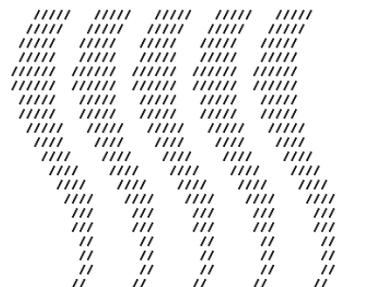




FYS 4220 / 9220 – 2011 / #7

Real Time and Embedded Data Systems and Computing

Scheduling of Real- Time processes, strategies and analysis

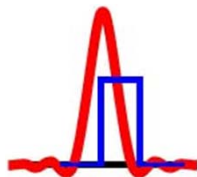


T O R N A D O
Development System
Host Based Shell
Version 2.0.2

Copyright 1995-1999 Wind River Systems, Inc.



PowerMIDAS M5000 (VME)





Some definitions to start with

- The **scheduler** is the component of the **kernel** that selects which process to run next. The scheduler (or process scheduler, as it is sometimes called) can be viewed as the code that divides the finite resource of processor time between the runnable processes on a system.
- Multitasking operating systems come in two flavors: **cooperative multitasking** and **preemptive multitasking**. Linux, like all Unix variants and most modern operating systems, provides **preemptive multitasking**. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to resume running. The act of involuntarily suspending a running process is called **preemption**. The time a process runs before it is preempted is predetermined, and is called the **timeslice** of the process. The timeslice, in effect, gives each process a slice of the processor's time. Managing the timeslice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the system
 - Conversely, in cooperative multitasking, a process does not stop running until it voluntarily decides to do so. This approach has many shortcomings



The challenge of scheduling in Real-Time systems

- Scheduling is the problem of assigning a set of processes (tasks) to a set of resources subject to a set of constraints. Examples of scheduling constraints for Real-Time processes include **deadlines** (e.g., job i must be completed by a time T), **resource capacities** (e.g., limited memory space), **precedence constraints** on the order of tasks (e.g., sequencing of cooperating tasks according to their activities), and **priorities** on tasks (e.g., finish job P as soon as possible while meeting the other deadlines).
 - There are zillions (well, may be a few less than that) of computer science papers on scheduling



Pick your own scheduling strategy ...

- The following is a list of common scheduling practices and disciplines (ref Wikipedia):
 - Borrowed-Virtual-Time Scheduling (BVT)
 - Completely Fair Scheduler (CFS)
 - Critical Path Method of Scheduling
 - Deadline-monotonic scheduling (DMS)
 - Deficit round robin (DRR)
 - Earliest deadline first scheduling (EDF)
 - Elastic Round Robin
 - Fair-share scheduling
 - First In, First Out (FIFO), also known as First Come First Served (FCFS)
 - Gang scheduling
 - Genetic Anticipatory
 - Highest response ratio next (HRRN)
 - Interval scheduling
 - Last In, First Out (LIFO)
 - Job Shop Scheduling
 - Least-connection scheduling
 - Least slack time scheduling (LST)
 - List scheduling
 - Lottery Scheduling
 - Multilevel queue
 - Multilevel Feedback Queue
 - Never queue scheduling
 - $O(1)$ scheduler
 - Proportional Share Scheduling
 - Rate-monotonic scheduling (RMS)
 - Round-robin scheduling (RR)
 - Shortest expected delay scheduling
 - Shortest job next (SJN)
 - Shortest remaining time (SRT)
 - Staircase Deadline scheduler (SD)
 - "Take" Scheduling
 - Two-level scheduling
 - Weighted fair queuing (WFQ)
 - Weighted least-connection scheduling
 - Weighted round robin (WRR)
 - Group Ratio Round-Robin: $O(1)$

**We will discuss a
few of these in more
details**



Scheduling of RT processes

- An absolute requirement in hard Real-Time systems is that deadlines are met! Missing a deadline may have catastrophic consequences. This requirement is the baseline for true Real-Time Operating Systems!
 - Note that a RTOS should have guaranteed worst case reaction times. However, this is obviously processor dependent.
- In soft Real-Time systems deadlines may however occasionally be missed without leading to catastrophe. For such applications standard OS's can be used (Windows, Linux, etc)
 - Since Linux is popular in many embedded systems, let us have a quick look at some of its characteristics

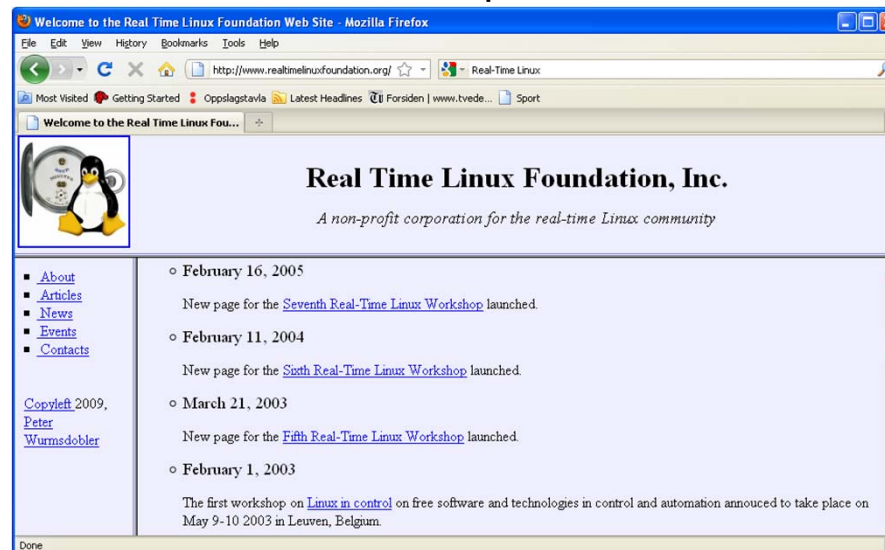


Scheduling in Linux and Real-Time

- A web search on Linux scheduling will give numerous hits, not least on hacker-type and/or Real-Time improvements of the scheduler
 - The following pages on Linux 2.6.8.1 have been copied from http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf . This web reference from 2007 is however no longer available and the information may not be completely up-to-date.
 - A number of so-called Real-Time extensions to Linux have been made over the years, some of them on commercial basis. Many of them have however been made obsolete due to a continuous development of Linux.

Real-Time Linux Foundation
web (active link)

See also Wind River
<http://www.rtlinuxfree.com>





Linux 2.6.8.1 1/2

Linux 2.6.8.1 Scheduler – static priorities:

All tasks has a static priority, the *nice* value. The *nice* values range from -20 to 19, the higher value being lower priority (nicer).

4.8 Soft Real-Time Scheduling

The Linux scheduler supports soft *real-time (RT)* scheduling. This means that it can effectively schedule tasks that have strict timing requirements. However, while the Linux 2.6.x kernel is usually capable of meeting very strict RT scheduling deadlines, it does not guarantee that deadlines will be met. RT tasks are assigned special scheduling modes and the scheduler gives them priority over any other task on the system. RT scheduling modes include a first-in-first-out (FIFO) mode which allows RT tasks to run to completion on a first-come-first-serve basis, and a round-robin scheduling mode that schedules RT tasks in a round-robin fashion while essentially ignoring non-RT tasks on the system.



Linux 2.6.8.1 2/2

5.8 Soft RT Scheduling

The Linux 2.6.8.1 scheduler provides soft RT scheduling support. The “soft” adjective comes from the fact that while it does a good job of meeting scheduling deadlines, it does not guarantee that deadlines will be met.

5.8.1 Prioritizing Real-Time Tasks

RT tasks have priorities from 0 to 99 while non-RT task priorities map onto the internal priority range 100-140. Because RT tasks have lower priorities than non-RT tasks, they will always preempt non-RT tasks. As long as RT tasks are runnable, no other tasks can run because RT tasks operate with different scheduling schemes, namely SCHED_FIFO and SCHED_RR. Non-RT tasks are marked SCHED_NORMAL, which is the default scheduling behavior.

5.8.2 SCHED_FIFO Scheduling

SCHED_FIFO tasks schedule in a first-in-first-out manner. If there is a SCHED_FIFO task on a system it will preempt any other tasks and run for as long as it wants to. SCHED_FIFO tasks do not have timeslices. Multiple SCHED_FIFO tasks are scheduled by priority - higher priority SCHED_FIFO tasks will preempt lower priority SCHED_FIFO tasks.

5.8.3 SCHED_RR Scheduling

SCHED_RR tasks are very similar to SCHED_FIFO tasks, except that they have timeslices and are always preempted by SCHED_FIFO tasks. SCHED_RR tasks are scheduled by priority, and within a certain priority they are scheduled in a round-robin fashion. Each SCHED_RR task within a certain priority runs for its allotted timeslice, and then returns to the bottom of the list in its priority array queue.



