CIS 721 - Real-Time Systems

Lecture 13: VxWorks RTOS

Mitch Neilsen
neilsen@ksu.edu

* some slides from Wind River's VxWorks documentation

Outline

- **Priority Inheritance Protocols**
  - Utilization-Based Test
  - **Response Time Analysis**
- **Real-Time Operating System (VxWorks)**
  - **System Development Environment**
  - **Priority Inheritance Protocols**
  - **Priority Ceiling Protocols**

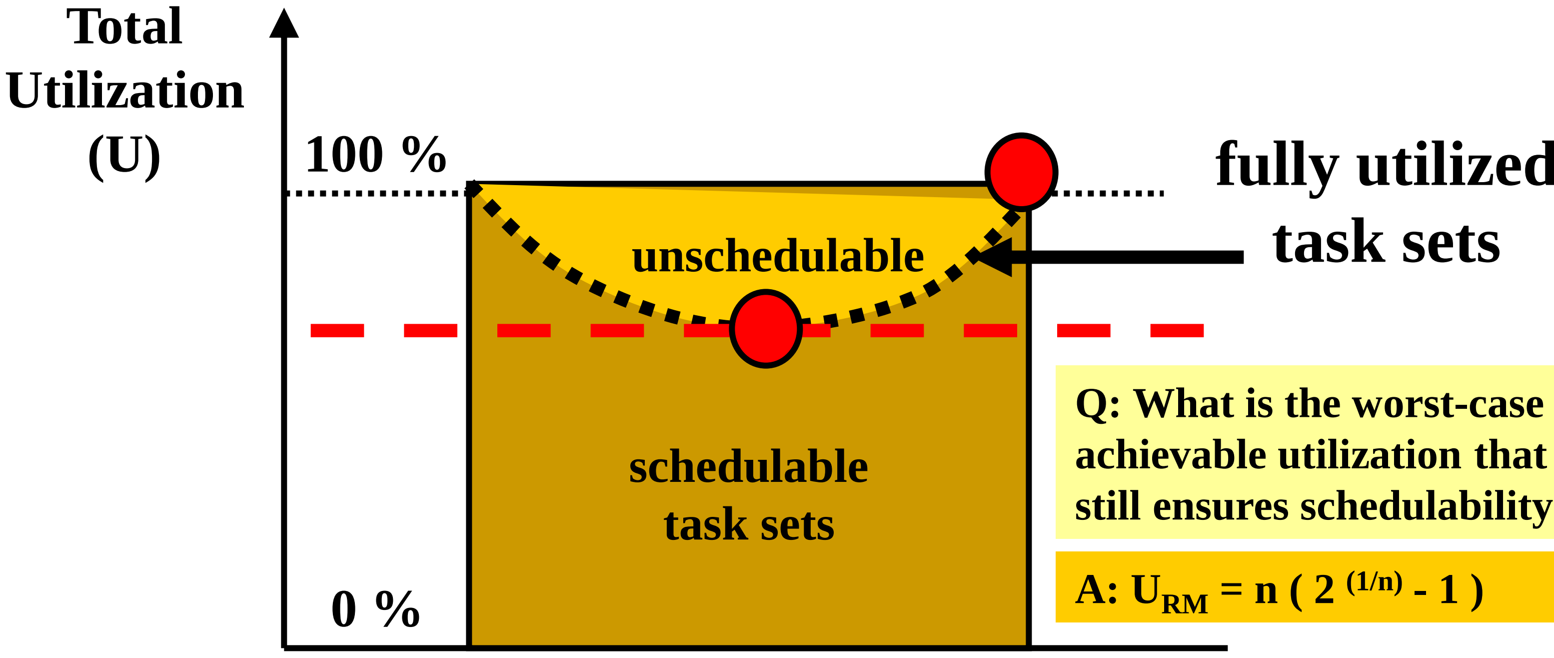
Utilization-Based Test

- Given a periodic task τ_i , the ratio $u_i = C_i \,/\, T_i$ is called the **utilization of task** τ_i .
- The **total utilization U** of all tasks in a system is the sum of the utilizations of all individual tasks:

$$\boxed{U = \sum_{i=1}^{n} \frac{C_i}{T_i}}$$

Utilization-Based Test

If $U + \max\{B_1 / T_1, B_2 / T_2, \ldots, B_n / T_n\} \le n(2^{1/n} - 1)$,
then the task set is feasible.



Q: What is the worst-case achievable utilization that still ensures schedulability?

A: $U_{RM} = n(2^{(1/n)} - 1)$

Example

Task τ_i	Period T_i	Blocking Time B_i	Run-Time C_i
τ_1	100	20	40
τ_2	150	30	40
τ_3	350	0	100

$U = 40 / 100 + 40 / 150 + 100 / 350$
$= 0.4 + 0.267 + 0.286 = 0.953$, so
$U + \max\{B_1 / T_1, B_2 / T_2, \ldots, B_n / T_n\}$
$= 0.953 + 0.2 > 1.0$.

Consequently, the test is **inconclusive**.

Response Time Analysis

- The **response time** (R_i) for task τ_i is given by the implicit equation:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

- Solve by forming a recurrence relation:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil * C_j$$

$$w_i^0 = C_i + B_i$$

Note: Arbitrary phasing is assumed.

