POSIX

Real-Time

Mário de Sousa

msousa@fe.up.pt

POSIX Real-Time

- The POSIX standards
- POSIX for RT Applications
- RT Linux

POSIX Standard

- POSIX → **Portable Operating System Interface for UNIX**
  - Describes the services that the OS must provide,
  - Describes the syntax and semantics of their interfaces
    (data types and function prototypes)
  - Interfaces defined at source code level => portable source code.
    Binary level portability is outside the scope of the standard.
  - Implementation of those services is not specified by the standard
    (left open for each OS vendor to decide how to achieve it)

  - Developed by IEEE, first version in 1988.
    Most recent version from 2017.

POSIX Standards

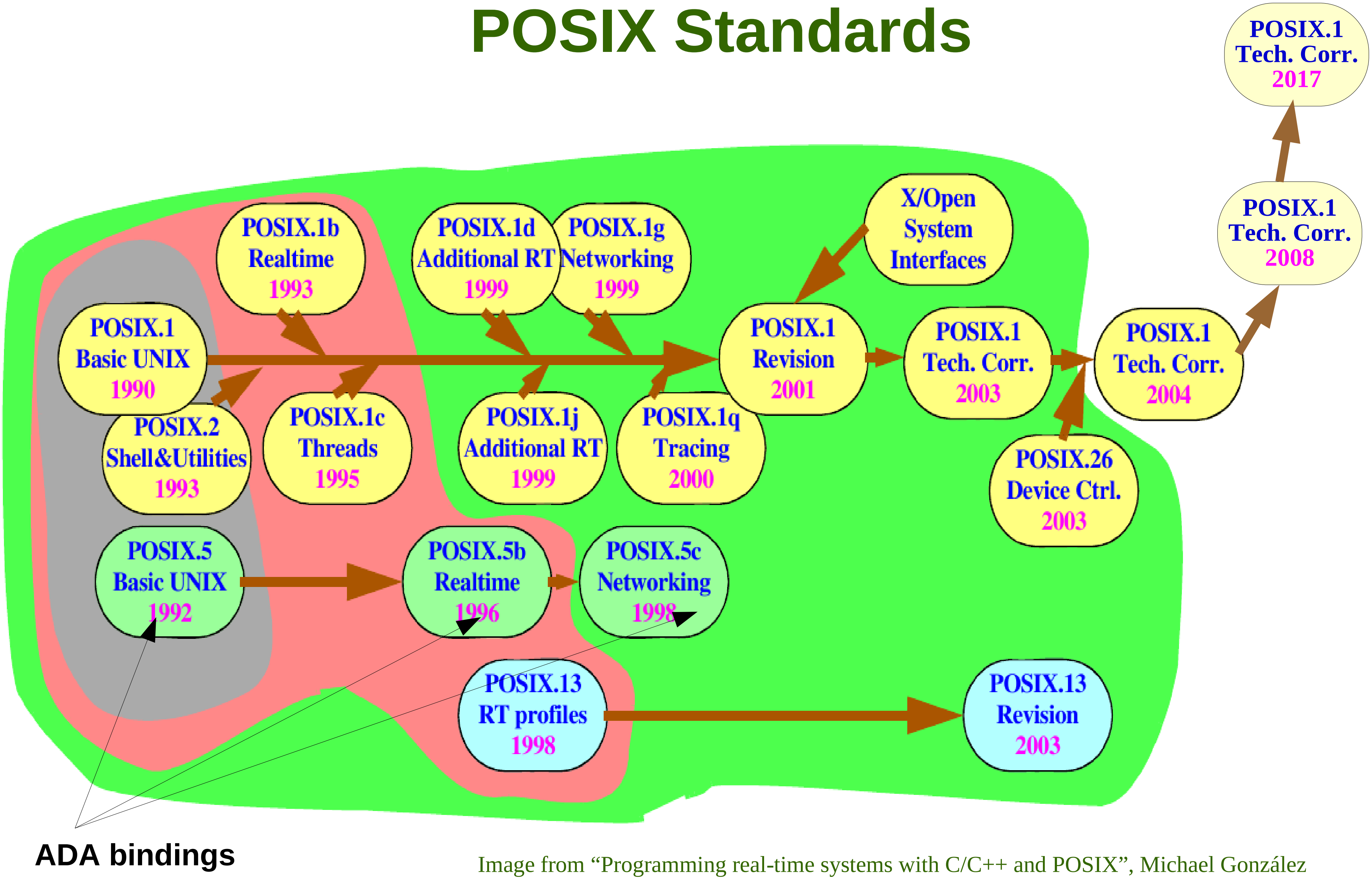


Image from “Programming real-time systems with C/C++ and POSIX”, Michael González Harbour

Mário de Sousa

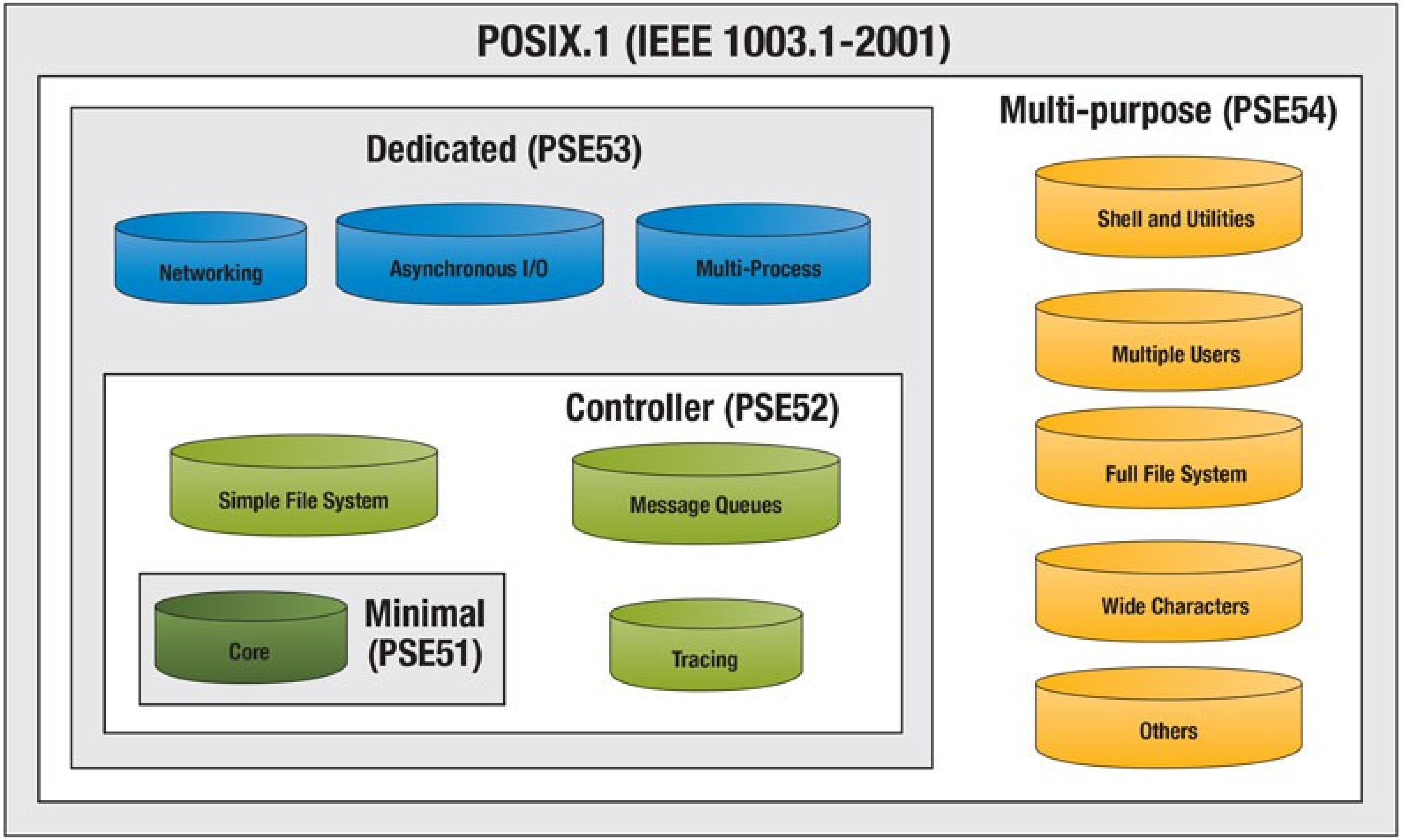
from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

Table 1: POSIX Standards

Standard	Name	Description
1003.1a	OS Definition	Basic OS interfaces; includes support for: (single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device specific, system database, pipes, FIFO, and C language
1003.1b	Real-time Extensions	Functions needed for real-time systems; includes support for: real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphores, and shared memory
1003.1c	Threads	Functions to support multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d	Additional Real-time Extensions	Additional interfaces; includes support for: new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control.
1003.1j	Advanced Real-time Extensions	More real-time functions including support for: typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21	Distributed Real-time	Functions to support real-time distributed communication; includes support for: buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols
1003.1h	High Availability	Services for Reliable, Available, and Serviceable Systems (SRASS); includes support for: logging, core dump control, shutdown/reboot, and reconfiguration

Table 2: POSIX 1003.13 Profiles

Profile	Number of Processes	Threads	File System
54	Multiple	Yes	Yes
53	Multiple	Yes	No
52	Single	Yes	Yes
51	Single	Yes	No



from "POSIX – 25 Years of Open Standard APIs" Arun Subbarao, LynuxWorks

from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

POSIX 1003.1b Real-Time Extensions

- Timers
  Periodic timers, delivery is accomplished using POSIX signals
- Priority scheduling
  Fixed priority preemptive scheduling (minimum of 32 priority levels)
- Real-time signals
  Additional signals with multiple levels of priority
- Semaphores
  Named and memory counting semaphores
- Message queues
- Shared memory
- Memory locking
  Prevent virtual memory swapping of physical memory pages (mlockall() ...)

from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

POSIX 1003.1c Threads

- Thread control
  Creation, deletion and management of individual threads
- Priority scheduling
  POSIX RT scheduling extended to scheduling on a per thread basis
- Mutexes
  Used to guard critical sections of code
  (include support for priority inheritance and priority ceiling protocols)
- Condition variables
  Used with mutexes, are used to create a synchronization point
- Signals
  Ability to deliver signals to individual threads

from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
  - Concurrency + Scheduling
  - Mutual exclusion synchronisation
  - Signal/wait synchronisation
  - Asynchronous notification
  - Message passing
  - Timing services
  - Memory management

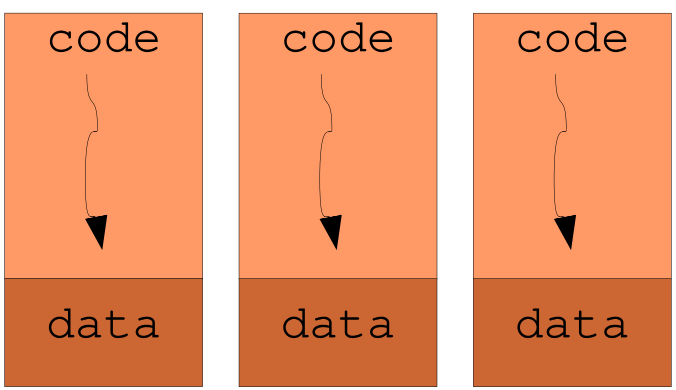
Processes vs Threads

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

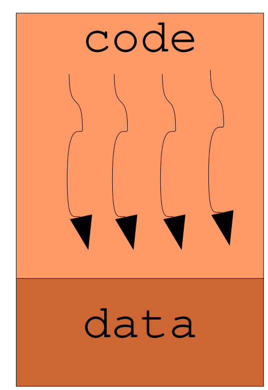
Each process may contain one or more threads that share the process's memory.

