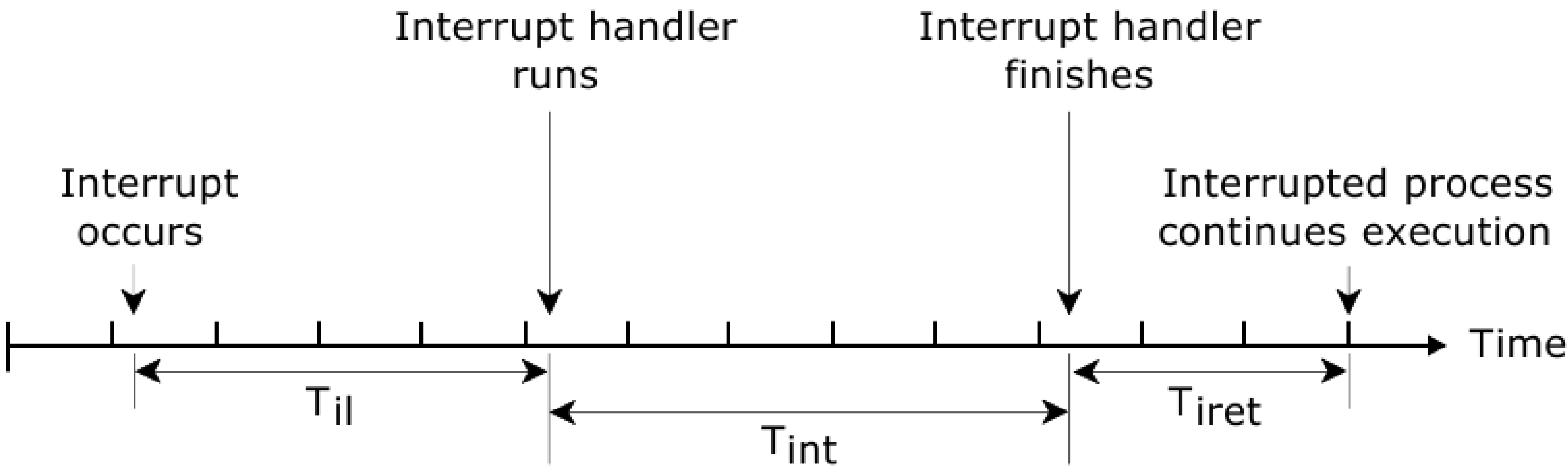
Real-Time Applications on QNX

NEUTRINO RTOS

- The QNX Architecture
- The QNX Neutrino Micro-Kernel
- Threads and Processes
- Thread CPU Scheduling
- POSIX IPC Services
- QNX Neutrino IPC Service
- Clocks and Timers
- **Interrupt Handling**