Solving The Recurrence

- The sequence $w_i^0, w_i^1, w_i^2, \ldots, w_i^n$

  is clearly non-decreasing:

  - If $w_i^{n+1} = w_i^n$, then a fixed point (solution) has been found.

  - If $w_i^{n+1} > T_i$, then no solution exists.

Algorithm

Input: $C_1, C_2, \ldots, C_m, B_1, B_2, \ldots, B_m, T_1, T_2, \ldots, T_m$

Output: $R_1, R_2, \ldots, R_m$

for $i = 1$ to m

$n = 0$

$w_i^n = C_i + B_i$

loop

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil * C_j$$

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

$R_i = w_i^n$

break out of loop {solution found}

if $w_i^{n+1} > T_i$ then

break out of loop {no solution}

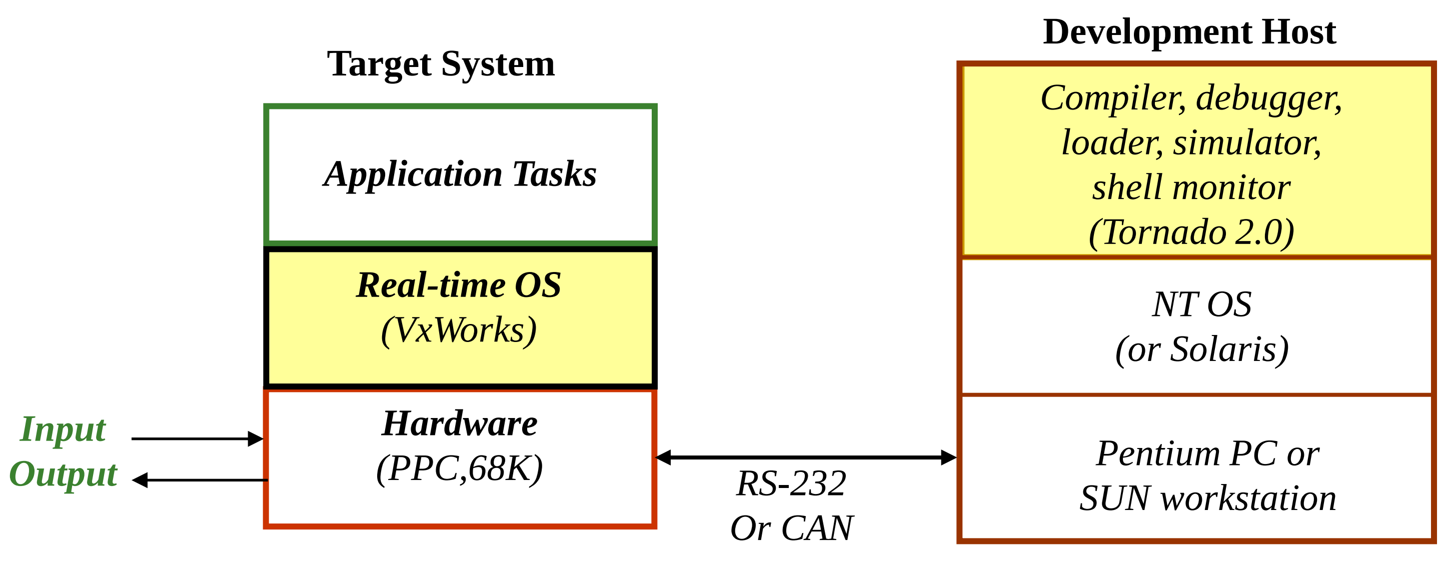
$n = n + 1$

end loop

end for

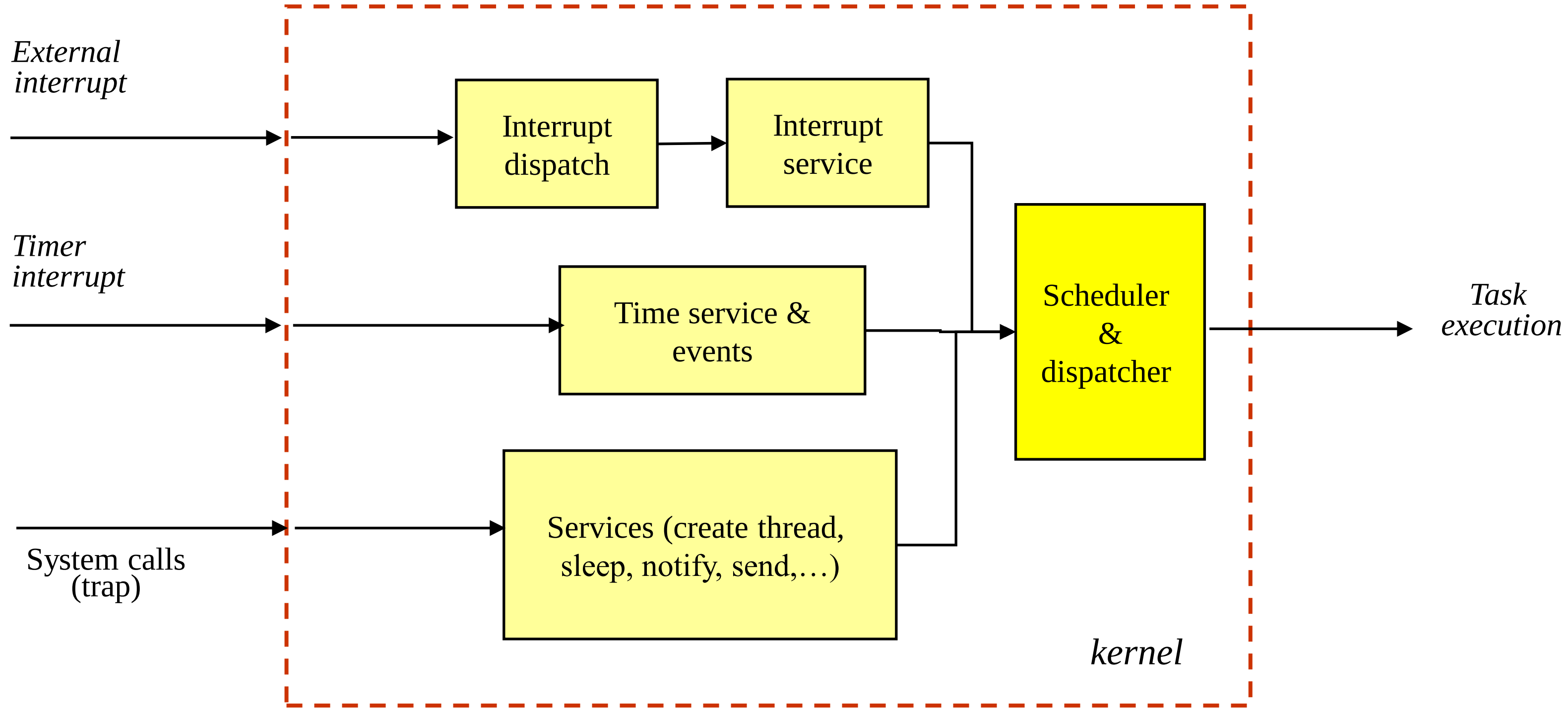
Real-Time System Development

- Real-Time Operating System (VxWorks)
- Real-Time Development Environment (Tornado 2.0)

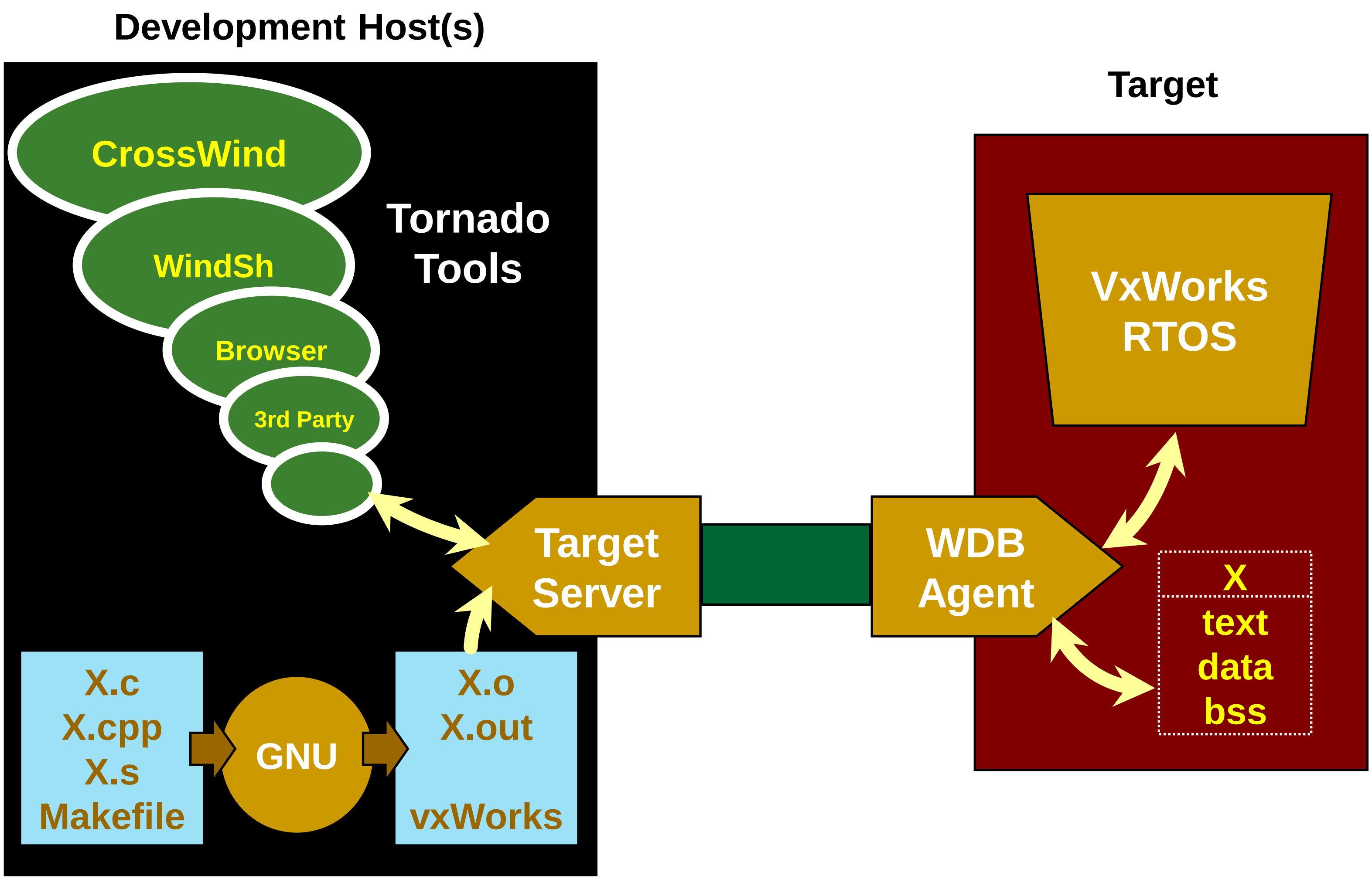


Real-Time OS (RTOS)

- Functions: **task management**, time service, device drivers, and interrupt service.