The Linux "Completely Fair Scheduler"

- The **Completely Fair Scheduler (CFS)** was merged into the 2.6.23 release of the Linux kernel. It aims to maximize overall CPU utilization while maximizing interactive performance. It was written by Ingo Molnar.
http://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- In contrast to the scheduler used in older Linux 2.6 kernels, the CFS scheduler implementation is not based on run queues.
 - Instead a *red-black tree*, a type of selfbalancing binary search (http://en.wikipedia.org/wiki/Red-black_tree) implements a 'timeline' of future task execution. Additionally, the scheduler uses nanosecond granularity accounting, the atomic units by which an individual process' share of the CPU was allocated (thus making redundant the previous notion of timeslices). This precise knowledge also means that no specific heuristics (read "rule of thumb") are required to determine the interactivity of a process, for example.
- The Linux **Brain Fuck Scheduler (BFS)** was presented in 2009 as an alternative to CFS. http://en.wikipedia.org/wiki/Brain_Fuck_Scheduler
- OK, so much about LINUX, over to the VxWorks RTOS



VxWorks multitasking

- *Multitasking* provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events. The VxWorks real-time kernel, *wind*, provides the basic multitasking environment. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. Each apparently independent program is called a *task*. Each task has its own *context*, which is the CPU environment and system resources that the task sees each time it is scheduled to run by the kernel. On a context switch, a task's context is saved in the task control block (TCB)
 - A TCB can be accessed through the ***taskTcb(task_ID)*** routine. However, do not directly modify a TCB!



VxWorks multitasking

- Each task has its own *context*, which is the CPU environment and system resources that the task sees each time it is scheduled to run by the kernel. On a context switch, a task's context is saved in the task control block (TCB). A task's context includes:
 - a thread of execution; that is, the task's program counter
 - the tasks' virtual memory context (if process support is included)
 - the CPU registers and (optionally) coprocessor registers
 - stacks for dynamic variables and function calls
 - I/O assignments for standard input, output, and error
 - a delay timer
 - a time-slice timer
 - kernel control structures
 - signal handlers
 - task variables
 - error status (errno)
 - debugging and performance monitoring values
- Note that in conformance with the POSIX standard, all tasks in a process share the same environment variables (unlike kernel tasks, which each have their own set of environment variables).
- A VxWorks task will be in one of states listed on next page



VxWorks Wind state transitions

Task State Transition

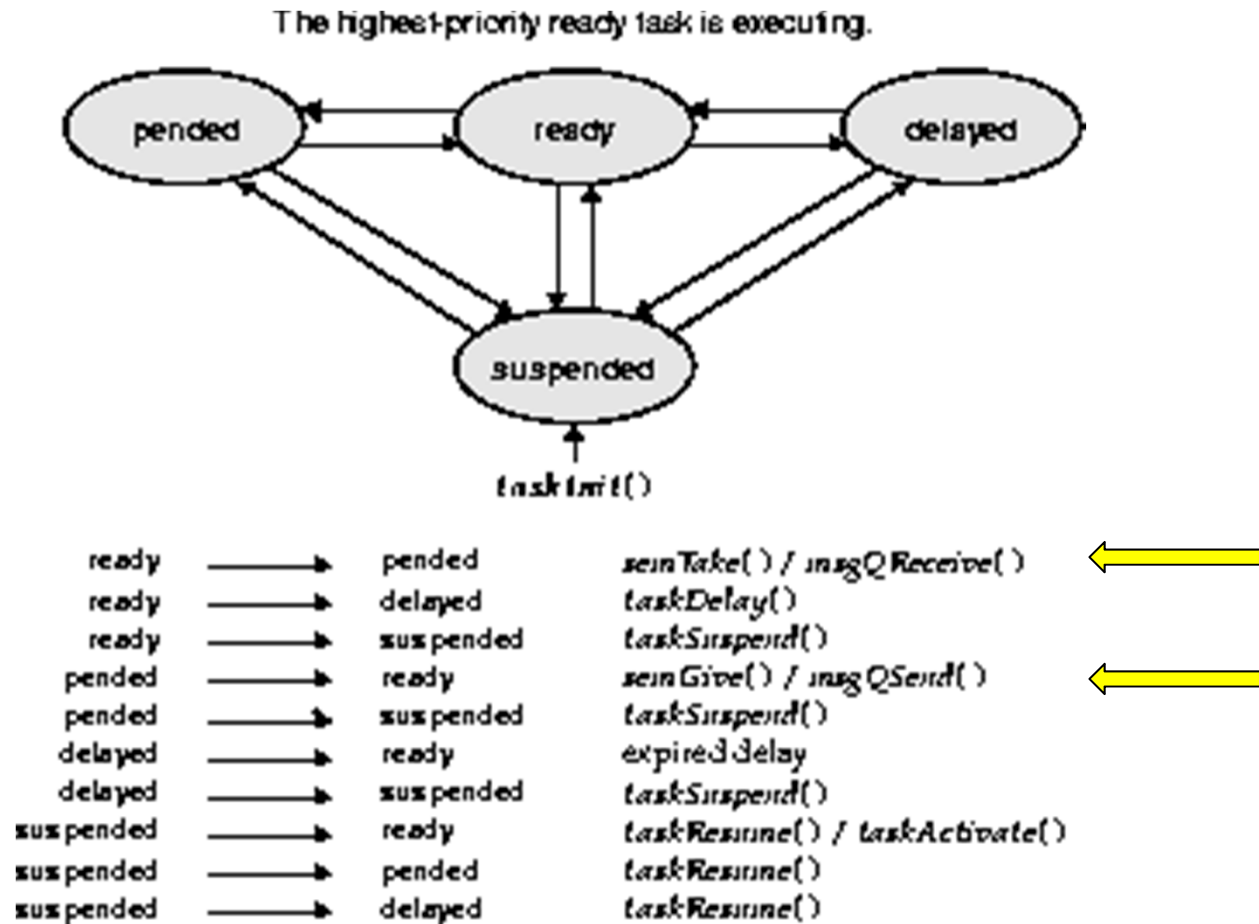
The kernel maintains the current state of each task in the system. A task changes from one state to another as the result of kernel function calls made by the application. When created, tasks enter the *suspended* state. Activation is necessary for a created task to enter the *ready* state. The activation phase is extremely fast, enabling applications to pre-create tasks and activate them in a timely manner. An alternative is the *spawning* primitive, which allows a task to be created and activated with a single function. Tasks can be deleted from any state.

The *wind* kernel states are shown in the state transition diagram in [Figure 2-1](#), and a summary of the corresponding *state symbols* you will see when working with Tornado development tools is shown in [Table 2-1](#).

Table 2-1: Task State Transitions

State Symbol	Description
READY	The state of a task that is not waiting for any resource other than the CPU.
PEND	The state of a task that is blocked due to the unavailability of some resource.
DELAY	The state of a task that is asleep for some duration.
SUSPEND	The state of a task that is unavailable for execution. This state is used primarily for debugging. Suspension does not inhibit state transition, only task execution. Thus <i>pended-suspended</i> tasks can still unblock and <i>delayed-suspended</i> tasks can still awaken.
DELAY + S	The state of a task that is both delayed and suspended.
PEND + S	The state of a task that is both pendend and suspended.
PEND + T	The state of a task that is pendend with a timeout value.
PEND + S + T	The state of a task that is both pendend with a timeout value and suspended.
<i>state</i> + I	The state of task specified by <i>state</i> , plus an inherited priority.

Wind task state diagram and task transitions



See **taskLib** for the **task---**() routines. Any system call resulting in a transition may affect scheduling!