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.

Several
Single threaded processes



Single
Multi threaded process



POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
  - Concurrency + Scheduling
    - Processes
    - Threads
    - Scheduling

Process Life-Cycle

Creation

fork()

Termination

void _exit(int status)

(cancel calling process)

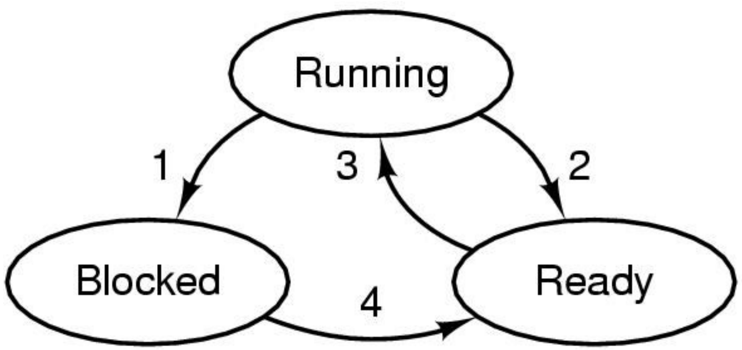
int kill(pid_t pid, int sig)

(kill another process)

the OS decides to kill it

(lack of resources, invalid operation,
invalid memory access, …)

the main() function returns



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

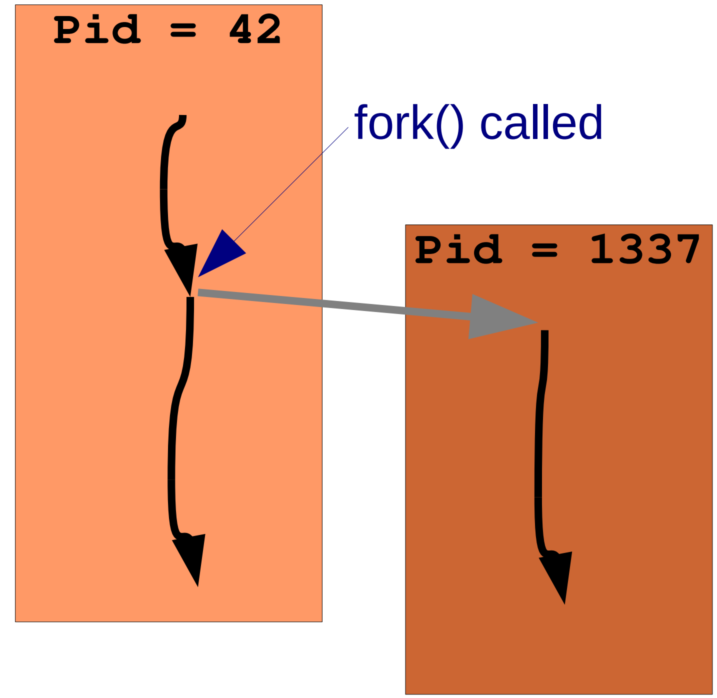
```
(...)
pid_t pid = fork();
if (pid == 0) {
  child_process_function();
} else  {
  parent_process_function();
}
```

The fork() function returns:

0 on the child process.

the child process ID, in the parent process.

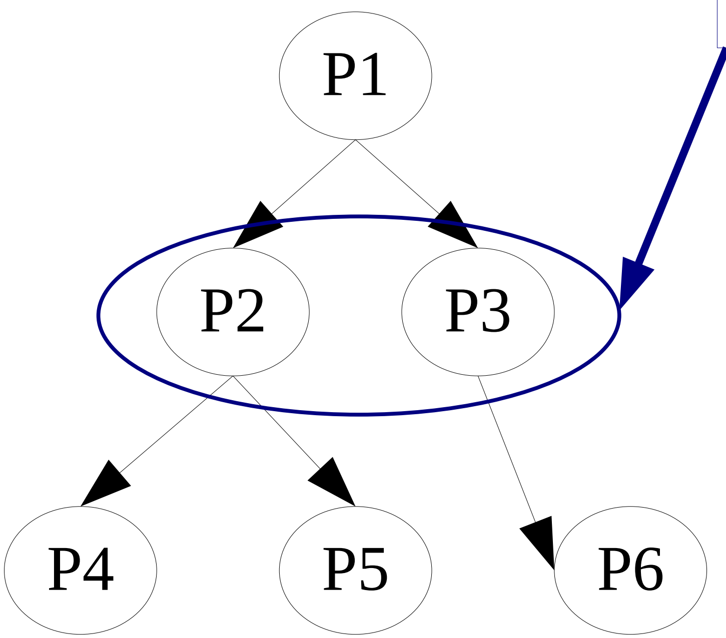
Process Creation



Process Relations

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

Parents with the same parent are known as a process group.




```
$pstree
 init-+-apmd
      |-atd
      |-bdflush
      |-crond
      |-kapmd
      |-kdeinit-+-artsd
      |         |-evolution-alarm
      |         |-kdeinit---bash---pstree
      |         |-kdeinit---bash---ssh
      |         `-soffice.bin---soffice.bin
      |-12*[kdeinit]
  (...)
```

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 *file, char *const argv[] );
int execve(const char *file, char *const argv[], char *const envp[]);
```

`argv` and **`envp`** specify the arguments to pass the new program's **`main()`** function

```
(...)
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()`

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 (IPC) primitives:

- Pipes
- Sockets
- Shared Memory
- Thread Synchronisation primitives
  - 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.
    - (e.g. QNX Neutrino allows inter-process thread synchronisation)

POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
  - Concurrency + Scheduling
    - Processes
    - Threads
    - Scheduling

Thread Life-Cycle

Creation

pthread_create(...)

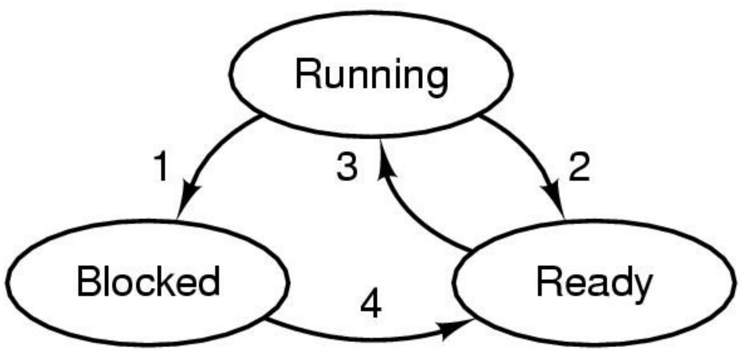
Termination

pthread_exit(...)
(cancel calling thread)

pthread_cancel(...)
(cancel another thread)

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

the thread function returns



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

Thread Life-Cycle
Reality: Example from QNX

Creation

pthread_create(...)

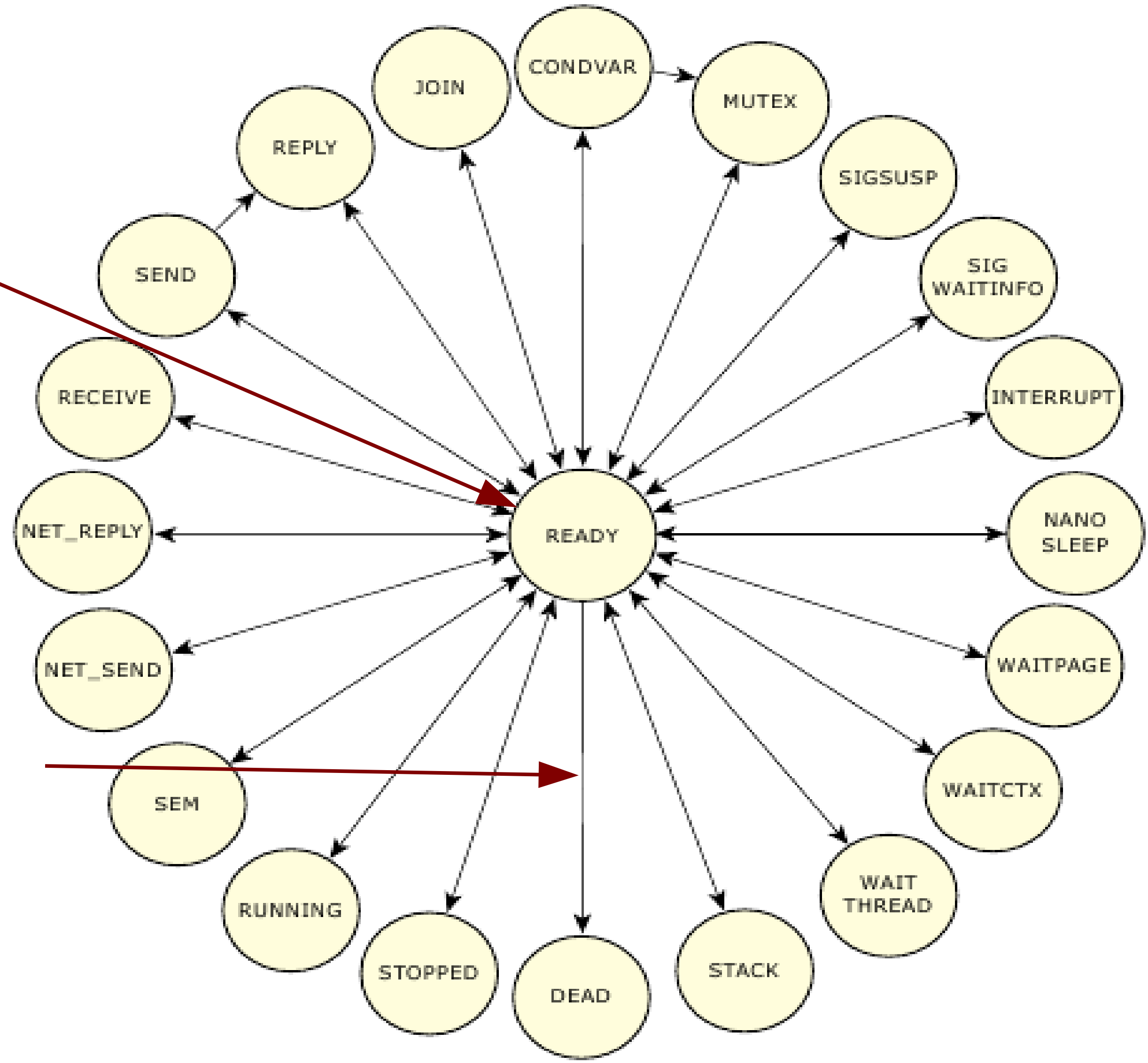
Termination

pthread_exit(...)

(cancel calling thread)

pthread_cancel(...)