Tornado Architecture



VxWorks Task

- A **task** is a *context of execution*. It has:
  - a **program counter** (PC).
  - private copies of other CPU **registers**.
  - a **stack** for local variables, function arguments, and function call chain information.
- Single processor system: only one task is scheduled for execution at any given time.
  - When a task is not executing, its context is stored in its *Task Control Block* (TCB) and stack. The TCB is the data structure which the kernel uses to represent and control the task.

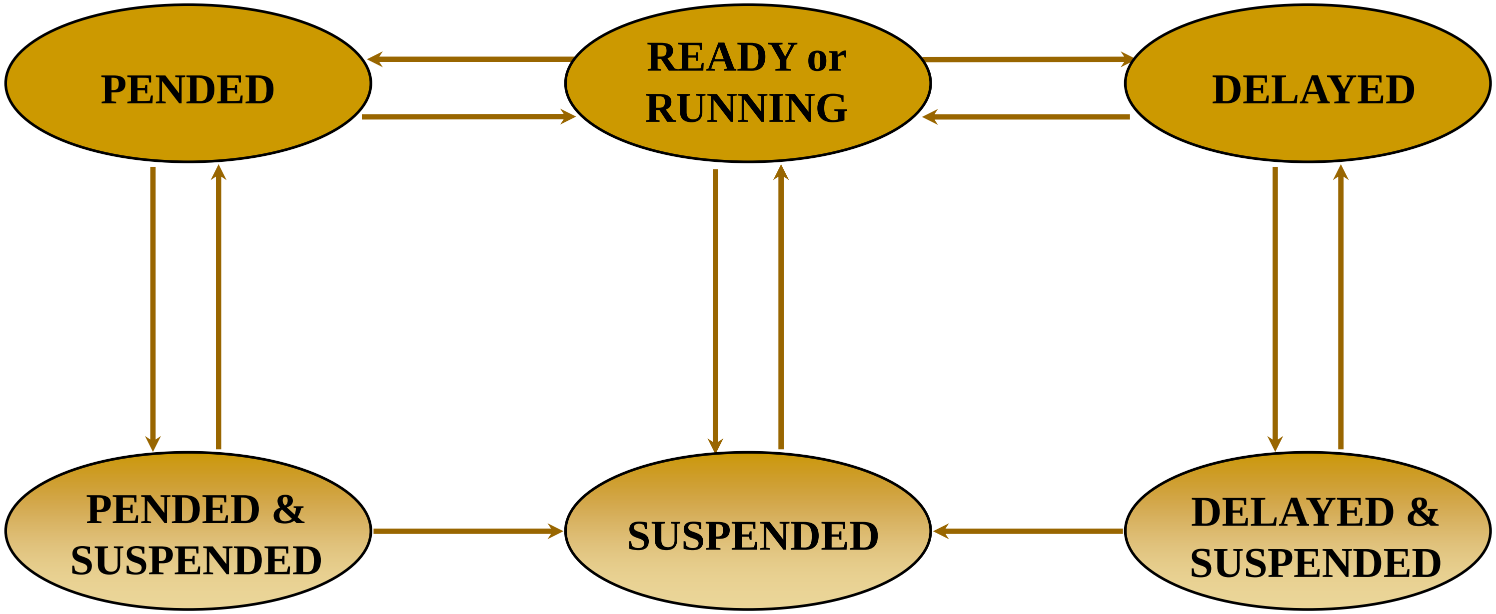
Task Creation Routines

- **taskSpawn( )** - spawn (create and activate) a new task.
- **taskInit( )** - initialize (create and suspend) a new task.
- **taskActivate( )** - activate an initialized task.