VxWorks task scheduling

- **The default algorithm is priority based pre-emptive**
 - With a **preemptive priority-based scheduler**, each task has a priority and the kernel ensures that the CPU is allocated to the highest priority task that is ready to run. This scheduling method is *preemptive* in that if a task that has higher priority than the current task becomes ready to run, the kernel immediately saves the current task's context and switches to the context of the higher priority task
 - A **round-robin scheduling algorithm** attempts to share the CPU fairly among all ready tasks of the *same priority*.
 - Note: VxWorks provides the following kernel scheduler facilities:
 - The VxWorks native scheduler, which provides options for preemptive priority-based scheduling or round-robin scheduling.
 - A POSIX thread scheduler that provides POSIX thread scheduling support in user-space (processes) while keeping the VxWorks task scheduling.
 - A kernel scheduler replacement framework that allows users to implement customized schedulers. See documentation for more information

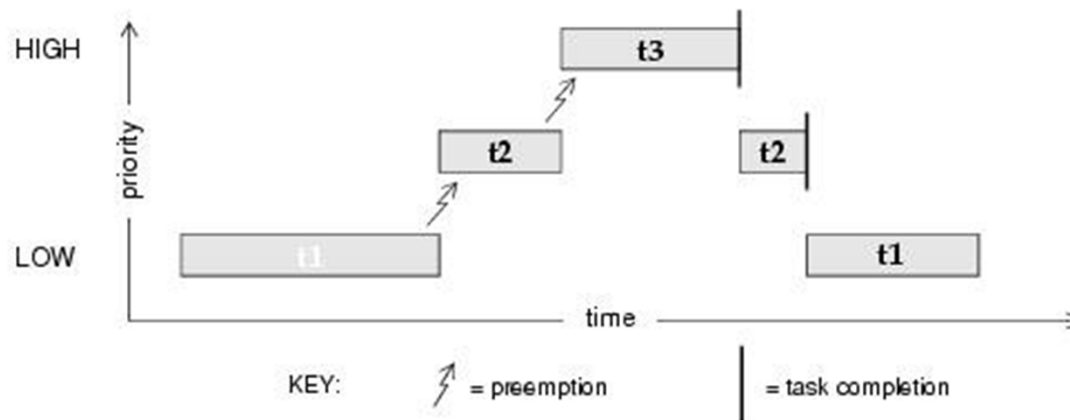


VxWorks task scheduling (cont)

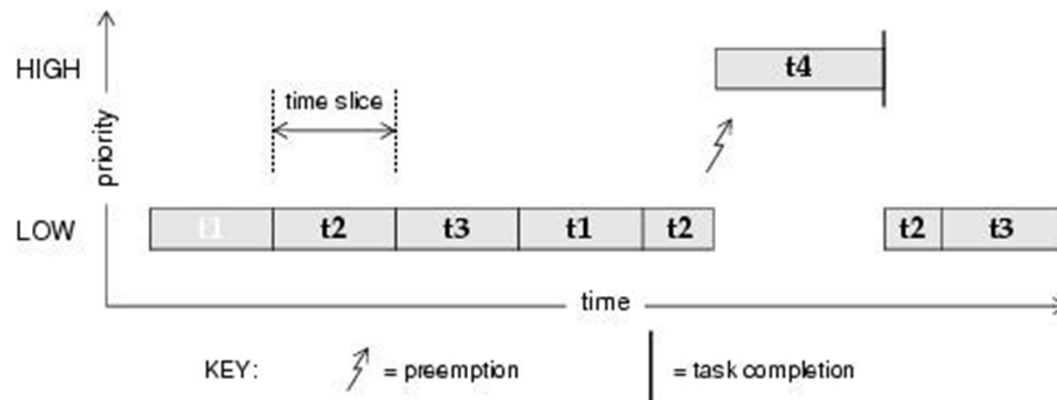
- The kernel has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.
 - All application tasks should be in the priority range from 100 to 255.
 - Tasks are assigned a priority when created. One can also change a task's priority level while it is executing by calling **taskPrioritySet()**. The ability to change task priorities dynamically allows applications to track precedence changes in the real world.
- Round-robin
 - A round-robin scheduling algorithm attempts to share the CPU fairly among all ready tasks of the *same priority*. Round-robin scheduling uses *time slicing* to achieve fair allocation of the CPU to all tasks with the same priority. Each task, in a group of tasks with the same priority, executes for a defined interval or *time slice*.
 - It may be useful to use round-robin scheduling in systems that execute the same application in more than one process. In this case, multiple tasks would be executing the same code, and it is possible that a task might not relinquish the CPU to a task of the same priority running in another process (running the same binary). Note that round-robin scheduling is global, and controls all tasks in the system (kernel and processes); it is not possible to implement round-robin scheduling for selected processes. Round-robin scheduling is enabled by calling **kernelTimeSlice(ticks)**, which takes a parameter for a time slice, or interval. If ticks = 0 round-robin is disabled. This interval is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task. Thus, the tasks rotate, each executing for an equal interval of time. No task gets a second slice of time before all other tasks in the priority group have been allowed to run.

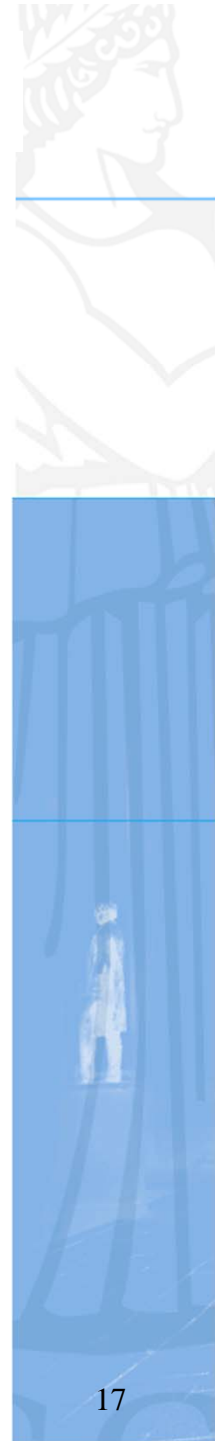
VxWorks task scheduling (cont)

- Priority-bases preemptive
 - VxWorks supply routines for pre-emption locks which prevent context switches .



- Round-Robin





What follows are presentation on scheduling from the book by Burns & Wellings , plus some add-ons.

The goal is to demonstrate numerical methods for analysis of scheduling behaviour

Scheduling



- In general, a scheduling scheme provides two features:
 - An algorithm for ordering the use of system resources (in particular the CPUs)
 - A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied
 - It is a good philosophy to assume worst-case behaviour. The problem is to identify it.
- The prediction can then be used to confirm the temporal requirements of the application

Simple Process Model



- The application is assumed to consist of a fixed set of processes
- All processes are periodic, with known periods
- The processes are completely independent of each other
- All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
- All processes have a deadline equal to their period (that is, each process must complete before it is next released)
- All processes have a fixed worst-case execution time

Notation

- B** Worst-case blocking time for the process (if applicable)
- C** Worst-case computation time (WCET) of the process
- D** Deadline of the process (see also **R**)
- I** The interference time of the process
- J** Release jitter of the process
- N** Number of processes in the system
- P** Priority assigned to the process (if applicable)
- R** Worst-case response time of the process. *Stands for when process terminates, that is, when the job is done!!*
- T** Minimum time between process releases (process period)
- U** The utilization of each process (equal to C/T)
- a-z** The name of a process

Process-Based Scheduling



- Priority scheduling approaches discussed here:
 - Fixed-Priority Scheduling (FPS)
 - Earliest Deadline First (EDF)
 - Value-Based Scheduling (VBS)

Fixed-Priority Scheduling (FPS)

- This is the most widely used approach and is the main focus of this course
- Each process has a fixed, static, priority which is computer pre-run-time
- The runnable processes are executed in the order determined by their priority
- In Real-Time systems, the “priority” of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity. (Well, there is some interdependency here!)

Earliest Deadline First (EDF) Scheduling

- The runnable processes are executed in the order determined by the absolute deadlines of the processes
- The next process to run being the one with the shortest (nearest) deadline
- Although it is usual to know the relative deadlines of each process (e.g. 25ms after release, i.e. startup), the absolute deadlines are computed at run time and hence the scheme is described as dynamic

Value-Based Scheduling (VBS)

- If a system can become overloaded then the use of simple static priorities or deadlines is not sufficient; a more **adaptive** scheme is needed
- This often takes the form of assigning a **value** to each process and employing an on-line **value-based scheduling algorithm** to decide which process to run next

Preemption and Non-preemption

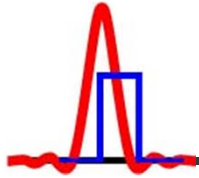
- With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one
- In a **preemptive** scheme, there will be an immediate switch to the higher-priority process
- With **non-preemption**, the lower-priority process will be allowed to complete before the other executes
- Preemptive schemes enable higher-priority processes to be more reactive, and hence they are preferred
- Alternative strategies allow a lower priority process to continue to execute for a bounded time
- These schemes are known as **deferred preemption** or **cooperative dispatching**
- Schemes such as EDF and VBS can also take on a preemptive or non preemptive form

FPS and Rate Monotonic Priority Assignment

- Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
- I.e, for two processes i and j ,

$$T_i < T_j \Rightarrow P_i > P_j$$

- This assignment is optimal in the sense that if any process set can be scheduled (using pre-emptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme
- *Note, priority 1 is here the lowest (least) priority. For VxWorks Wind priorities the scale is inverted!*



Rate-monotonic scheduling

- Theorem:
 - Given a set of periodic tasks and preemptive priority scheduling, then assigning priorities such that the tasks with shorter periods have higher priorities (rate-monotonic), yields an optimal scheduling algorithm.
 - Liu, C.L and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", journal of the ACM, Vol. 20, No. 1, 1973

Example Priority Assignment

Process	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Note, only relative priority values are relevant