(cancel another thread)

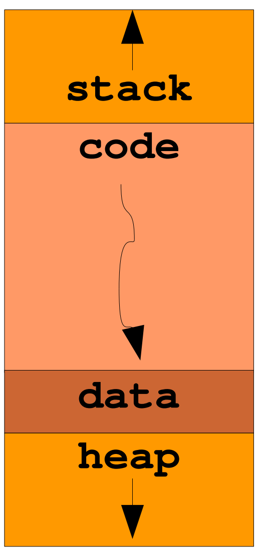


Threads

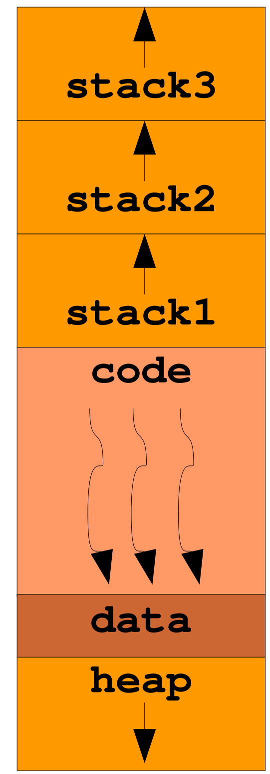
Threads within a process share everything within the process's address space (e.g. open files)

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 – Thread ID (a unique number within the process)
- Signal mask
- Thread local storage
- Cancellation handlers
  - (callback functions that are executed when the thread terminates)



Single threaded process

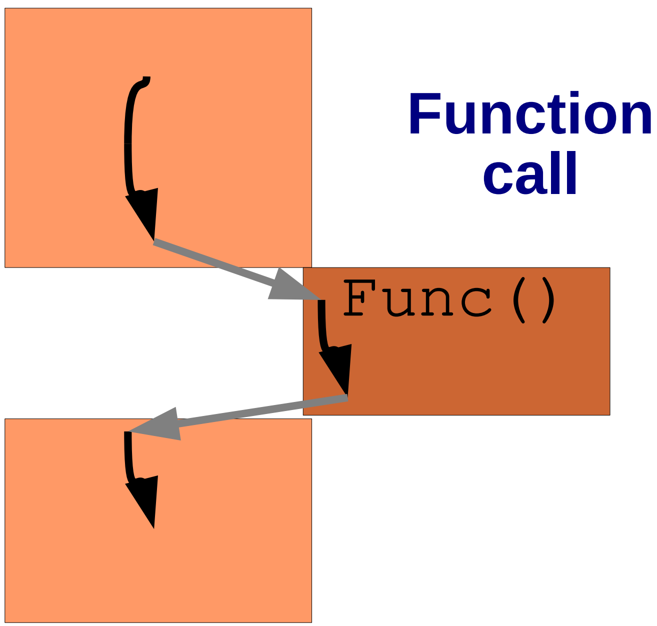


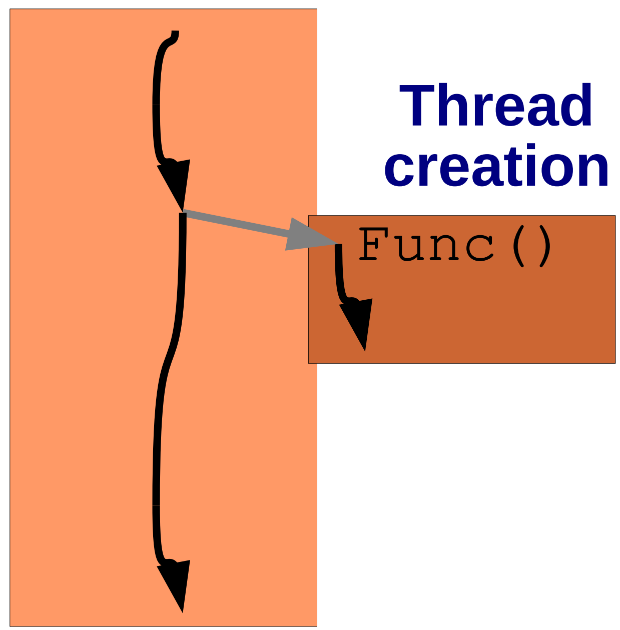
Multi threaded process

Mário de Sousa

Thread Creation

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





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
```

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;        //using a single parm variable -> does not work
  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);
    ...
```

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;
}
```

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 is available to all threads in the process and binds an optional destructor function 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.

POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
  - Concurrency + Scheduling
    - Processes
    - Threads
    - Scheduling

Concurrency + Scheduling

- Concurrency
  - Processes or Threads
- Scheduling Algorithms...
  - SCHED_OTHER → Not Fixed Priority. No good for RT!!
  - SCHED_FIFO (FIFO for threads/processes of same priority)
  - SCHED_RR (Like FIFO, but with max. quantum execution time)
  - SCHED_SS (Sporadic Server → good for aperiodic tasks)

Fixed Priority

Support is optional.
Config. Param.:
- replenishment_period
- budget
- high priority
- low priority
- max pending replenishments

All algorithms are compatible, and may co-exist!

Concurrency + Scheduling

- Concurrency
  - Processes or Threads
- Scheduling Algorithms...
  - ...Applied per thread or per process **-> may co-exist !**
    - System contention scope
      All threads in the system compete, regardless of the process to which they belong.
    - Process contention scope
      The scheduler works at two levels: It first chooses a process according to its priority, and then chooses the highest priority thread of that process.
    - Mixed contention scopes
      some threads have a “system” scope, and other threads have a “process” scope.

Thread Scheduling

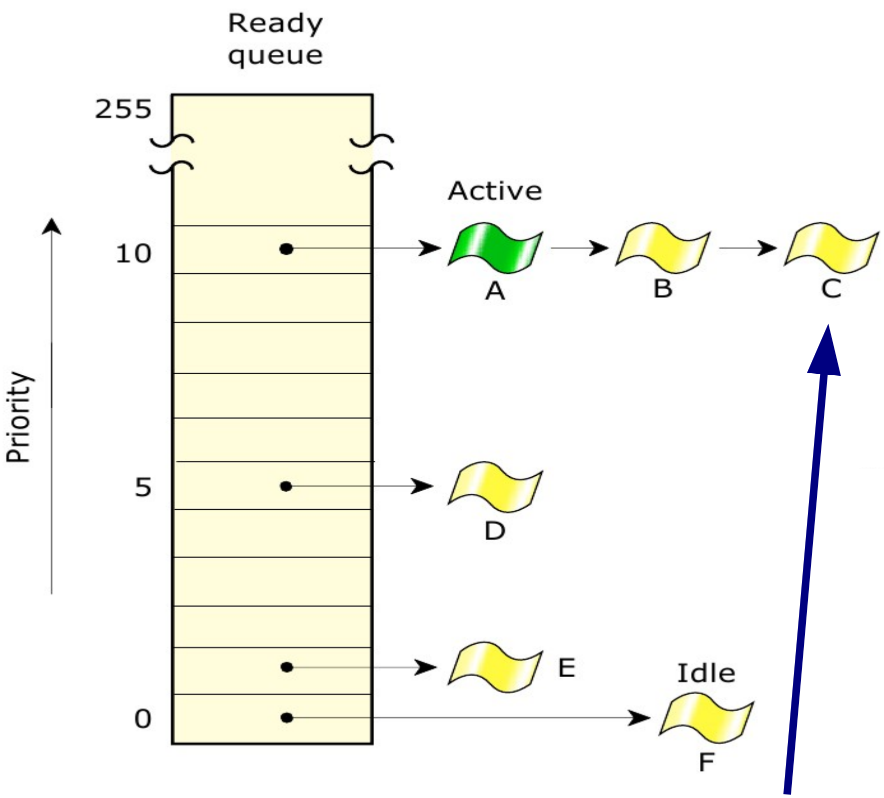
The scheduler will perform a context switch from one thread to another whenever the running thread

- 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 (e.g. timer expires).
- yields
  - By calling sched_yield()