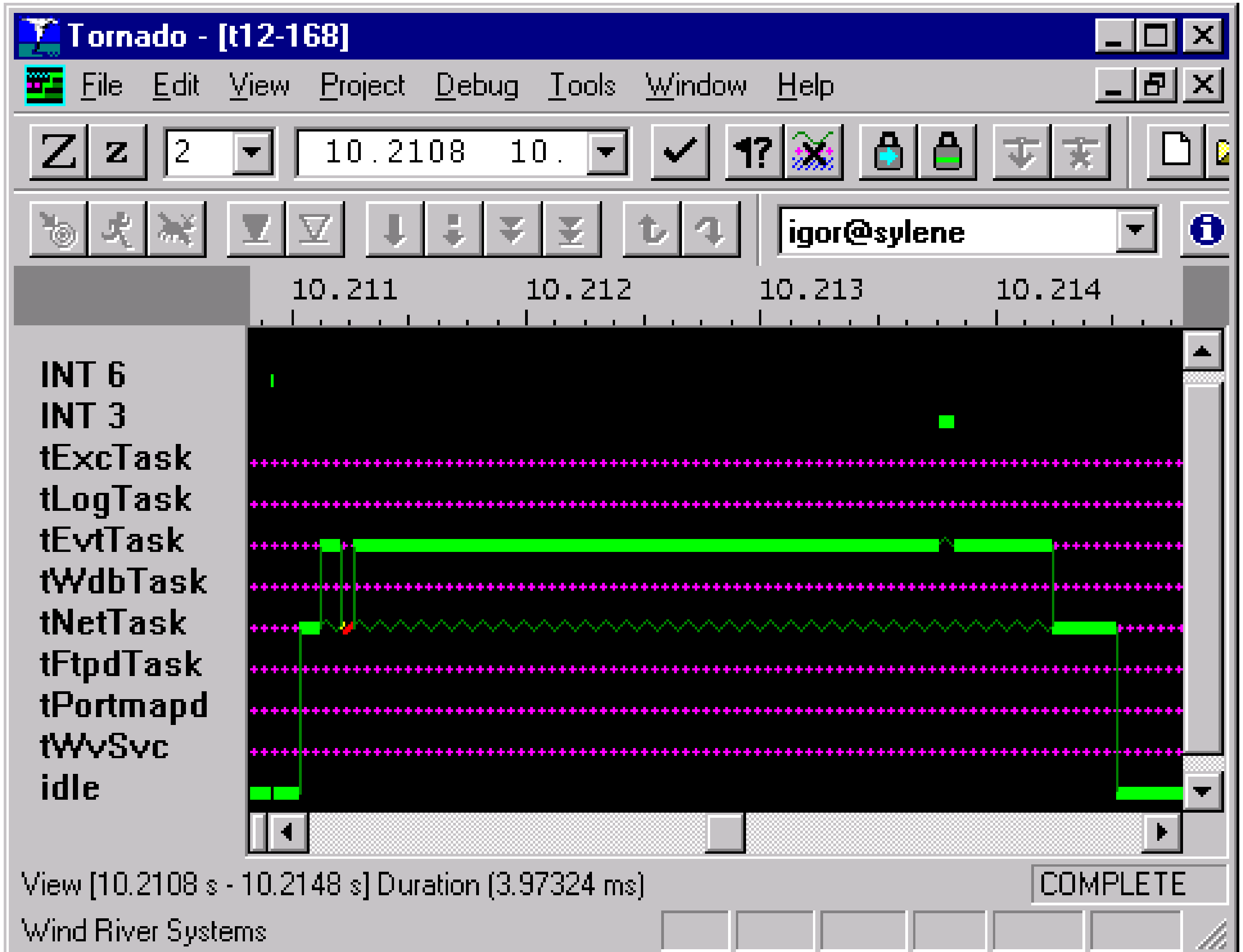
Task State Transitions

- **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 pended-suspended tasks can still unblock and delayed-suspended tasks can still awaken.

Task State Transition Diagram



Preemptive Priority Scheduling



At any time, the highest priority task *ready* to run, executes!

A higher priority task that becomes ready preempts an executing task.

Interrupts preempt any task.

Notes: On this board, INT 6 is the system clock, INT 3 is the network interrupt.

Round Robin Scheduling

- Preemptive priority scheduling can be augmented with round-robin scheduling.
- Round-robin scheduling is used to share the CPU fairly among all ready tasks at the **same priority level** using time slicing.
- Each task of a group of tasks executes for a defined interval, or time slice; then another task executes for its interval in the cycle.
- The allocation is fair in that no task in the group gets a second slice of time before the other tasks in the group are given a slice.

Kernel Timeslice

- Round-robin scheduling can be enabled with the routine **kernelTimeSlice**( ), which takes a parameter for a timeslice or interval length.
- This interval (a variable-length quantum) is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task.

taskLib

- taskLib contains the kernel functions for creating, destroying, starting and stopping tasks.
- Example routines:
  - To create and start a new task:
    - int **taskSpawn** (name, priority, options, stackSize, entryPt, arg0, … , arg9)
  - To delay the executing task for a certain number of system clock ticks:
    - STATUS **taskDelay** (ticks)

Semaphores

- Semaphores are VxWorks kernel objects that allow blocking and unblocking of tasks; they are used to coordinate actions of a given task with those of other tasks and with external events.
- VxWorks provides three varieties of semaphores:
  - Binary (synchronization) semaphores.
  - Counting semaphores.
  - Mutex (mutual exclusion) semaphores.
- Each type of semaphore is intended primarily for a particular kind of programming problem.

Binary Semaphores

- Binary semaphores allow tasks to wait for an event without taking up CPU time polling for the event to occur.
- The event could be generated by an interrupt or another task.
- Usage:
  - Create the binary semaphore using **semBCreate**( )
  - Call **semTake**( ) to block until the event occurs
  - Signal a blocked task using **semGive**( )

Example

```
#include "vxWorks.h"
#include "semLib.h"
SEM_ID semMutex;

/* Create binary semaphore initialized to 1, enqueue blocked tasks in
  priority order */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);

semTake (semMutex, WAIT_FOREVER);
   . .  critical section ..
semGive (semMutex);
```

Mutex Semaphores

- Mutex semaphores are used when multiple tasks share a resource (data structure, file, hardware).
- When used correctly, mutex semaphores prevent multiple tasks from accessing the resource at the same time, and possibly corrupting it. Usage:
  - Create mutex for the resource with **semMCreate**( ).
  - A task wanting to use the resource calls **semTake**( ) to block until the resource is available (or time-out).
  - When done with the resource, a task calls **semGive**( ) to allow other tasks to use the resource. Any task locking a resource must also unlock the resource.