Utilisation-Based Analysis

- For D=T task sets only
- A simple, sufficient but not necessary, schedulability test exists:

$$U \equiv \sum_{i=1}^N \frac{C_i}{T_i} \leq N (2^{1/N} - 1)$$

Liu and Layland

$$U \leq 0.69 \text{ as } N \rightarrow \infty$$

Utilization Bounds



N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

Approaches 69.3% asymptotically

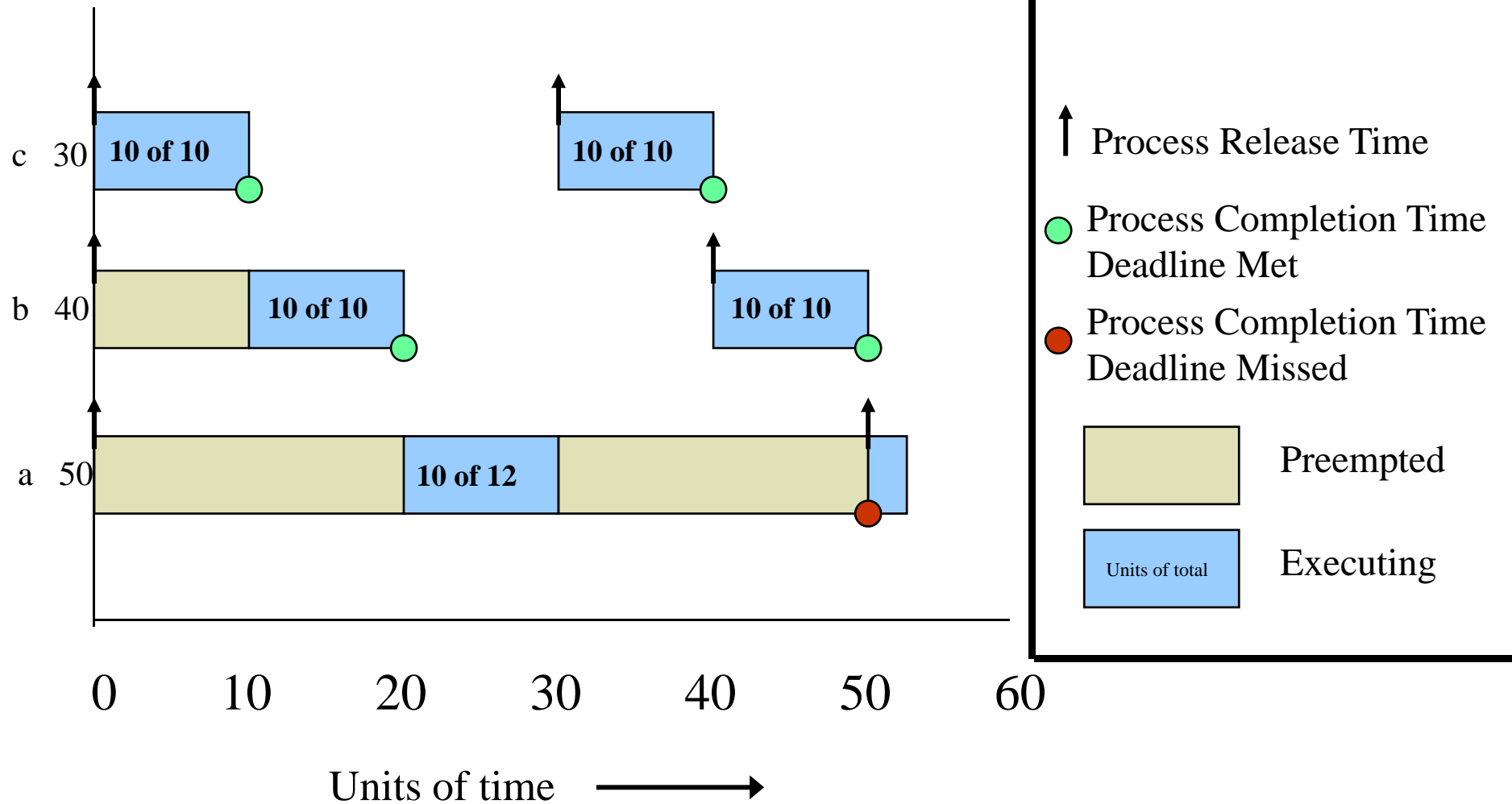
Process Set A

Process	Period T	ComputationTime C	Priority P	Utilization U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three processes (0.78) and, hence, this process set fails the utilization test

Time-line for Process Set A

process - T



Note! Process "c" has the highest priority

Process Set B

Process	Period T	ComputationTime C	Priority P	Utilization U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

- The combined utilization is 0.775
- This is below the threshold for three processes (0.78) and, hence, this process set will meet all its deadlines

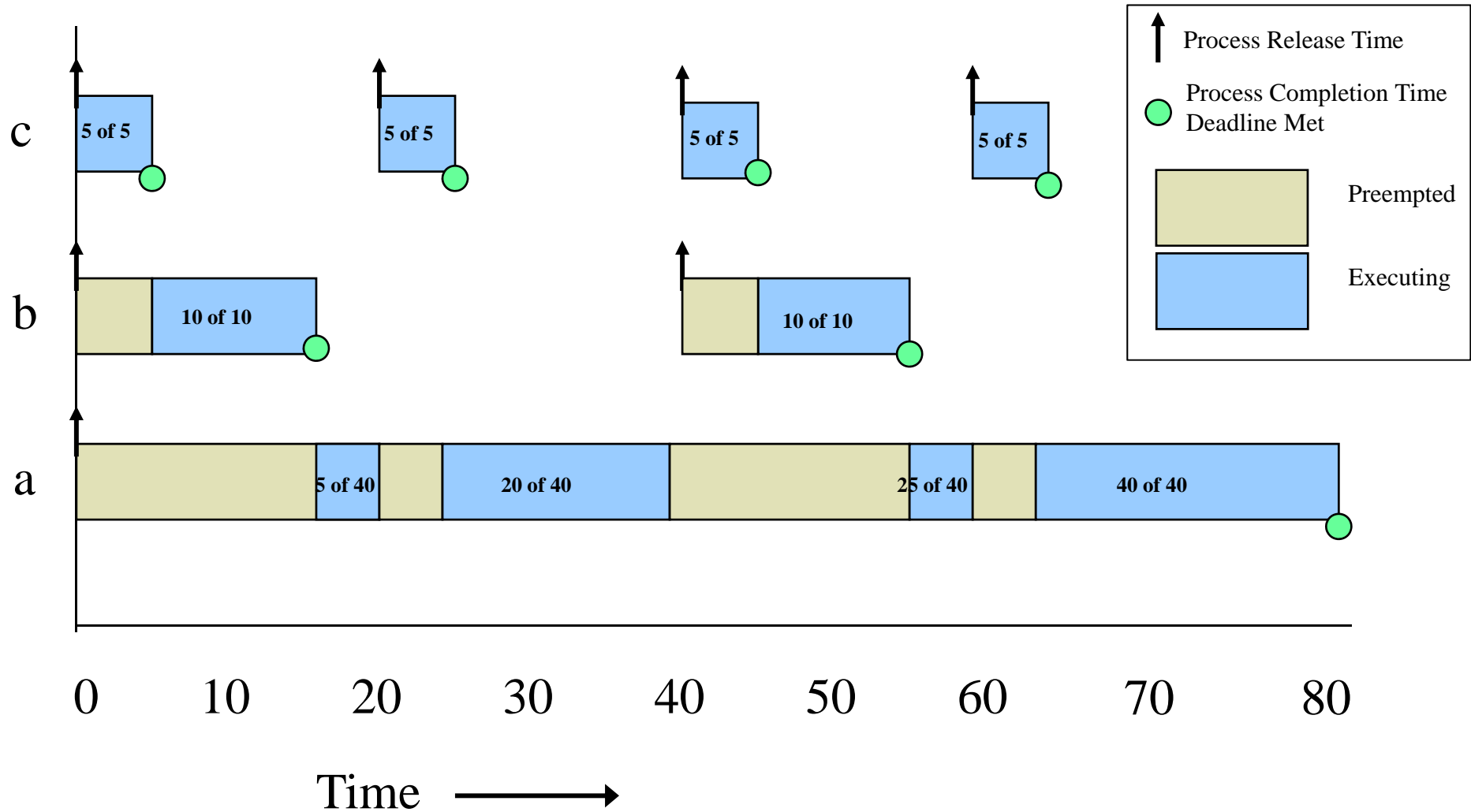
Process Set C

Process	Period T	ComputationTime C	Priority P	Utilization U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

- The combined utilization is 1.0
- This is above the threshold for three processes (0.78) but the process set will meet all its deadlines. (Just a lucky combination of C and T!)
 - Well, the overhead (context switching, clock interrupt handling etc) has been assumed to be zero, which is not true!!