```
#include <sched.h>

int sched_yield( void );
```

Thread Scheduling – Fixed 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 between
  lowest: sched_get_priority_min(policy)
  highest: sched_get_priority_max(policy)

- At least 32 distinct priority levels

  QNX priorities:

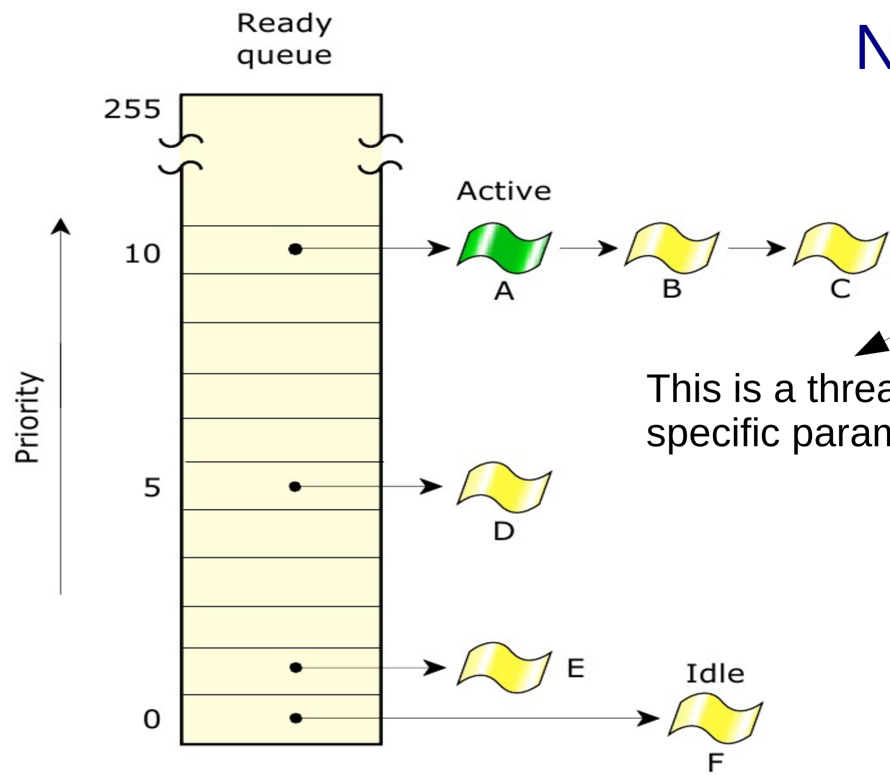
  0: idle thread

  1-63: non-root threads

  1-255: root threads

- Multiple threads may have the same priority...

Thread Priority



This is a thread-specific parameter!

Note that:

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.

- FIFO:
  thread will execute until it decides to yield(), or it blocks waiting for a resource.

- FIFO:
  thread will execute until it decides to yield(), or it blocks waiting for a resource, or it times out on amount of time on the CPU

Thread Scheduling

FIFO

A thread runs until

- it voluntarily relinquishes control of the CPU (i.e. blocks or yields)

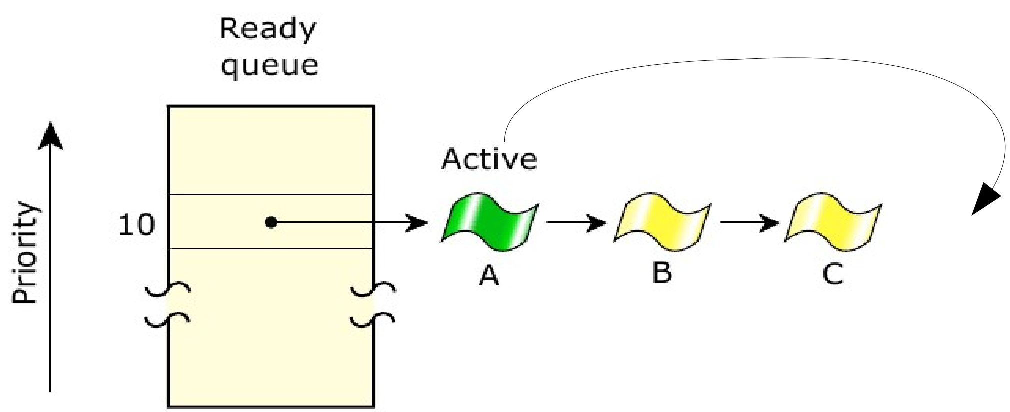
Round-Robin

A thread runs until

- it voluntarily relinquishes control of the CPU (i.e. blocks or yields),

OR

- it exhausts its time-slice



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

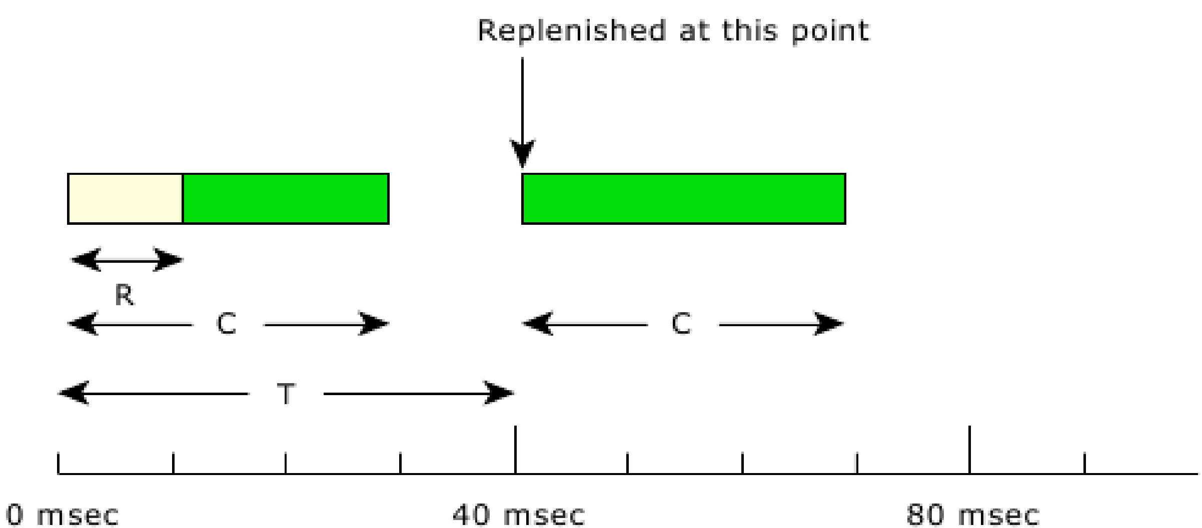
Sporadic Scheduling (SCHED_SS)

A thread has two priorities:

- N: normal priority
- L: low priority

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

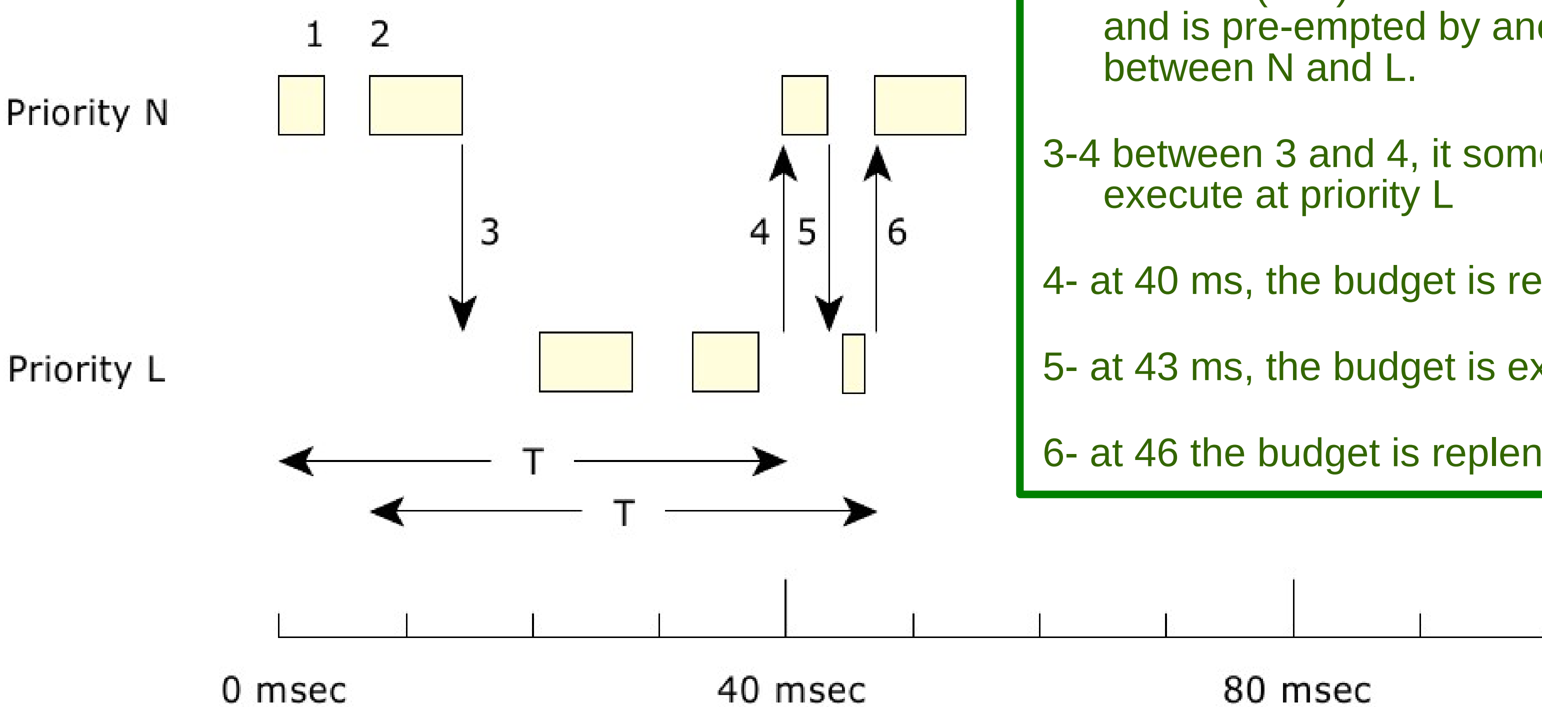
- at N for C time units
- at L if C is exhausted



Sporadic Scheduling (SCHED_SS)

T = 40 ms

C = 10 ms



1- the thread is blocked after executing for 3 ms

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

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.

Concurrency + Scheduling

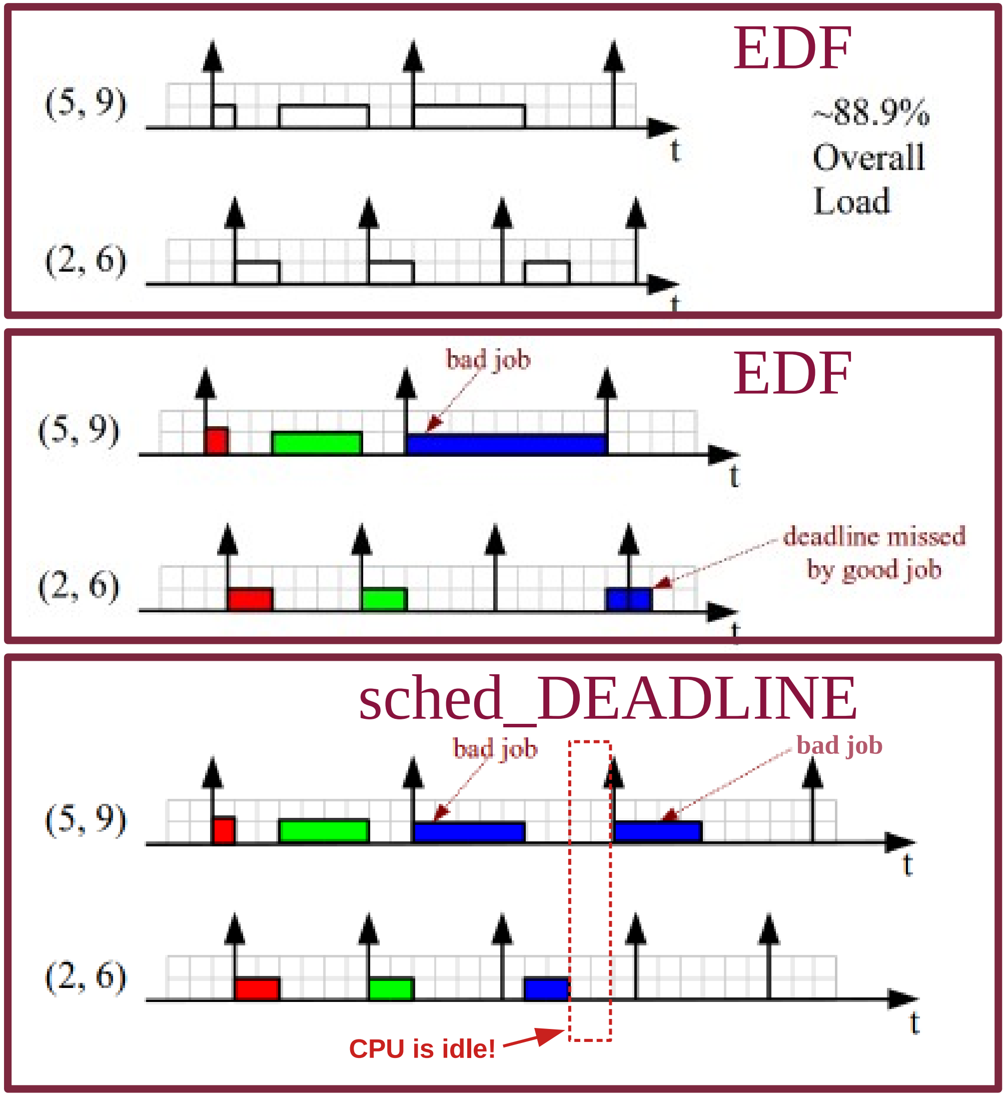
POSIX		LINUX
...	→	SCHED_IDLE
– SCHED_OTHER	→	SCHED_OTHER, SCHED_BATCH
– SCHED_FIFO	→	SCHED_FIFO
– SCHED_RR	→	SCHED_RR
– SCHED_SS	→	
–	→	SCHED_DEADLINE

Fixed Priority

Implements EDF with CBS
(Constant Bandwidth Server)
Config. Param.:
- runtime
- deadline
- period

Deadline Scheduling (SCHED_DEADLINE)

Documentation: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.rst



Three configuration paramaters:

Runtime: expected execution time

Used as the Capacity reserved every period

Deadline: deadline for each activation

Period: expected activation period

Used for admission control,
Used as replenishment perio

Implements Constant Bandwidth Server

But capacity (and deadline) is only reset at the end of the period !

Remember, traditional CBS resets capacity (and deadline) as soon as capacity is exausted.

Mário de Sousa

Images from: https://lwn.net/Articles/398470/

Deadline Scheduling (SCHED_DEADLINE)

WARNING

A process configured as SCHED_DEADLINE must not fork, otherwise bandwidth reservation gaurantees do not hold!

Admission control is only done when scheduling algorithm (or parameters) are changed. Fork creates a new process with same reservation, but bypassing admission control.

Three configuration paramaters:

Runtime: expected execution time

Used as the Capacity reserved every period

Deadline: deadline for each activation

Period: expected activation period

Used for admission control,
Used as replenishment perio

Implements Constant Bandwidth Server

But capacity (and deadline) is only reset at the end of the period !

Remember, traditional CBS resets capacity (and deadline) as soon as capacity is exausted.

Deadline Scheduling (SCHED_DEADLINE)