Mutex Semaphores

While binary or counting semaphores may be used for mutual exclusion, mutex semaphores are designed to deal with three common problems which arise in mutual exclusion:

- Mutual exclusion semaphores can be locked in a **recursive fashion**. A task that owns a mutex can acquire it again without blocking. The task must release the mutex as many times as it has acquired it.
- If a task *tA* owns a ***delete-safe*** mutex semaphore, and another task *tB* tries to delete *tA*, then *tB* will block until *tA* gives up ownership of the mutex. Thus, the mutex owner is safe from deletion while operating on the shared resource.
- Mutual exclusion semaphores can use a priority inversion protocol to be ***inversion-safe***.

Priority Inheritance Protocol

- *Inversion-safe* mutexes prevent the priority inversion problem. A task which owns such a mutex is temporarily boosted to the priority of the highest priority task waiting for the mutex. It falls back to its regular priority when it no longer owns an inversion-safe mutex. This prevents a high priority task from being blocked for an indeterminate amount of time waiting for a mutex owned by a low priority task which cannot execute because it has been preempted by an intermediate priority task!
- The following example creates a mutual-exclusion semaphore that uses the priority inheritance algorithm:

```
semId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

Counting Semaphores

- **Binary semaphores** keep track of whether or not an event has occurred, but not *how many times* the event has occurred (since the last time the event was serviced).