Time-line for Process Set C

Process



Note! Process "c" has the highest priority

Utilization-based Test for EDF

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq 1 \quad \text{A much simpler test}$$

- Superior to FPS; it can support high utilizations. However
- FPS is easier to implement as priorities are static
- EDF is dynamic and requires a more complex run-time system which will have higher overhead
- It is easier to incorporate processes without deadlines into FPS; giving a process an arbitrary deadline is more artificial
- It is easier to incorporate other factors into the notion of priority than it is into the notion of deadline
- During overload situations
 - FPS is more predictable; Low priority process miss their deadlines first
 - EDF is unpredictable; a domino effect can occur in which a large number of processes miss deadlines

Response-Time Analysis

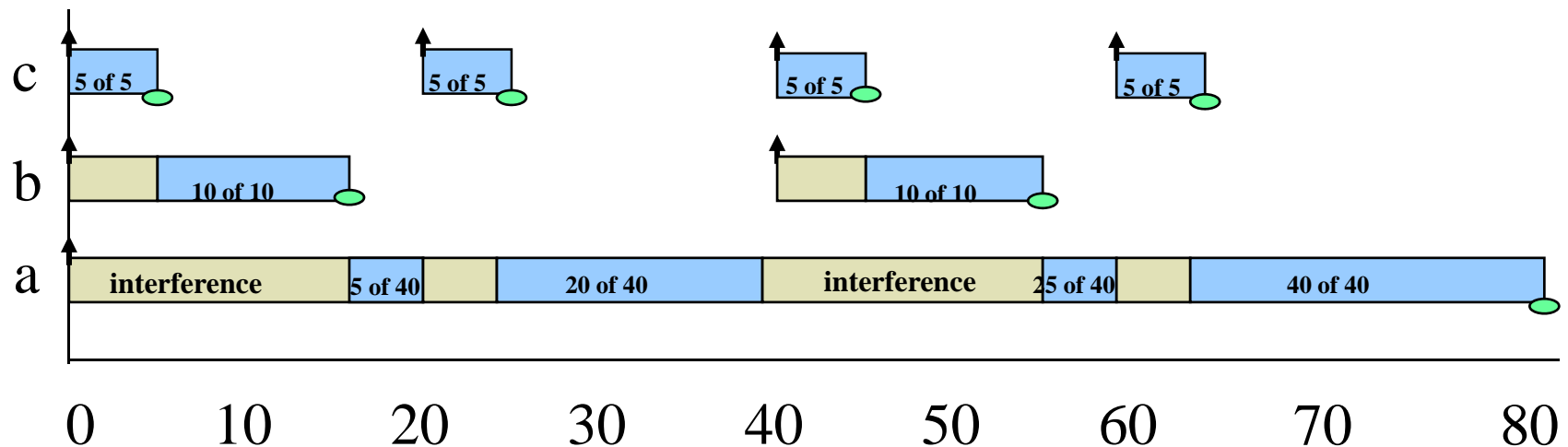
For a detailed presentation, see file [Scheduling_BW_ResponseTimeAnalysis.pdf](#)

- Here task i 's worst-case response time, R , is calculated first and then checked (trivially) with its deadline

$$R \leq D$$

$$R_i = C_i + I_i$$

where I is the interference (CPU utilization) from higher priority tasks



Note! Process "c" has the highest priority

Calculating R

During R , each higher priority task j will execute a number of times:

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

The ceiling function $\lceil \cdot \rceil$ gives the smallest integer greater than the fractional number on which it acts. So the ceiling of $1/3$ is 1, of $6/5$ is 2, and of $6/3$ is 2.

Total interference is given by:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Response Time Equation

$$R_i = C_i + \sum_{j \in hp(i)} \left[\frac{R_i}{T_j} \right] C_j$$

Where $hp(i)$ is the set of tasks with priority higher than task i

Can be solved by forming a recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left[\frac{w_i^n}{T_j} \right] C_j$$

The set of values $w_i^0, w_i^1, w_i^2, \dots, w_i^n, \dots$ is monotonically non decreasing
When $w_i^n = w_i^{n+1}$ the solution to the equation has been found, w_i^0
must not be greater than R_i (e.g. 0 or C_i)

Response Time Algorithm

for i **in** $1..N$ **loop** -- for each process in turn

$n := 0$

loop $w_i^n := C_i$

calculate new w_i^{n+1}

if $w_i^{n+1} = w_i^n$ **then**

$R_i = w_i^n$
exit value found

end if

if $w_i^{n+1} > T_i$ **then**
exit value not found

end if

$n := n + 1$

end loop

end loop

```

/* Fixed Priority Analysis, ref B & W */
/* The process parameters are organized from highest
   |to lowest priorities, i.e. from shortest periods */
int FPS()
{
  int T[3] = {T1,T2,T3};           /* defined elsewhere */
  int C[3] = {C1,C2,C3};           /* defined elsewhere */
  int R[3], W[3][10];
  int i, j, n, k, q, N = 3, converge = 1;

  /* loop over the processes, cf. algorithm in 13.5 */
  for (i = 0; i < N; i++)
  {
    n = 0;
    W[i][n] = C[i];

    /* max 10 iterations, cf. array W */
    for (k = 0; k < 10; k++)
    {
      W[i][n+1] = C[i];

      /* summa over higher priority processes, (13.5) */
      for (j = i-1; j >= 0; j--)
      {
        q = W[i][n]/T[j];
        if (W[i][n] > q*T[j]) q = q+1;
        W[i][n+1] = W[i][n+1] + q*C[j];
      }
    }
    #ifdef DEBUG
    printf("- i,n,Wn,Wn+1 = %d %d %d %d\n",
           i,n,W[i][n],W[i][n+1]);
    #endif

    /* test for convergence */
    if (W[i][n+1] == W[i][n])
    {
      R[i] = W[i][n];
      printf("- converged R[%d] = %d\n", i, R[i]);
      break;
    };

    if (W[i][n+1] > T[i])
    {
      printf("- R[i] > T[i] for i = %d, abort\n", i);
      converge = 0;
      break;
    }
    n++;
  }
}
if (converge == 0)
{
  printf("- response analysis did not converge\n");
  return (ERROR);
}

/* compare worst-case response with period */
for (i = 0; i < N; i++)
{
  printf("- proc %d, T = %3d, worst-case response %3d",
         (i+1), T[i], R[i]);
  if (R[i] <= T[i]) printf(" OK\n");
  else               printf(" FAILURE\n");
}
return (OK);
}

```

Implemented in the FYS4220 VxWorks program *FPS_analysis.c*

Process Set D

Process	Period T	ComputationTime C	Priority P
a	7	3	3
b	12	3	2
c	20	5	1

$$w_b^0 = 3$$

$$R_a = 3 \quad w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$R_b = 6$$

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$$R_c = 20$$



Demo: Response Time Analysis `FPS_analysis.c` coded for VxWorks

- Running the program on process set D
 - Code not shown here, only printouts

The image shows two overlapping windows from a VxWorks simulation. The top window, titled "VxWorks Simulator for Windows", displays the following text:

```
-> Current scheduling is FIFO (Preemptive Priority)
CLOCK_REALTIME resolution = 16666666 ns
Rate Monotonic Scheduling
WIND prioritizes for P1 P2 P3 = 17 22 30
Calibrating Busy loop, takes some time ....
CalBusy: 30 iterations to TickCalScale = 3864096
CalBusy: CPU ticks set, measured 20 19

137: P1 exit
233: P2 exit
389: P3 exit
```

The bottom window, titled "Tornado - [Shell vxsim@FYSPC-ELG045]", shows the command prompt and the following output:

```
-> result
Response time simulation for configuration:
Periods      T1 T2 T3 = 7 12 20
Comp. time C1 C2 C3 = 3 3 5
```

At the bottom of the Tornado window, the text "Wind River Systems" and "NUM" are visible.