```
 int main (int argc, char **argv)     {
        int ret;
        int flags = 0;
        struct sched_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.size = sizeof(attr);
        /* This creates a 200ms / 1s reservation */
        attr.sched_policy   = SCHED_DEADLINE;
        attr.sched_runtime  =  200000000; /*200 ms*/
        attr.sched_deadline = attr.sched_period = 1000000000; /*1 s*/
        ret = sched_setattr(0, &attr, flags);
        if (ret < 0) {
            perror("sched_setattr failed to set the priorities");
            exit(-1);
        }
        do_useful_computation();
        exit(0);
}
```


from: https://lwn.net/Articles/743946/

Deadline Scheduling (SCHED_DEADLINE)

```
int main (int argc, char **argv)     {
        int ret;
        int flags = 0;
        struct sched_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.size = sizeof(attr);
        /* This creates a 200ms / 1s reservation */
        attr.sched_policy   = SCHED_DEADLINE;
        attr.sched_runtime  =  200000000; /*200 ms*/
        attr.sched_deadline = attr.sched_period = 1000000000; /*1 s*/
        ret = sched_setattr(0, &attr, flags);
        if (ret < 0) {
            perror("sched_setattr failed to set the pri
            exit(-1);
        }
        do_useful_computation();
        exit(0);
}
```

Run as Bandwidth Server

```
void do_useful_computation(void) {
  /* do_the_computation_without_blocking */
  while (1) {
    /* do whatever we need to do, no blocking needed */
  }
}
```

from: https://lwn.net/Articles/743946/

Deadline Scheduling (SCHED_DEADLINE)

```
int main (int argc, char **argv)     {
        int ret;
        int flags = 0;
        struct sched_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.size = sizeof(attr);
        /* This creates a 200ms / 1s reservation */
        attr.sched_policy   = SCHED_DEADLINE;
        attr.sched_runtime  =  200000000; /*200 ms*/
        attr.sched_deadline = attr.sched_period = 1000000000; /*1 s*/
        ret = sched_setattr(0, &attr, flags);
        if (ret < 0) {
            perror("sched_setattr failed to s
            exit(-1);
        }
        do_useful_computation();
        exit(0);
}
```

**Run periodic task,
with execution time enforcement**

```
void do_useful_computation(void) {
  /* do periodic computation, with execution time enforcement */
  while (1) {
          do_the_computation();
          /*
           * Notify the scheduler the end of the computation
           * This syscall will block until the next replenishment
           */
          sched_yield();
  }
}
```

WARNING

sched_yield() has special semantics when used with SCHED_DEADLINE; it suspends the task until the next replenishment period!

from: https://lwn.net/Articles/743946/

Deadline Scheduling (SCHED_DEADLINE)

Run an Aperiodic task

```
int main (int argc, char **argv)     {
       int ret;
       int flags = 0;
       struct sched_attr attr;
       memset(&attr, 0, sizeof(attr));
       attr.size = sizeof(attr);
       /* This creates a 200ms / 1s reservation */
       attr.sched_policy   = SCHED_DEADLINE;
       attr.sched_runtime  =  200000000; /*200 ms*/
       attr.sched_deadline = attr.sched_period = 1000000000; /*1 s*/
       ret = sched_setattr(0, &attr, flags);
       if (ret < 0) {
           perror("sched_setattr failed to s
           exit(-1);
       }
       do_useful_computation();
       exit(0);
}
```

```
void do_useful_computation(void) {
  /* run aperiodic task */
  while (1) {
          /*
           * Block in a blocking system call waiting for wakeup event.
           */
          block_waiting_for_the_next_event();
          process_the_data();
          produce_the_result();
  }
}
```

Mário de Sousa

POSIX Thread Interface: libpthreads

```
int pthread_create( pthread_t *id,
                    const pthread_att_t attr,
                    void *(*start_fn)(void *),
                    void *arg)
```

- *id
  - initialized with the new thread's identifier;
- *attr
  - data structure used to configure pthread creation semantics.
  - May be initialized with default values

    ```
    int pthread_attr_init(pthread_attr_t *attr)
    ```

POSIX Thread Interface: libpthreads

```
int pthread_create( pthread_t *id,
                    const pthread_att_t attr,
                    void *(*thr_fun)(void *),
                    void *arg)
```

- *thr_fun
  - The function to be called
  - Must have the following prototype: `void *thr_fun(void *)`
- *arg
  - The data value to pass the function thr_fun().

POSIX Thread Interface: libpthreads

```
#include <pthread.h>
void *my_fun(void *arg) {
      ...
}

int main(void) {
   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, &my_arg);
   ...
```

POSIX Thread Interface: libpthreads

`void pthread_exit(void *value ptr)`

Terminate the thread

`int pthread_join(pthread t thread, void **value prt)`

Wait until thread identified by **`thread`** terminates.

`int pthread_detach(pthread_t thread);`

Resources are released back to the system without the need for another thread to join with the terminated thread.

`int pthread_equal(pthread_t t1, pthread_t t2);`

Compares two thread identifiers.

POSIX Thread Interface: libpthreads

```
/* initialize thread attributes structure */
int pthread_attr_init(pthread_attr_t *attr);

/* destroy thread attributes structure */
int pthread_attr_destroy(pthread_attr_t *attr);
```

```
/* Set/Get thread detach state attribute */
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

`detachstate:`

`PTHREAD_CREATE_DETACHED` → created threads in a detached state.
i.e. thread destroys all local data when it terminates
Without waiting pthread_join() to be called.

`PTHREAD_CREATE_JOINABLE` → created threads in a joinable state.
i.e. possible to call pthread_join() on the thread.

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling policy */
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
```

policy:

- **SCHED_FIFO** → created threads using FIFO scheduling algorithm
- **SCHED_RR** → created threads using RR scheduling algorithm
- **SCHED_OTHER** → created threads using OTHER scheduling algorithm

---

```
/* Set thread scheduling priority */
int pthread_setschedprio(pthread_t thread, int prio);
```

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling policy and priority */
int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);

int pthread_getschedparam(pthread_t thread,
                          int *policy,
                          struct sched_param *param);



struct sched_param {
    int sched_priority;/* Scheduling priority */
};
```

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling contention scope */
int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

scope:

PTHREAD_SCOPE_SYSTEM → **created thread uses scheduling contention using system scope**

PTHREAD_SCOPE_PROCESS → **created thread uses scheduling contention using process scope**

POSIX Thread Interface: libpthreads

```
/* Set/Get thread stack size attribute */
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);

int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize);
```

stacksize → minimum size in bytes

```
------------------------------------------------------------------------

/* Set/Get thread stack address */int
pthread_attr_setstackaddr(pthread_attr_t *attr,
                          void *stackaddr);

int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **stackaddr);
```

POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
  - Concurrency + Scheduling
  - Mutual exclusion synchronisation
  - Signal/wait synchronisation
  - Asynchronous notification
  - Message passing
  - Timing services
  - Memory management

Mutual Exclusion Synchronisation

Where tasks must coordinate to atomically access a common resource

Mutex

- Protecting against Unbounded Priority Inversion...
  - No protection (Priority Inheritance - PTHREAD_PRIO_NONE)
  - immediate priority ceiling (Priority Protection - PTHREAD_PRIO_PROTECT)
    good for static systems where it is possible to determine a priority ceiling
  - priority inheritance (Priority Inheritance - PTHREAD_PRIO_INHERIT)
    useful in dynamic systems where it is impossible to assign a ceiling.
- Implemented as a variable
  - all threads/processes accessing the mutex must be able to access the mutex var.
    => using mutexes between processes requires mutex var be placed in shared memory
    [mmap()]

POSIX Mutexes

- A mutex is a variable of type pthread mutex_t

```
#include <pthread.h>
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;
```

```
void thread1(void *arg) {
  /* outside critical section */
  pthread_mutex_lock(&my_lock);
  /* within critical section */
  pthread_mutex_unlock(&my_lock);
  /* outside critical section */
}
```

```
void thread2(void *arg) {
  /* outside critical section */
  pthread_mutex_lock(&my_lock);
  /* within critical section */
  pthread_mutex_unlock(&my_lock);
  /* outside critical section */
}
```

Mutexes

```
#include <pthread.h>

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

NOTE 1: A mutex must be initialized before use.

```
int pthread_mutex_lock   (pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

NOTE 2: pthread_mutex_trylock() attempts to lock but does not block the calling thread if the mutex is already locked, unlike pthread mutex lock().

Mutexes

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

`pthread_mutexattr_destroy, pthread_mutexattr_init`

→ destroy and initialize the mutex attributes object

`pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling`

→ get and set the prioceiling attribute of the mutex attributes object (REALTIME THREADS)

`pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol`

→ get and set the protocol attribute of the mutex attributes object (REALTIME THREADS)

(PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT)

`pthread_mutexattr_getpshared, pthread_mutexattr_setpshared`

→ get and set the process-shared attribute (to allow sharing between processes)

`pthread_mutexattr_getrobust, pthread_mutexattr_setrobust`

→ get and set the mutex robust attribute (what to do if thread is terminated while holding mutex:
(PTHREAD_MUTEX_STALLED → do nothing; PTHREAD_MUTEX_ROBUST → notify next thread attempting a mutex_lock())

`pthread_mutexattr_gettype, pthread_mutexattr_settype`

→ get and set the mutex type attribute (changes semantics of pthread_lock(): allow recursive locking, ...)

(PTHREAD_MUTEX_NORMAL, MUTEX_ERRORCHECK, MUTEX_RECURSIVE, MUTEX_DEFAULT)

Mutexes

Semantics of: `pthread_mutex_lock()`
`pthread_mutex_unlock()`

Mutex Type	Robustness	Relock When Owner	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive (counting lock)	error returned
DEFAULT	non-robust	undefined	undefined
DEFAULT	robust	undefined	error returned

POSIX standard: IEEE 1003

Signal/Wait Synchronisation

Where tasks must synchronise the execution of actions

- Counting Semaphores
- Condition Variables
  - Used in conjunction with a mutex

    Allows checking of complex synchronisation conditions while mutex is held

Semaphores

- A semaphore is a variable of type sem_t

```
#include <semaphore.h>
sem_t my_semaphore;

int sem_init(sem_t *sem, int pshared,
                unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);    /* down */
int sem_trywait(sem_t *sem); /* down, no wait */
int sem_timedwait(sem_t *sem, … abstime);
int sem_post(sem_t *sem);    /* up */
int sem_getvalue(sem_t *sem, int *sval); /* get current value */
```

Asynchronous Notification

Where tasks must be asynchronously notified of event occurrence

- Signals
  - When issued, a signal handler function is executed
  - Signal is sent to any of the threads interested in that signal.
    Best is to have a single thread interested in each signal.

Sending a signal:

kill: `kill(pid_t pid, int sig)`

→ send a signal to a process or a group of processes

killpg: `killpg(pid_t pgrp, int sig);`

→ send a signal to a process group

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

→ send a signal to a thread

sigqueue: `sigqueue(pid_t pid, int signo, union sigval value)`

→ send a signal to a process, including a value

Asynchronous Notification

Where tasks must be asynchronously notified of event occurrence

Receiving a signal:

sigwait: `sigwait(const sigset_t *restrict set, int *restrict sig)`

→ wait for queued signals (only waits for signals specified in sigset)

sigaction: `sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact)`

→ specify action to take when receiving a signal

```
struct sigaction

    void(*) (int)                        sa_handler
```

→ Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.

```
    sigset_t                             sa_mask
```

→ set of signals to be blocked during execution of signal-catching function.

```
    int                                  sa_flags
```

→ Special flags to affect behavior of signal.

```
    void(*) (int, siginfo_t *, void *)   sa_sigaction
```

→ Pointer to a signal-catching function.