- **Counting semaphores** keep a count of how many times the event has occurred, but not serviced.
  - May be used to ensure that the event is serviced as many times as it occurs.
  - May also be used to maintain an atomic count of multiple equivalent available resources.

Semaphore Operations

- **semTake**(S.semid, timeout)

  if (S.count > 0) then S.count --;

  else block calling task in S.waitingQ;

- **semGive**(S.semid)

  if (S.waitingQ is non-empty) then

      wakeup a process in S.waitingQ;

  else S.count ++;

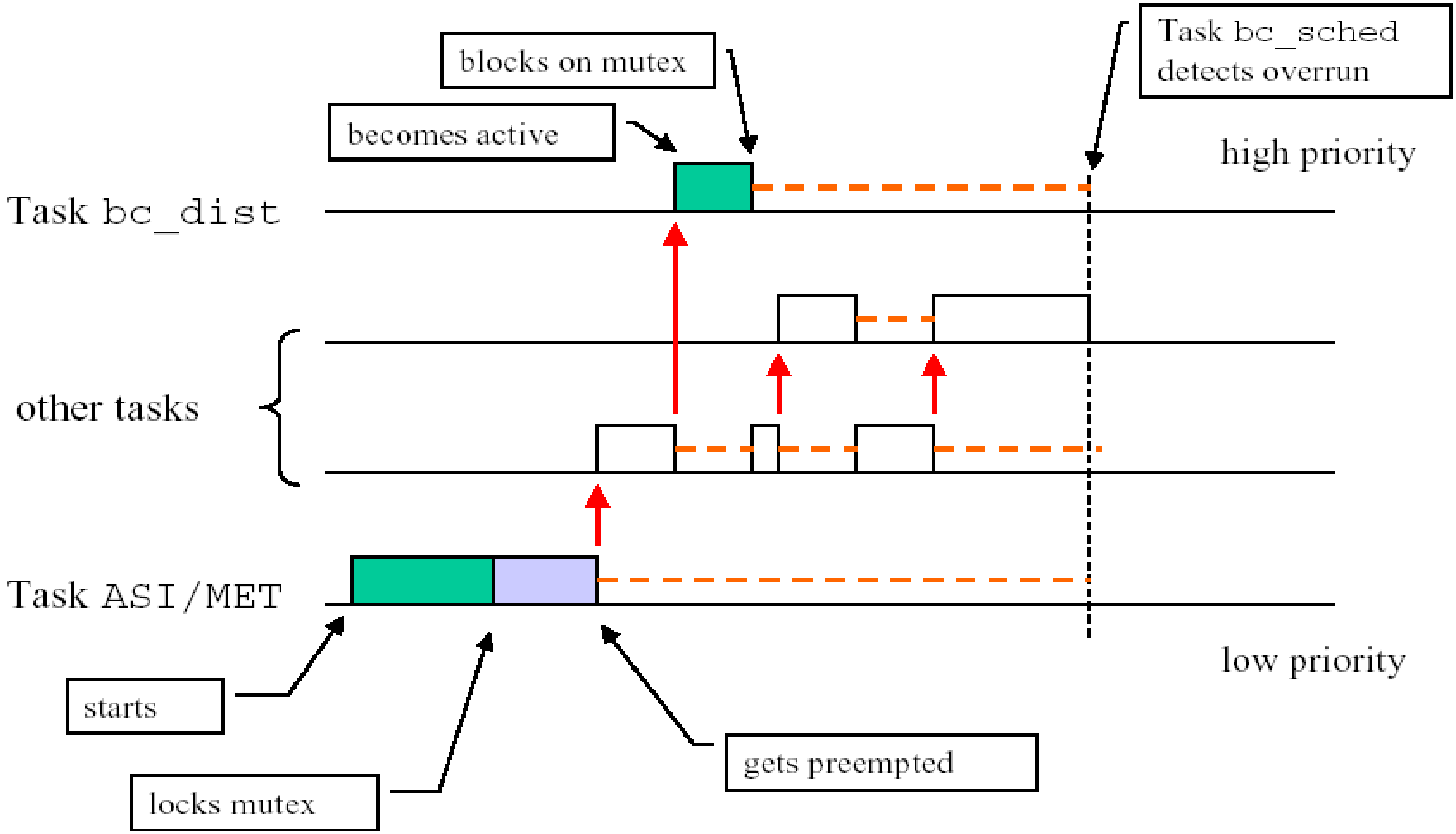
Semaphore Options

- Tasks pending on a semaphore can be queued in priority order (**SEM_Q_PRIORITY**) or in first-in first-out order (**SEM_Q_FIFO**).
- When tasks try to lock (take) a semaphore, they can specify a specific length of time to wait or **WAIT_FOREVER**.
- Priority inheritance protocols can be enabled using **SEM_INVERSION_SAFE**, this option can only be used with a priority queue; e.g., **SEM_Q_PRIORITY.**

Example: Mars Pathfinder

- Based on VxWorks.
- Developed a mysterious communications problem shortly after successfully landing on Mars on July 4, 1997.
- The problem was due to a classical case of priority inversion which caused a high priority thread to detect a missed deadline and reset the system.
- The design team dynamically modified the system through the debugging software which was still enabled.

Priority Inversion Problem



Example: myproject\cobble.c

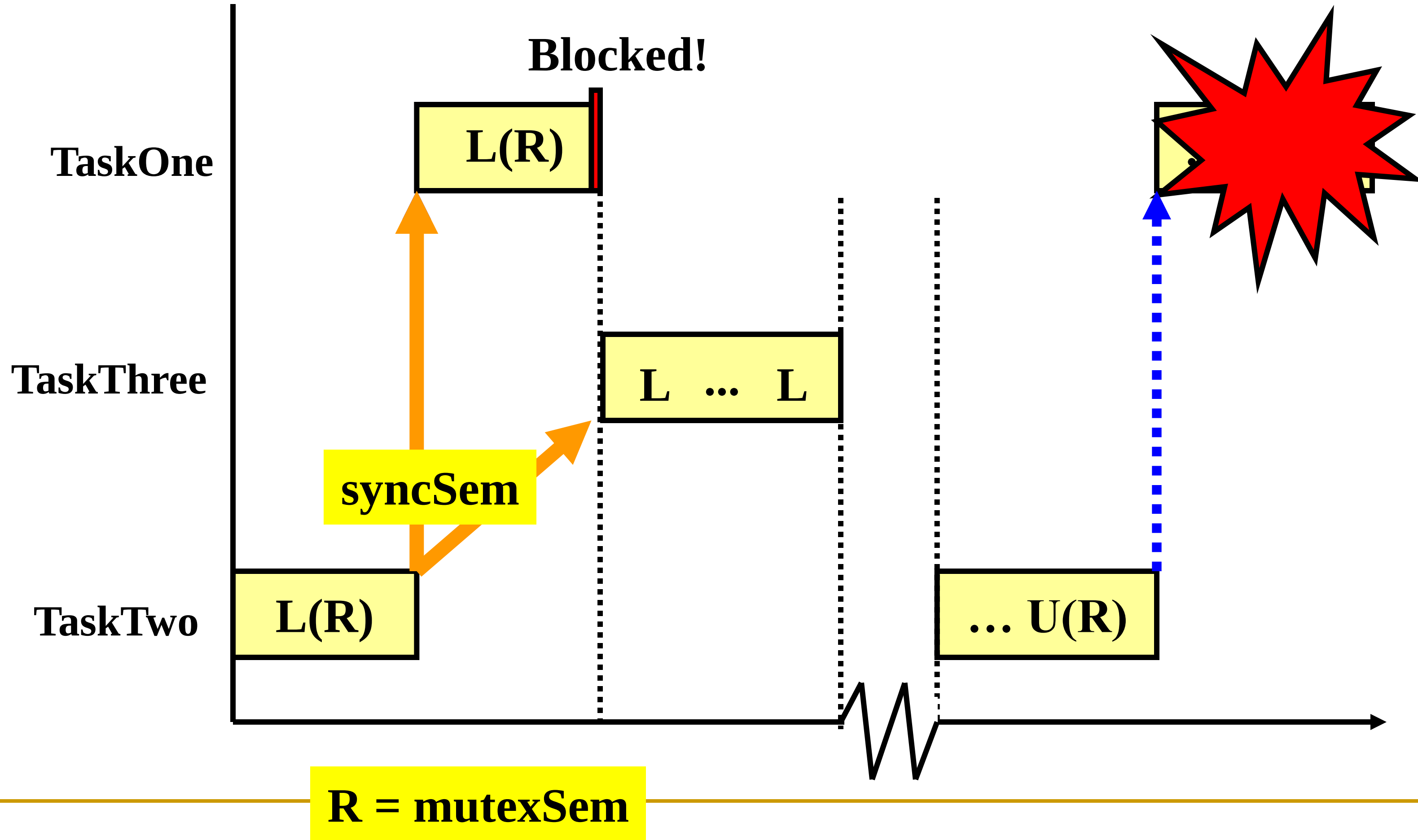
```
STATUS progStart (void)
{
    syncSemId = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    mutexSemId = semMCreate (  SEM_Q_PRIORITY
                             | SEM_DELETE_SAFE);
    runState = 0;