Response Time Analysis - VxWorks

- P1 – P2 – P3 scheduling according to set D: all deadlines met

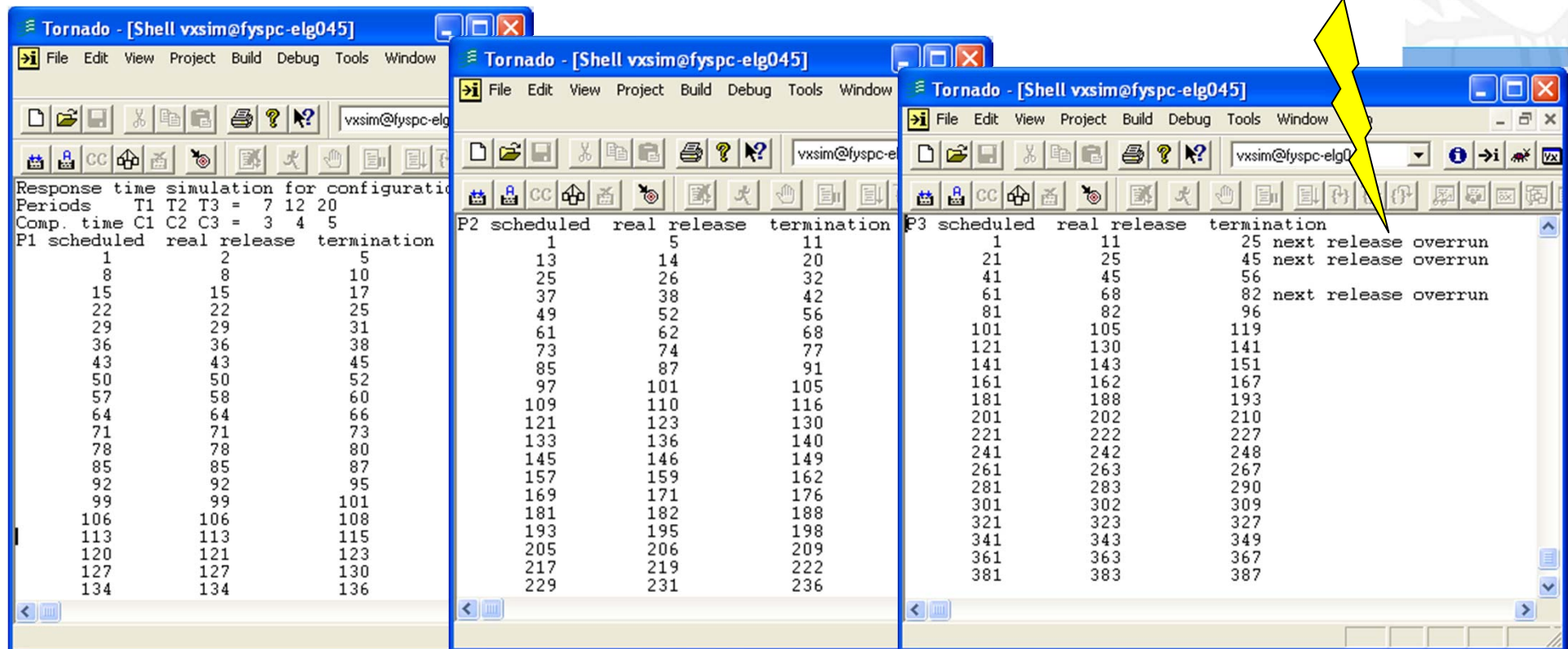
P1 scheduled real release termination			
1	2	4	
8	8	11	
15	15	17	
22	23	25	
29	29	31	
36	36	38	
43	43	46	
50	50	52	
57	58	61	
64	64	66	
71	71	73	
78	78	81	
85	85	87	
92	92	94	
99	99	101	
106	106	108	
113	113	116	
120	121	123	
127	127	129	
134	135	137	

P2 scheduled real release termination			
1	4	7	
13	14	19	
25	26	31	
37	38	41	
49	52	55	
61	63	68	
73	74	76	
85	87	90	
97	101	104	
109	110	112	
121	123	126	
133	137	140	
145	146	148	
157	159	161	
169	171	175	
181	183	187	
193	195	197	
205	206	208	
217	219	222	
229	231	233	

P3 scheduled real release termination			
1	7	20	
21	25	41	
41	46	56	
61	68	81	
81	83	96	
101	104	117	
121	126	133	
141	143	150	
161	163	167	
181	187	192	
201	203	210	
221	222	229	
241	243	247	
261	263	270	
281	282	286	
301	303	309	
321	323	327	
341	343	350	
361	362	366	
381	383	389	

Response Time Analysis (cont)

- What happens when P2 computation time is increased from 3 to 4?



Response time simulation for configuration
Periods T1 T2 T3 = 7 12 20
Comp. time C1 C2 C3 = 3 4 5

P1 scheduled real release termination		
1	2	5
8	8	10
15	15	17
22	22	25
29	29	31
36	36	38
43	43	45
50	50	52
57	58	60
64	64	66
71	71	73
78	78	80
85	85	87
92	92	95
99	99	101
106	106	108
113	113	115
120	121	123
127	127	130
134	134	136

P2 scheduled real release termination		
1	5	11
13	14	20
25	26	32
37	38	42
49	52	56
61	62	68
73	74	77
85	87	91
97	101	105
109	110	116
121	123	130
133	136	140
145	146	149
157	159	162
169	171	176
181	182	188
193	195	198
205	206	209
217	219	222
229	231	236

P3 scheduled real release termination		
1	11	25 next release overrun
21	25	45 next release overrun
41	45	56
61	68	82 next release overrun
81	82	96
101	105	119
121	130	141
141	143	151
161	162	167
181	188	193
201	202	210
221	222	227
241	242	248
261	263	267
281	283	290
301	302	309
321	323	327
341	343	349
361	363	367
381	383	387

Revisiting: Process Set C

Process	Period T	ComputationTime C	Priority P	Response time R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

- The combined utilization is 1.0
- This was above the utilization threshold for three processes (0.78), therefore it failed the test
- However, a response time analysis shows that the process set will just meet its deadlines. `FPS_analysis.c` shows two marginal overruns.
- RTA Is sufficient and necessary . If the process set passes the test they will meet all their deadlines; if they fail the test then, at run-time, a process will miss its deadline (unless the computation time estimations themselves turn out to be pessimistic)

Worst-Case Execution Time - WCET



- Obtained by either measurement or analysis
- The problem with measurement is that it is difficult to be sure when the worst case has been observed
- The drawback of analysis is that an effective model of the processor (including caches, pipelines, memory wait states and so on) must be available. In real life this is simply not feasible.

WCET— Finding C



Most analysis techniques involve two distinct activities.

- The first takes the process and decomposes its code into a directed graph of basic blocks
- These basic blocks represent straight-line code.
- The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time
- Once the times for all the basic blocks are known, the directed graph can be collapsed
- Experimental approaches: see following pages



VxWorks timexLib

- **timexLib – execution timer facilities**

- **ROUTINES**

[timexInit\(\)](#) - include the execution timer library

[timexClear\(\)](#) - clear the list of function calls to be timed

[timexFunc\(\)](#) - specify functions to be timed

[timexHelp\(\)](#) - display synopsis of execution timer facilities

[timex\(\)](#) - time a single execution of a function or functions

[timexN\(\)](#) - time repeated executions of a function or group of functions

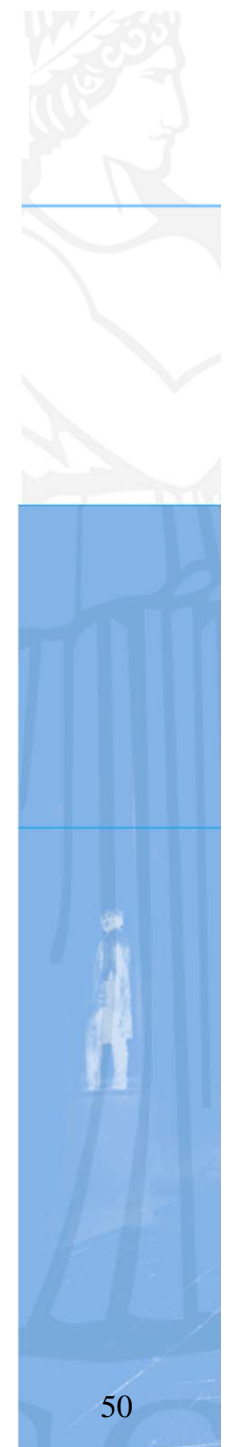
[timexPost\(\)](#) - specify functions to be called after timing

[timexPre\(\)](#) - specify functions to be called prior to timing

[timexShow\(\)](#) - display the list of function calls to be timed

- **DESCRIPTION**

- This library contains routines for timing the execution of programs, individual functions, and groups of functions. The VxWorks system clock is used as a time base. Functions that have a short execution time relative to this time base can be called repeatedly to establish an average execution time with an acceptable percentage of error.
- Comment: quite useful!