Message Passing

Asynchronously pass data between tasks, using message queues

Message Queues

- Message passing between processes or threads
- Supports
  - variable sized messages
  - polling for message availability,
  - block while waiting for message (may have timeout)

Timing Services

Synchronise with Time!

- Clocks
  - CLOCK_REALTIME
    Represents the oficial time - subject to changes
    (e.g. setting the clock, adjusting by clock synchronisation services, ...)
  - CLOCK_MONOTONIC
    Monotonic, at constant rate
    (like 'realtime' but not subject to any adjustments)
  - Execution time clocks
    Based on execution time of system, process, thread, etc...
    CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Timing Services

Synchronise with Time!

- Clocks

  Available services based on these clocks:

  - Sleep until a clock reaches an absolute time [clock_nanosleep()]
  - Sleep for some (relative) time interval [clock_nanosleep(), nanosleep()]
  - Create a timer (software entity) that will notify...
    - notify when clock reaches an absolute time
    - notify when an interval has elapsed.
    - expire periodically.

Execution time clocks may be used to monitor the CPU usage from a given thread, and make sure it does not produce overload situation

Periodic Thread: using clock_nanosleep()

```
#include <time.h>

int       nanosleep(const struct timespec *req, struct timespec *rem);
int clock_nanosleep(clockid_t clock_id,
                    int flags,
                    const struct timespec *request,
                    struct timespec *remain);
```

```
void* thread_function(void* arg) {
  struct timespec period = {0 /*sec*/, 50*1000000 /*ns*/};
  struct timespec next_start;

  clock_gettime(CLOCK_MONOTONIC, &next_start);

  while (1) {  //infinite loop
    next_start += period;  //not correct C code!
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                    &next_start, NULL);
    // … do work …
  }
}
```

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

Memory management

Manage unpredictable memory management

- Swapping to disk
  - Allows locking memory into RAM (mlockall())
  - Other memmory management functions:
    - mlock(), mmap(), munmap(), ...
  - Should also consider allocating all required memory at startup (malloc)
    Not POSIX specific!!

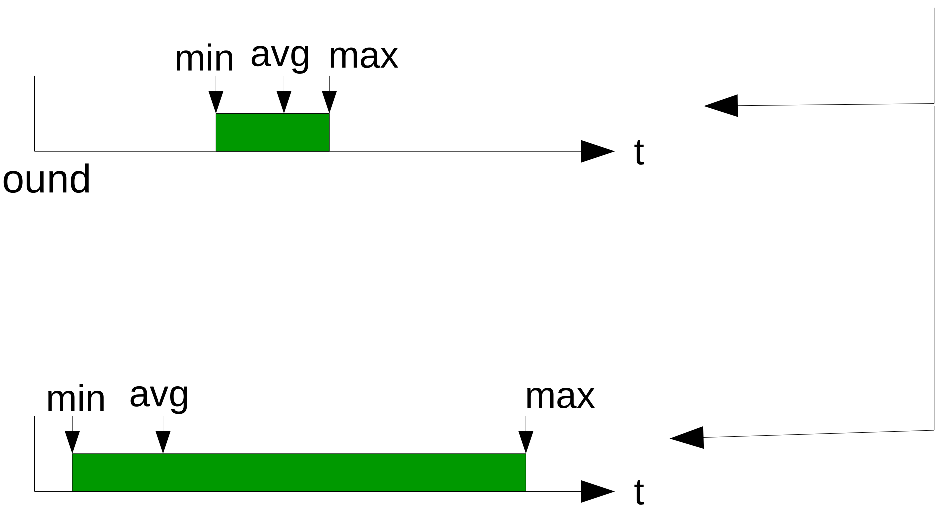
POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
- Real-Time Linux
  - Overview

LINUX & Real-Time

Contradictory requirements

- Real-Time Operating System
  - Optimised for time determinism
    - Optimize (minimize) execution time upper bound
    - may be slower on average



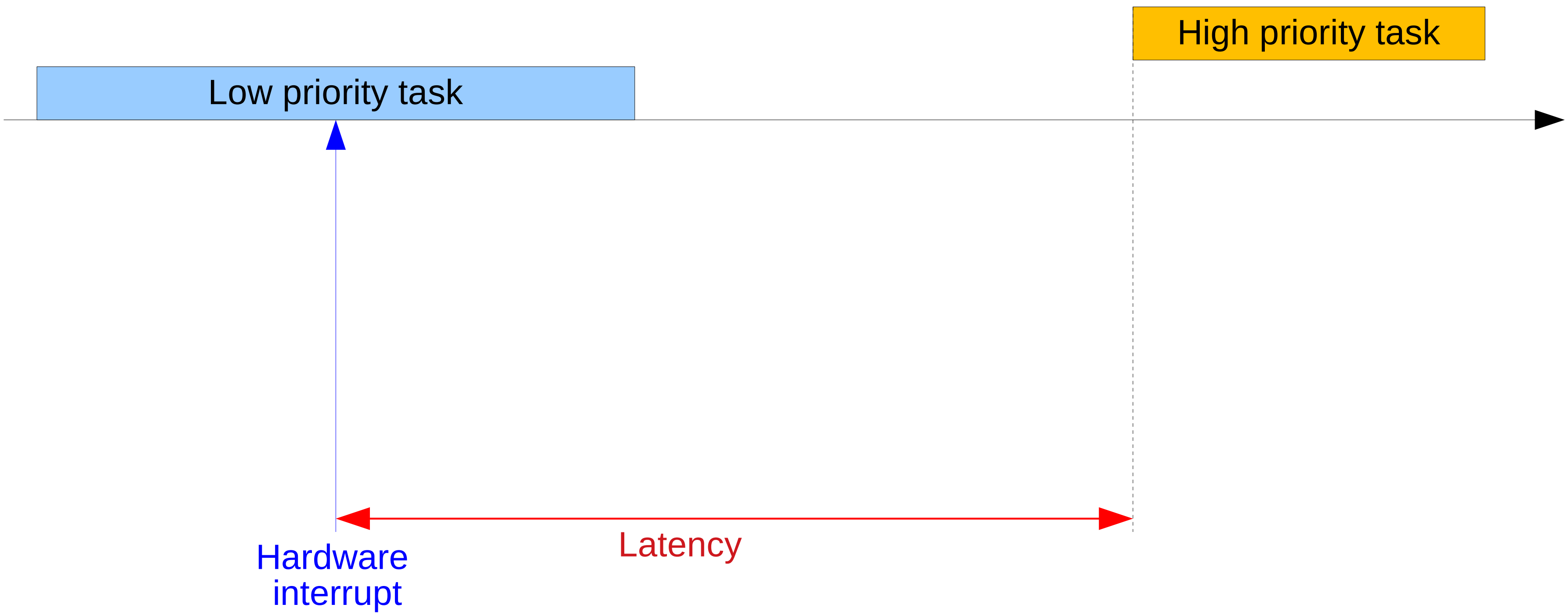
- Standard Linux
  - Optimised for throughput
    - maximize average amount of work done using available resources (CPU, memory, …)
    - Time determinism not taken into account
- Real-Time Linux → 2 approaches
  - (1) Improve kernel so it provides (shorter) bounded latencies
  - (2) Add layer below linux with RT-scheduler
    - Method used by RTAI, RTLinux, and Xenomai

POSIX Processes and Threads

- The POSIX standards
- POSIX for RT Applications
- Real-Time Linux
  - Overview
  - RT_PREEMPT

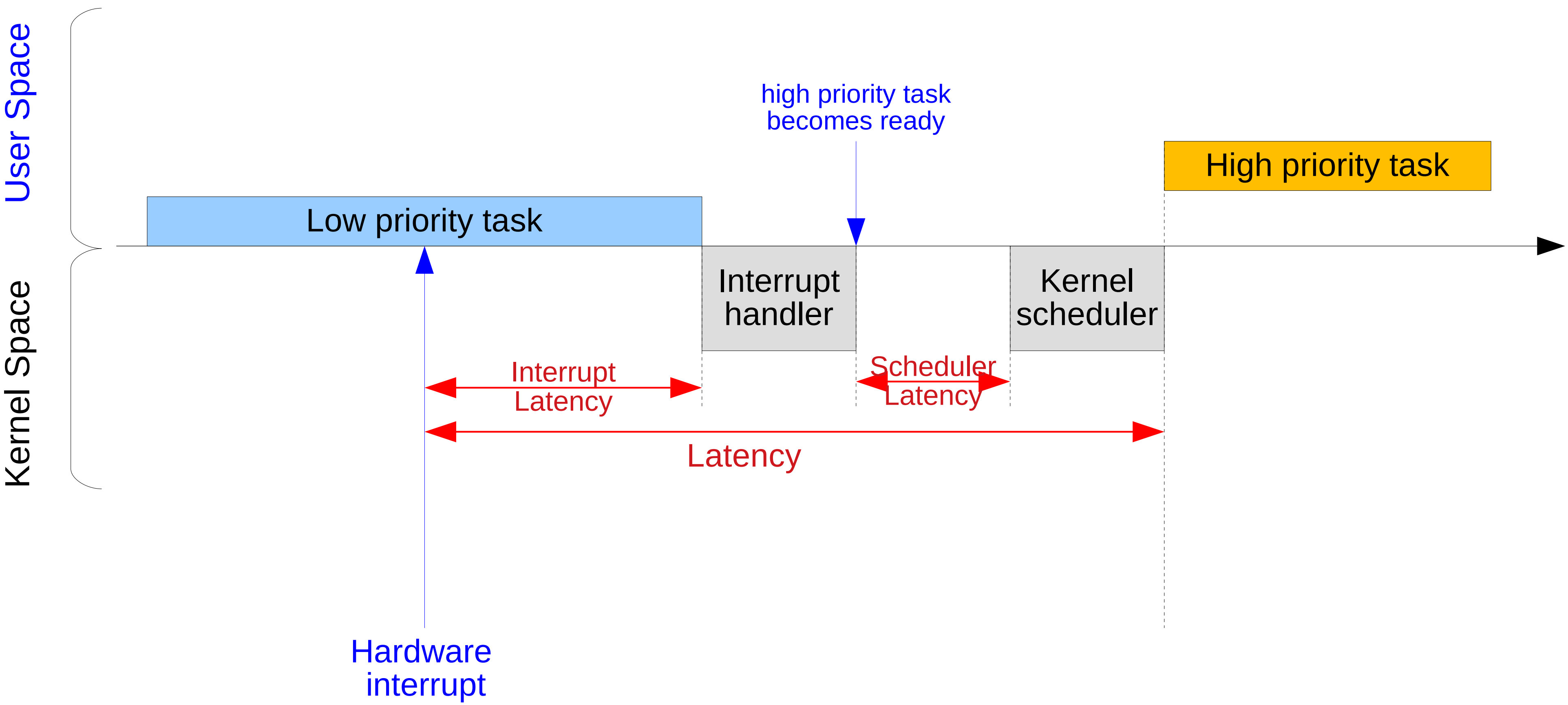
LINUX Kernel → PREEMPT_RT patch

- Typical RT Requirement
  - React to an external physical event (hardware interrupt) within a maximum delay (latency)