NEUTRINO RTOS

Interrupt Latency



T_{il} interrupt latency

T_{int} interrupt processing time

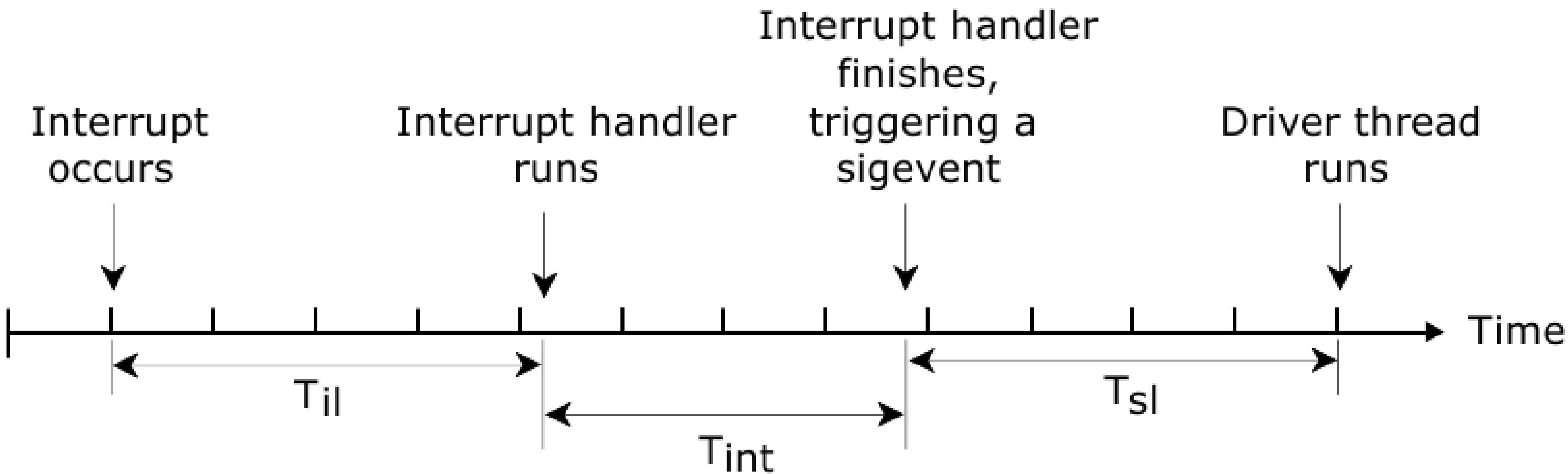
T_{iret} interrupt termination time

QNX provides a short upper bound on the Interrupt Latency.

The exact value is dependent on the hardware!