Using the Real-Time clock

- If a routine can be executed a large number of times, a 60 Hz Real-Time clock can give a good estimate
 - To obtain a good accuracy the number of loops must be large such that rounding-off errors is acceptable.
 - However, one can easily get fooled by such repetitive execution! After the first loop the code and data will probably be cached, which means that one will basically measure a cache based timing. To get a worst-case value: disable the caches!

```
.....  
time_t  starttime, stoptime, timer;  
    starttime = time(&timer);    /* calendar time in seconds */  
/* Execute many_loops os whatever it may be*/  
.....  
    stoptime = time(&timer);  
    microseconds_per_loop = 1000000*(stoptime-starttime)/many_loops;  
.....
```



Using instrumentation – state / bus analyzers

- A logic state or a bus analyzer can be used to accurately time the execution of a routine if the processor can access for instance VMEbus or PCI bus. One inserts identifiable bus transactions at the entry and exit of the routine, and the time difference between these transactions can be accurately measured.
 - This method is more suitable for periods up to a few millisecc
- The VxWorks WindView for Tornado and Stethoscope for Workbench are a very useful tools, but requires a high precision timer on the processor board in order to make time measurements in the microsecond range
 - We will take a closer look at Stethoscope in a later lecture



Characteristics of Periodic and Sporadic processes

- **Periodic:**
 - Activate regularly between fixed time intervals
 - Used for polling, monitoring and sampling
 - However, do not try to use periodic processes for sampling say audio at 44 kHz!
 - Predetermined amount of work (execution time) every period
- **Sporadic**
 - Event driven – some external signal or change
 - Used for fault detection, change of operating modes, etc
 - Sporadic processes can be handled as equivalent periodic processes if they take little time to execute
- **Aperiodic processes:**
 - Sporadic which do not have a minimum inter-arrival time.

The distinction between Sporadic and Aperiodic is mainly an issue for computer scientists, ref numerous papers one can find on scheduling

Sporadic Processes

- Sporadic processes have a minimum inter-arrival time
- They also require $D < T$
- The response time algorithm for fixed priority scheduling works perfectly for values of D less than T as long as the stopping criteria becomes $W_i^{n+1} > D_i$
- It also works perfectly well with any priority ordering — $hp(i)$ always gives the set of higher-priority processes

Aperiodic Processes

- These do not have minimum inter-arrival times
- Can run aperiodic processes at a priority below the priorities assigned to hard processes, therefore, they cannot steal, in a pre-emptive system, resources from the hard processes
- This does not provide adequate support to soft processes which will often miss their deadlines
- To improve the situation for soft processes, a **server** can be employed.
- Servers protect the processing resources needed by hard processes but otherwise allow soft processes to run as soon as possible.
- POSIX supports Sporadic Servers

http://faculty.cs.tamu.edu/bettati/Courses/663/2007C/Slides/aperiodic_sporadic.pdf

<http://www.usenix.org/publications/login/standards/22.posix.html>

Hard and Soft Processes



- In many situations the worst-case figures for sporadic processes are considerably higher than the averages
- Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation
 - How to handle hardware malfunction is a key issue in Real-Time systems, more about that in a later lecture
- Measuring schedulability with worst-case figures may lead to very low processor utilizations being observed in the actual running system

General Guidelines

Rule 1 — all processes should be schedulable using average execution times and average arrival rates

Rule 2 — all hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

- A consequent of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines
- This condition is known as a **transient overload**
- Rule 2 ensures that no hard real-time process will miss its deadline
- If Rule 2 gives rise to unacceptably low utilizations for “normal execution” then action must be taken to reduce the worst-case execution times (or arrival rates)

Process Interactions and Blocking



- If a process is suspended waiting for a lower-priority process to complete some required computation then the priority model is, in some sense, being undermined
- It is said to suffer **priority inversion**
- If a process is waiting for a lower-priority process, it is said to be **blocked**

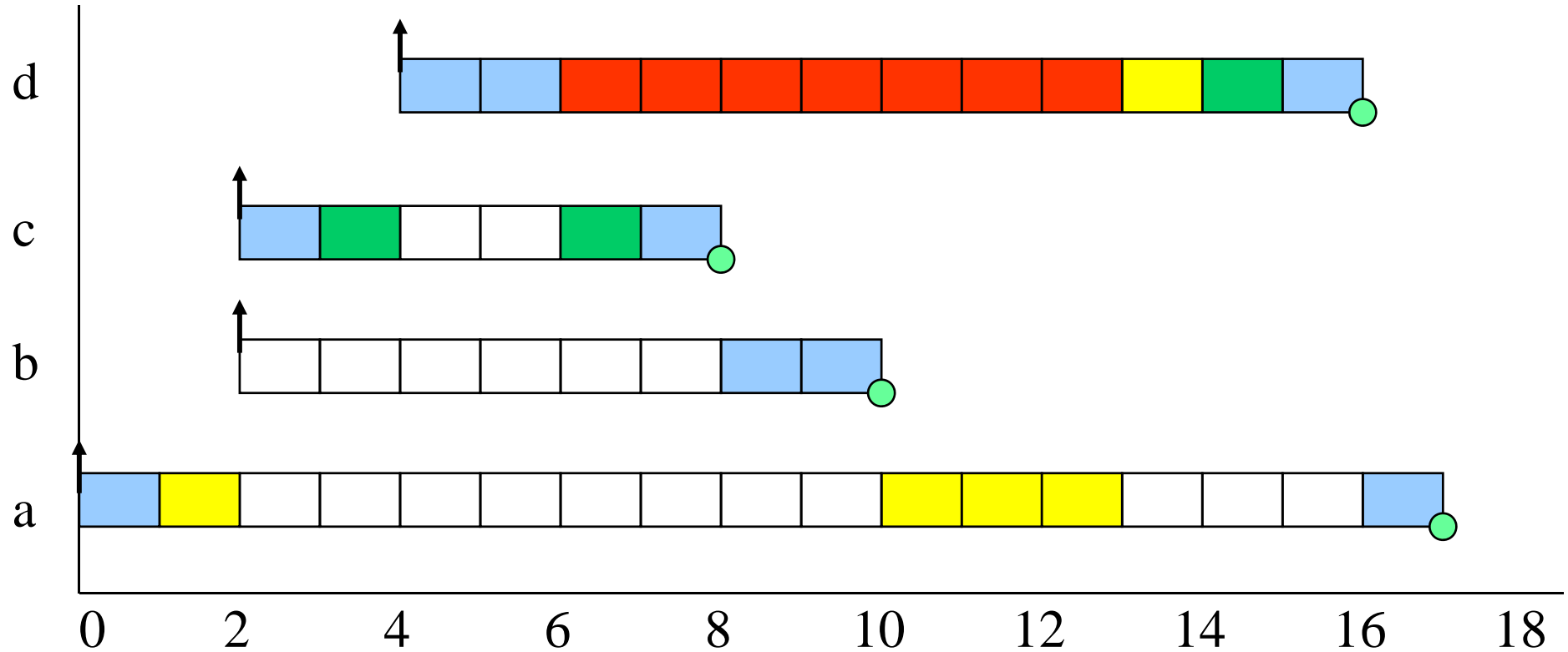
Priority Inversion

- To illustrate an extreme example of priority inversion, consider the executions of four periodic processes: a, b, c and d; and two resources: Q and V

Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Example of Priority Inversion