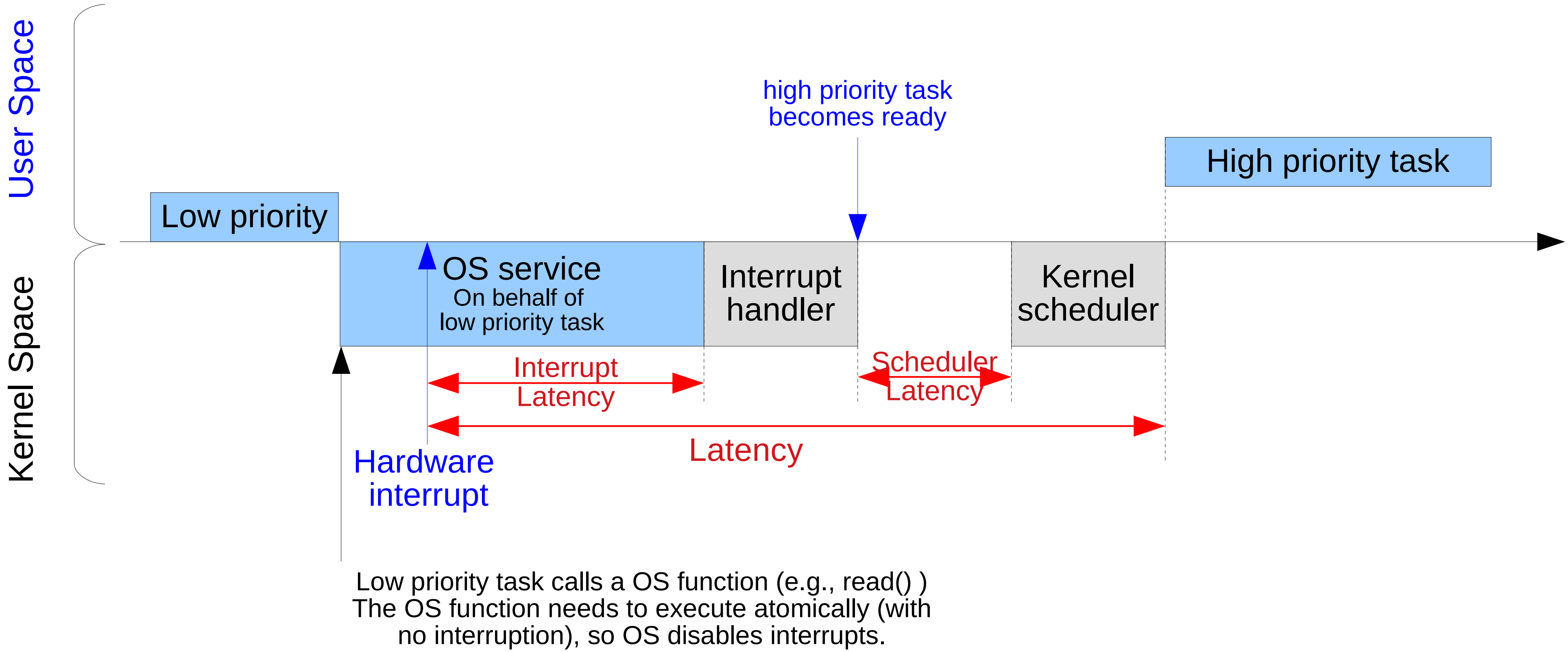
LINUX Kernel → PREEMPT_RT patch

Kernel Latency



LINUX Kernel → PREEMPT_RT patch

Kernel Latency



Standard LINUX:
unbounded maximum interrupt latency

LINUX Kernel → PREEMPT_RT patch

- PREEMPT_RT patch features:
  - New preemption models

    (Reduce interrupt latency)
  - High resolution timers

    (In mainline kernel since 2.6.24-rc1)
  - Threaded Interrupt handlers
  - rt_mutex

    (replaces mutexes used within the kernel)
  - Sleeping spinlocks
  - Preemptible RCU mechanisms

    (RCU – Read-copy update)

LINUX Kernel → PREEMPT_RT patch

Preemption Models:

Notes:
- preemption models affect only the kernel - user space threads are always preemptible.
- preemption model must be selected before compiling the kernel!

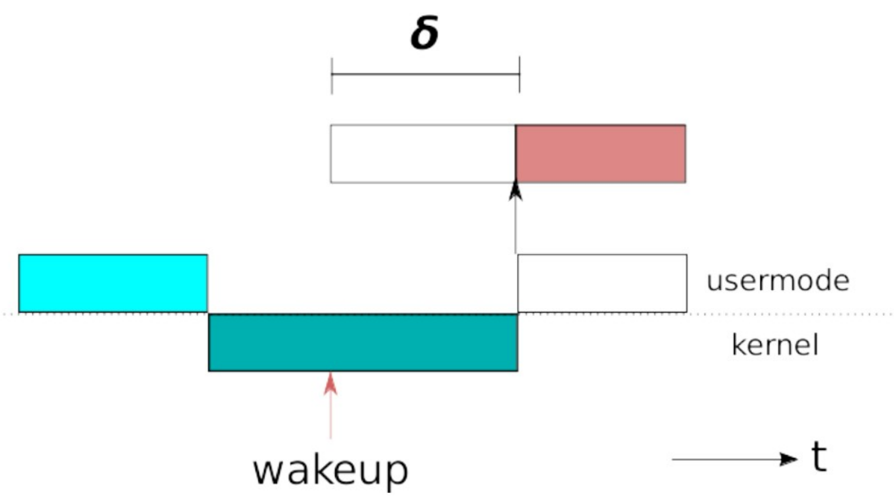
- No Forced Preemption (server):
  - CONFIG_PREEMPT_NONE
  - traditional Linux preemption model, optimized for throughput
    1). System call returns and interrupts are the only preemption points.
  - reduces task switching (=> reduced context switching) to maximize CPU and cache usage .
- Voluntary Kernel Preemption (desktop):
  - CONFIG_PREEMPT_VOLUNTARY
  - Adds more “explicit preemption points” to the kernel code
    2). In addition to explicit preemption points, system call returns and interrupt returns are implicit preemption points.
  - results in: slightly lower throughput, & lower kernel latency

LINUX Kernel → PREEMPT_RT patch

- Preemption Models:
  - Preemptible Kernel (Low-Latency Desktop):
    - all kernel code is preemptible (except for critical sections)
      3). An implicit preemption point is located after each preemption disable section.
  - Preemptible Kernel (Basic RT):
    - 4) Like (3), but adds threaded interrupt handlers
      (equivalent to the kernel command line parameter threadirqs).
    - mainly used for testing and debugging of substitution mechanisms of PREEMPT_RT patch.
  - Fully Preemptible Kernel (RT):
    - Like (4), adds use of sleeping spinlocks and rt_mutex, and large preemption disabled sections are substituted by separate locking constructs.
    - preemption model to use when requiring real-time behavior.

LINUX Kernel → PREEMPT_RT patch

- Preemption Models:



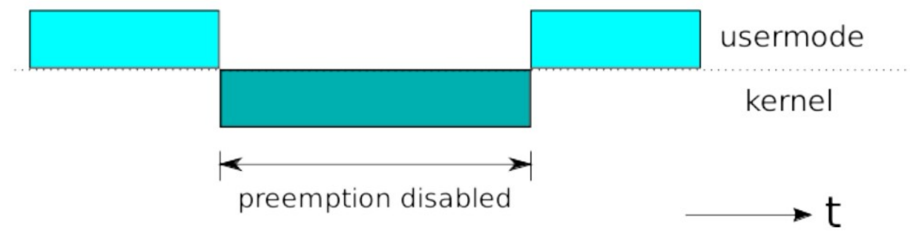
Preemption enabled

Preemption disabled

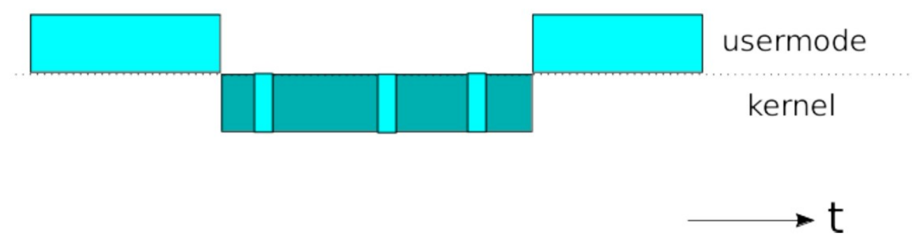
LINUX Kernel → PREEMPT_RT patch

- Preemption Models:

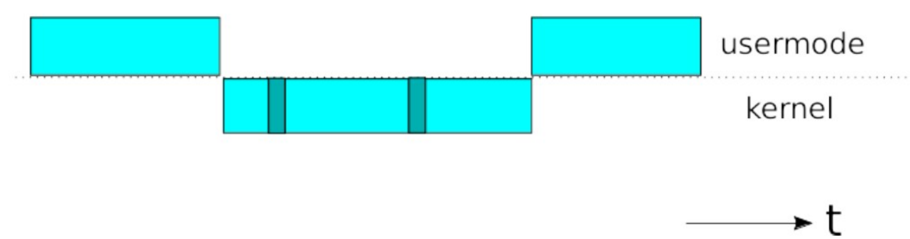
CONFIG_PREEMPT_NONE



CONFIG_PREEMPT_VOLUNTARY



CONFIG_PREEMPT
CONFIG_PREEMPT_RT



Preemption enabled

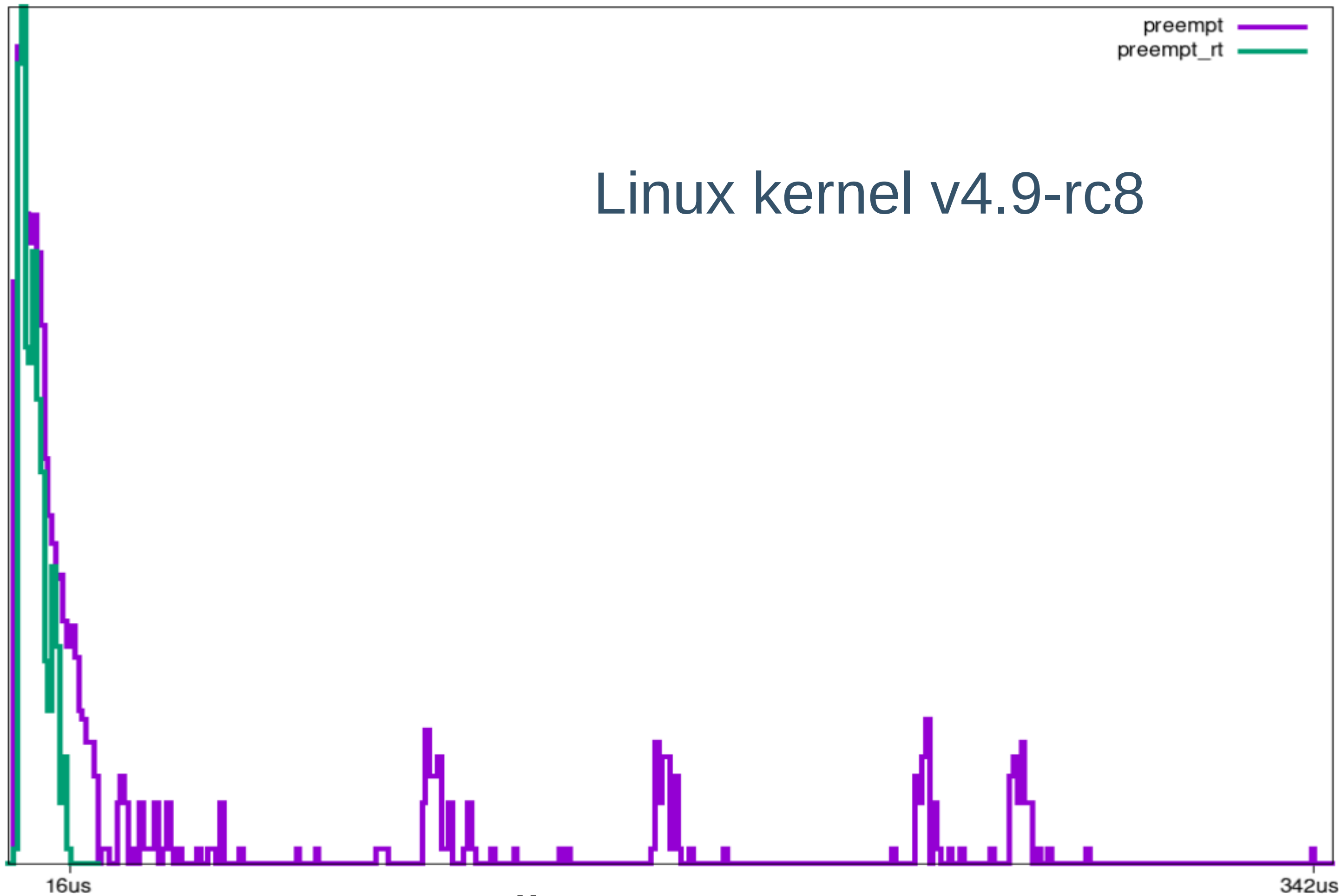
Preemption disabled

LINUX Kernel → PREEMPT_RT patch

- Preemption Models:

Number of iterations

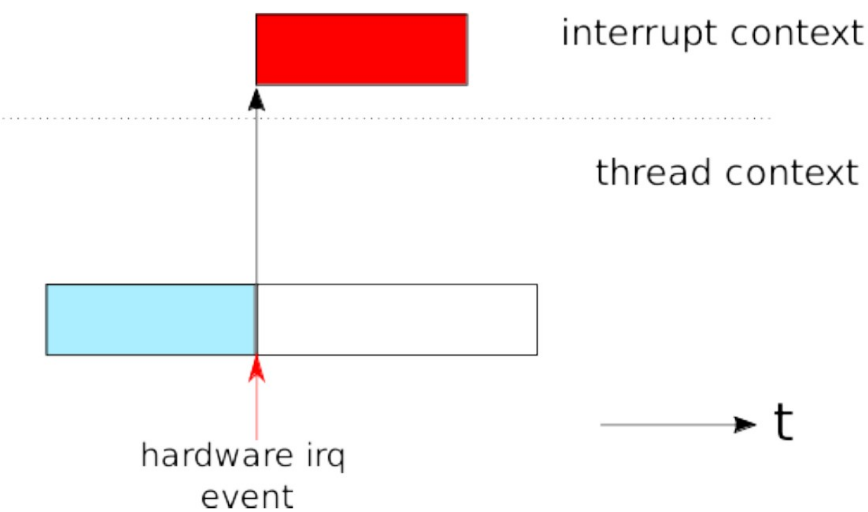
```
While (1) {
 T0 = now();
 Sleep(T);
 T1 = now();
 jitter=(T1-T0)-T;
}
```



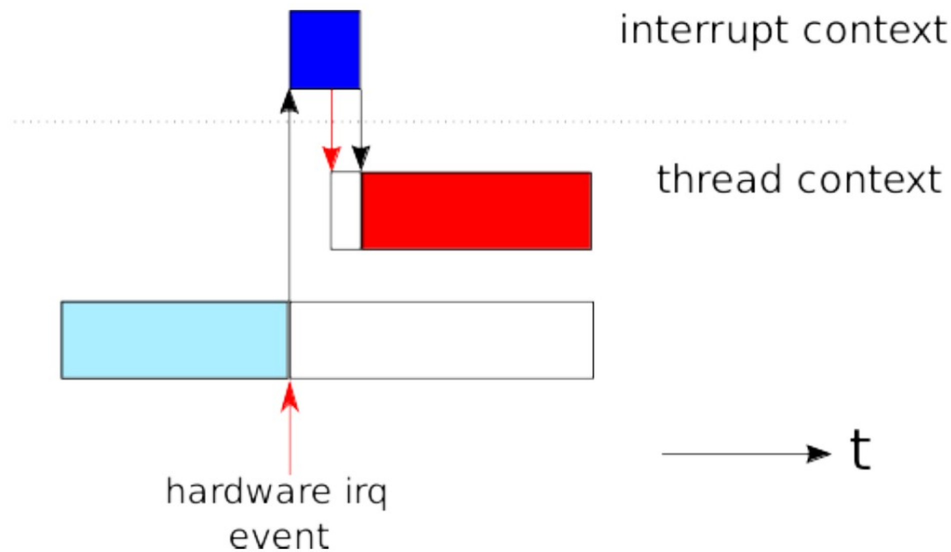
jitter

LINUX Kernel → PREEMPT_RT patch

- Threaded Interrupt Handlers:



Default interrupt handlers

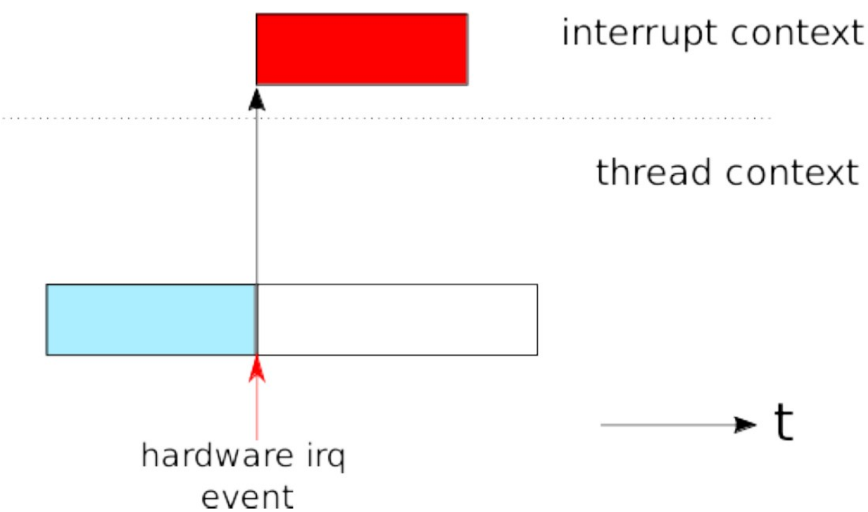


Threaded interrupt handlers

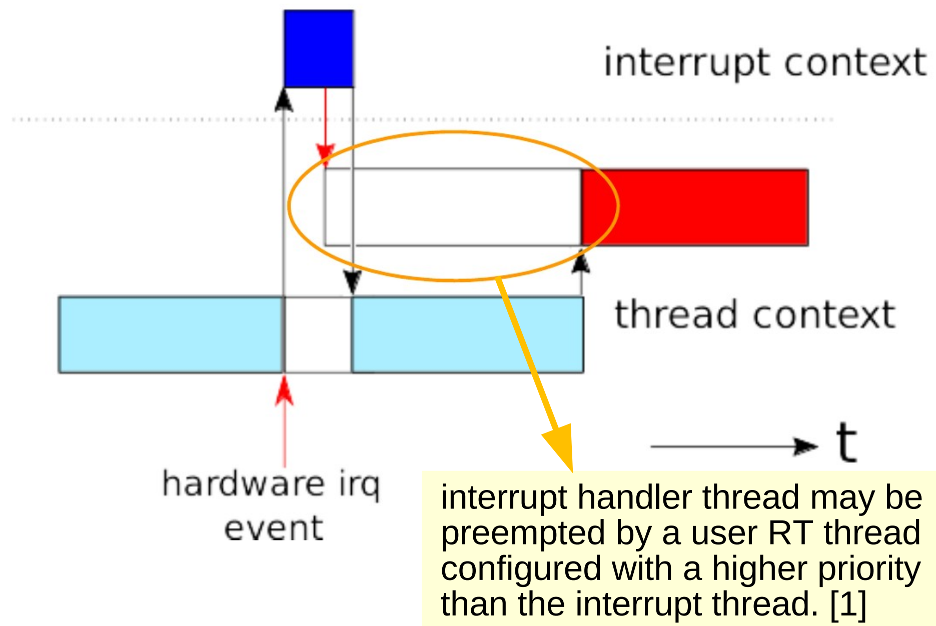
Images: Julia Cartwright,
Senior Software Engineer,
National Instruments, julia.cartwright@ni.com

LINUX Kernel → PREEMPT_RT patch

- Threaded Interrupt Handlers:



Default interrupt handlers



Threaded interrupt handlers

Images: Julia Cartwright,
Senior Software Engineer,
National Instruments, julia.cartwright@ni.com

[1] “Internals of the RT Patch”, Steven Rostedt, Darren V. Hart, OLS 2007]

LINUX Kernel → PREEMPT_RT patch

Threaded Interrupt Handlers:

- In mainline Linux the interrupt service routine is
  - processed in hard interrupt context
  - Processed with hardware interrupts disabled (=> preemption disabled)
  - Interrupt handlers are processed in context of the interrupt service routine with hard interrupts disabled as well.
- kernel command line option \`threadirqs\` => interrupt handlers run as a normal kernel thread.
  - The real interrupt handler (i.e. the interrupt service routine), as executed by the CPU, is only in charge of masking the interrupt and waking-up the corresponding thread
  - thread uses SCHED_FIFO with default priority of 50.
Priorities of different interrupt handlers is configurable.
  - Some interrupts are not threaded (ex. Inter-Processor Interrupts (IPIs)).
(Interrupt handlers set up with the IRQF_TIMER or IRQF_PER_CPU flag are marked as IRQF_NO_THREAD implicitly.
  - PREEMPT_RT patch forces the \`threadirqs\` command line option.

LINUX Kernel → PREEMPT_RT patch

- High resolution timers:
  - resolution of the timers used to be resolution of the system tick (usually 10 ms to 4 ms).
  - Increasing the regular system tick frequency => consume too much resources
  - high-resolution timers uses hardware timers to program interrupts at the right moment.
  - Hardware timers are multiplexed => a single hardware timer can handle a large number of software-programmed timers.
  - Usable directly from user-space using the usual timer APIs

  NOTE: “the delivery of signals at the expiry of itimer and posix interval timers must be done in thread context of softirq threads. To avoid long latencies the softirq threads have been separated in the realtime preemption patch a while ago. A problem remains: the hrtimers softirq thread can be arbitrarily delayed by higher priority tasks. A workaround is to increase the priority of the hrtimer softirq thread, but this has the effect that all timer related signals are delivered at high priority and therefore introduce latency impacts to high priority tasks.
  Note that (clock_)nanosleep functions do not suffer from this problem as the wakeup function at timer expiry is executed in the context of the high resolution timer interrupt.”

LINUX Kernel → PREEMPT_RT patch

- High resolution timers:
  - Dynamic ticks:
    - Disables ticks when idle process is running, or when only one thread is on the CPU's ready queue.
    - Allows the CPU to enter sleep state for longer continuous periods

LINUX Kernel → PREEMPT_RT patch

- rt mutexes:
  - All mutexes in the Linux kernel are replaced by rt_mutexes.
  - rt_mutex implements priority inheritance to avoid priority inversion.
    This also applies to sleeping spinlocks and rwlocks.
  - Remember: In userspace, priority inheritance must be explictly enabled on a per-mutex basis.

- Other sources of latency (hardware):
  - Cache synchronization delays
  - Bus delays
  - NUMA: non-uniform memory access (Multi-core architectures)
  - Unpredictable IO events (Ethernet, WIFI)
  - Temperature changes → CPU throttling
  - Etc....