QNX NEUTRINO RTOS

Scheduling Latency



T_{il} interrupt latency

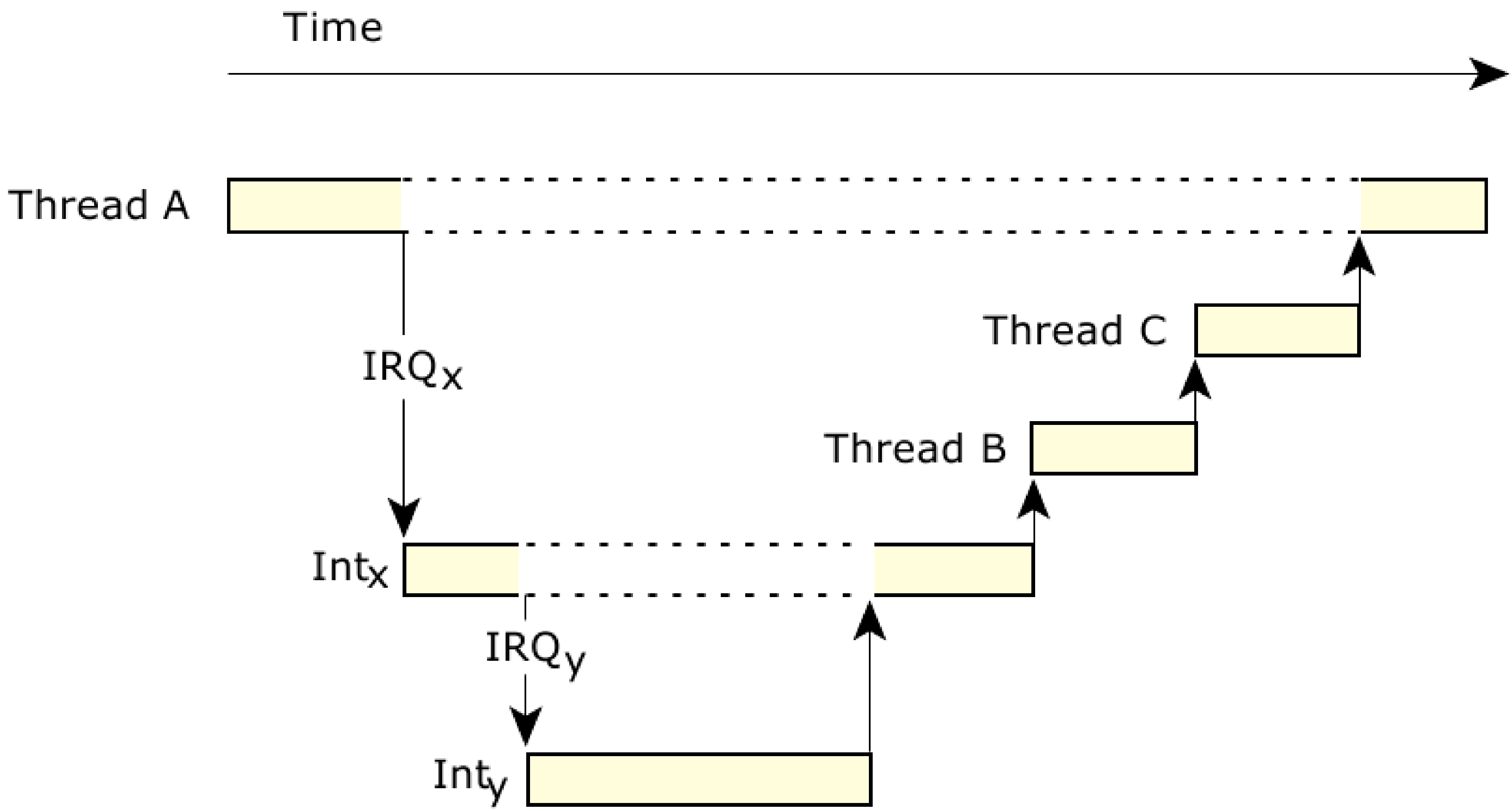
T_{int} interrupt processing time

T_{sl} scheduling latency

QNX provides a short upper bound on the Scheduling Latency.

The exact value is dependent on the hardware!

Nested Interrupts



QNX NEUTRINO RTOS

Interrupt Handlers

```
#include <sys/neutrino.h>

//attach a direct interrupt handler
InterruptAttach()
//attach an event to an interrupt
InterruptAttachEvent()
```

Writing an interrupt handler is easy, since it executed within the address space of the thread that created that handler!

No system calls are allowed inside an interrupt handler!

When multiple interrupt handlers are directly attached to the same interrupt, these are executed acording to the thread's priority.

When multiple events are attached to the same interrupt, these are handled by normal scheduling mechanism.

Interrupt Handlers

```
#include <sys/neutrino.h>

// Detach from an interrupt
InterruptDettach()
// Wait for an interrupt
InterruptWait()
// Enable hardware interrupts
InterruptEnable()      - InterruptLock()
// Disable hardware interrupts
InterruptDisable()     - InterruptUnlock()
// Mask a hardware interrupt
InterruptMask()
// UnMask a hardware interrupt
InterruptUnmask()
```

SMP safe versions!
Use spinlock!

Only threads with root priority can attach interrupt handlers

(it is very easy to freeze the system by disabling interrupts!)

Interrupt Handlers

```
struct sigevent event;
volatile unsigned counter;
const struct sigevent *handler( void *area, int id ) {
  if ( ++counter < 100 ) return( NULL )
  counter = 0;         // Wake up the thread
  return(&event);      // every 100th interrupt
}
int main() {
  int i, id;
  ThreadCtl( _NTO_TCTL_IO, 0 ); //Request I/O privileges
  event.sigev_notify = SIGEV_INTR;// Initialize event
  // Attach ISR vector
  id=InterruptAttach( SYSPAGE_ENTRY(qtime)->intr,
                      &handler, NULL, 0, 0 );
  for( i = 0; i < 10; ++i ) {
    // Wait for Interrupt handler to wake us up
    InterruptWait( 0, NULL );
    printf( "100 events\n" );
  }
  InterruptDetach(id); // Disconnect the ISR handler
  return 0;
}
```

Programming

Real-Time Applications on

Kysymyksiä?

질문?

Perguntas?

Въпроси?

Spørsmål?

質問ですか？

Pitanja?

Vragen?

QUESTIONS?

Frågor?

Domande?

Spørsmål?

问题？

Otázky?

Pytania?

الأسئلة؟

Fragen?

¿Preguntas?

Questions?

Întrebări?

Вопросы?

Ερωτήσεις;

Mário de Sousa

msousa@fe.up.pt