Process



Executing



Preempted



Executing with Q locked



Blocked

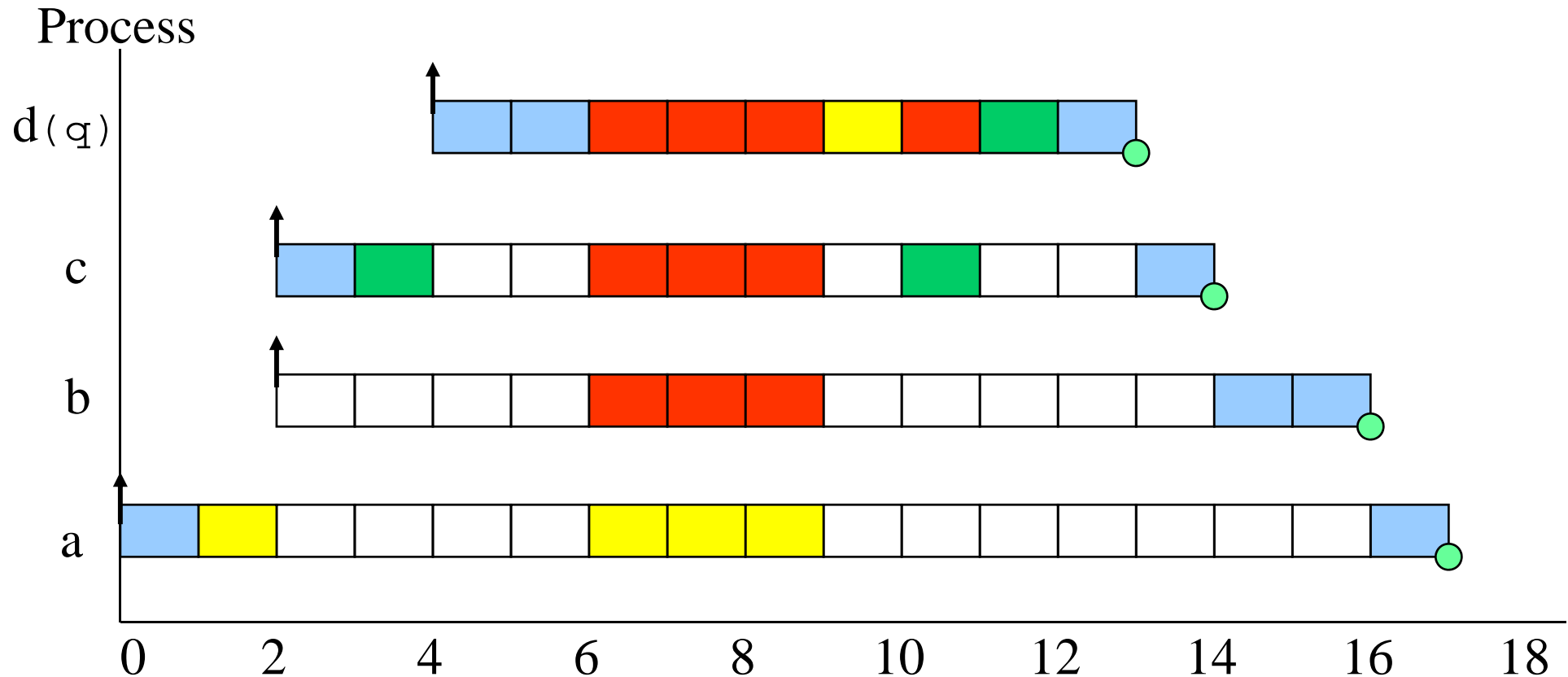


Executing with V locked

Process d has the highest priority

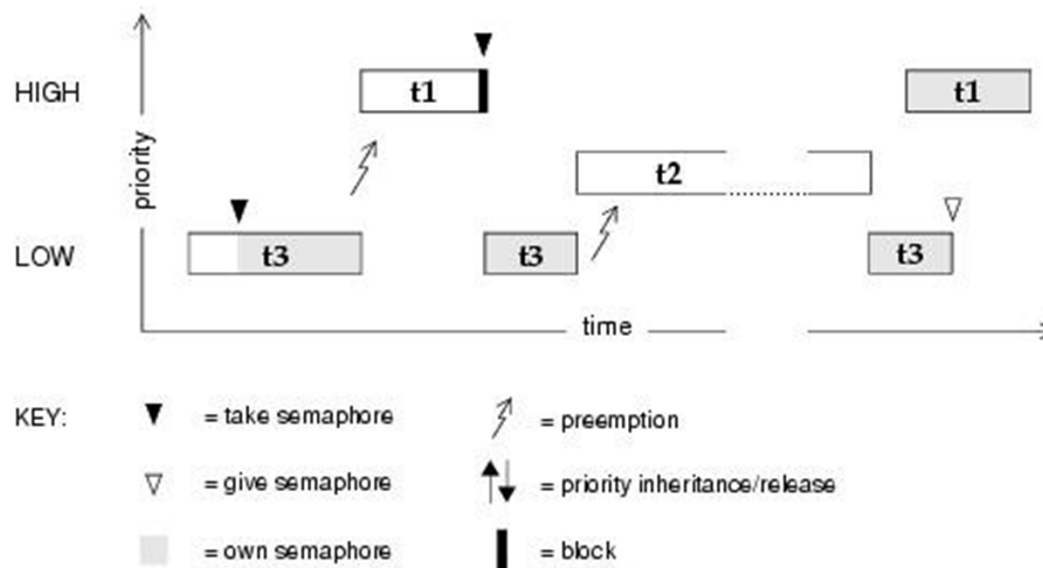
Priority Inheritance

- If process p is blocking process q , then p runs with q 's priority. In the diagram below, q corresponds to d , while both a and c can be p



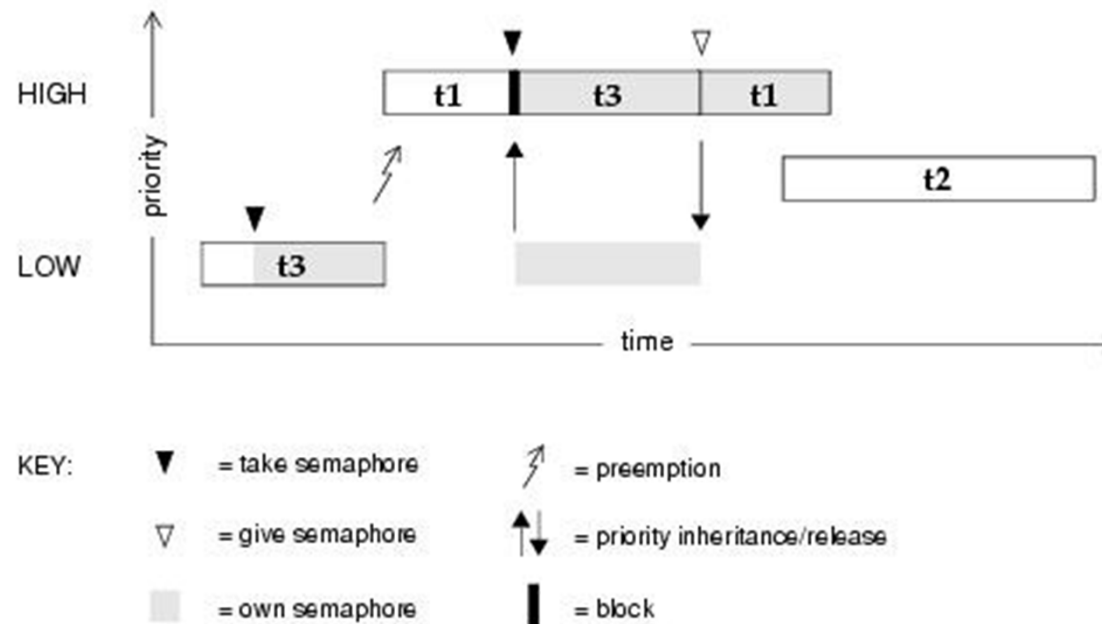
VxWorks Mutual-Exclusion semaphores and Priority inversion

- The mutual-exclusion semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.
- The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:
 - It can be used only for mutual exclusion.
 - It can be given only by the task that took it.
 - The **semFlush()** operation is illegal.
- Priority inversion problem:



VxWorks Priority inheritance

- In the figure below, priority inheritance solves the problem of priority inversion by elevating the priority of **t3** to the priority of **t1** during the time **t1** is blocked on the semaphore. This protects **t3**, and indirectly **t1**, from preemption by **t2**. The following example creates a mutual-exclusion semaphore that uses the priority inheritance algorithm:
 - `semId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);`
- Other VxWorks facilities which implement priority inheritance: next page



VxWorks POSIX Mutexes and Condition Variable



- Thread mutexes (mutual exclusion variables) and condition variables provide compatibility with the POSIX 1003.1c standard. They perform essentially the same role as VxWorks mutual exclusion and binary semaphores (and are in fact implemented using them). They are available with pthreadLib. Like POSIX threads, mutexes and condition variables have *attributes* associated with them. Mutex attributes are held in a data type called **pthread_mutexattr_t**, which contains two attributes, **protocol** and **prioceiling**.
- The **protocol** mutex attribute describes how the mutex variable deals with the priority inversion problem described in the section for VxWorks mutual-exclusion semaphores.
 - Attribute Name: **protocol**
 - Possible Values: **PTHREAD_PRIO_INHERIT** (default) and **PTHREAD_PRIO_PROTECT**
 - Access Routines: **pthread_mutexattr_getprotocol()** and **pthread_mutexattr_setprotocol()**
- Because it might not be desirable to elevate a lower-priority thread to a too-high priority, POSIX defines the notion of priority ceiling. Mutual-exclusion variables created with *priority protection* use the **PTHREAD_PRIO_PROTECT** value.

Priority Ceiling Protocols



Two other protocol to minimize blocking:

- Original ceiling priority protocol
- Immediate ceiling priority protocol

Will not be discussed further here

POSIX



- POSIX supports priority-based scheduling, and has options to support priority inheritance and ceiling protocols
- Priorities may be set dynamically
- Within the priority-based facilities, there are four policies:
 - FIFO: a process/thread runs until it completes or it is blocked
 - Round-Robin: a process/thread runs until it completes or it is blocked or its time quantum has expired
 - Sporadic Server: a process/thread runs as a sporadic server
 - OTHER: an implementation-defined
- For each policy, there is a minimum range of priorities that must be supported; 32 for FIFO and round-robin
- The scheduling policy can be set on a per process and a per thread basis

POSIX



- Threads may be created with a **system contention** option, in which case they compete with other system threads according to their policy and priority
- Alternatively, threads can be created with a **process contention option** where they must compete with other threads (created with a process contention) in the parent process
 - It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention
- A specific implementation must decide which to support

Other POSIX Facilities



POSIX allows:

- priority inheritance to be associated with mutexes (priority protected protocol= ICPP)
- message queues to be priority ordered
- functions for dynamically getting and setting a thread's priority
- threads to indicate whether their attributes should be inherited by any child thread they create