    taskOneId = taskSpawn("taskOne", 200, 0, STACK_SIZE,
        (FUNCPTR) taskOne,0,0,0,0,0,0,0,0,0,0);

    taskTwoId = taskSpawn("taskTwo", 220, 0, STACK_SIZE,
        (FUNCPTR) taskTwo,0,0,0,0,0,0,0,0,0,0);

    taskThreeId = taskSpawn("taskThree", 210, 0, STACK_SIZE,
        (FUNCPTR) taskThree,0,0,0,0,0,0,0,0,0,0);
    return (OK);
}
```

High Priority

Low Priority

Medium Priority

Priority Inversion



TaskOne – High Priority

```
void taskOne()
{
    int i,j;

    semTake (syncSemId, WAIT_FOREVER);
    semTake (mutexSemId, WAIT_FOREVER);
        for (i=1; i<=10; i++)
        {
            printf("TaskOne in CS, i = %d\n",i);
            for (j=1; j<=10000; j++);
        }
    semGive (mutexSemId);
    for (i=11; i<=20; i++)
    {
        printf("TaskOne out of CS, i = %d\n",i);
        for (j=1; j<=10000; j++);
    }
    runState++;
}
```

critical section

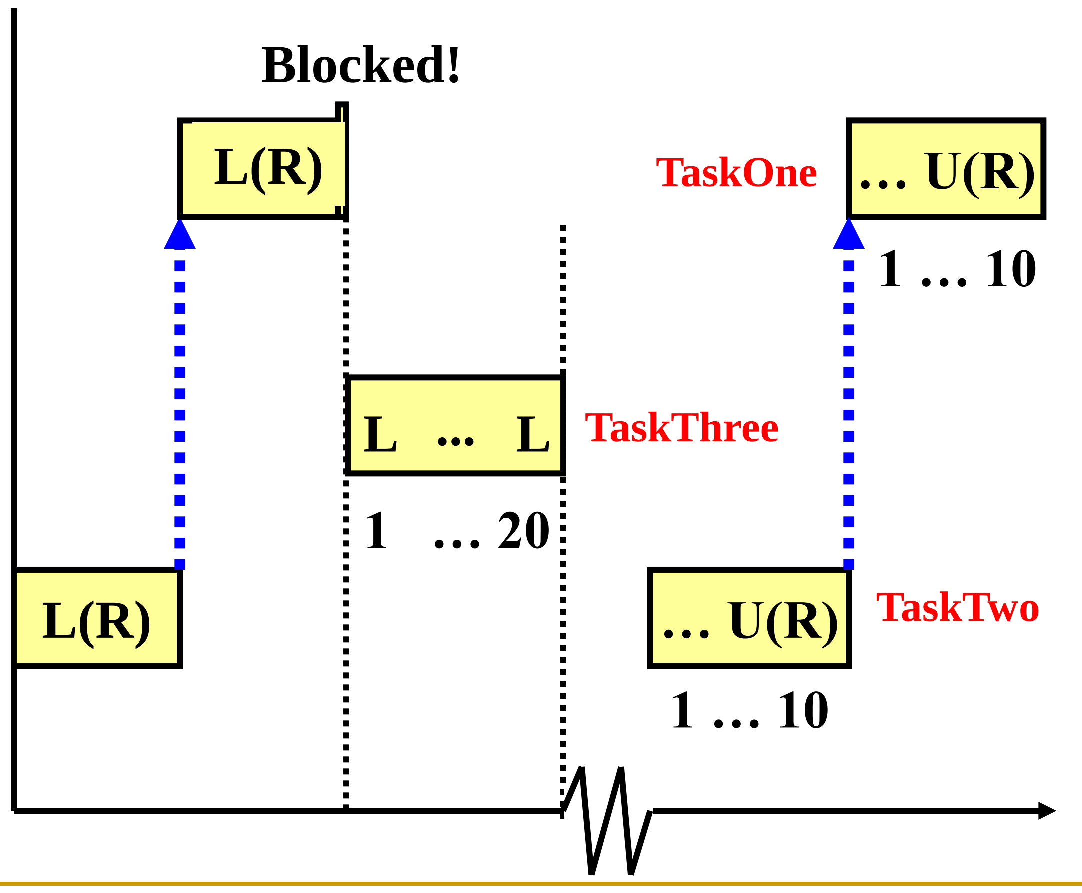
TaskTwo – Low Priority

```
void taskTwo()
{
    int i,j;

    semTake (mutexSemId, WAIT_FOREVER);
        semGive (syncSemId);
        semGive (syncSemId,);
        for (i=1; i<=10; i++)
        {
            printf("TaskTwo in CS, i = %d\n",i);
            for (j=1; j<=10000; j++);
        }
    semGive (mutexSemId);
    for (i=11; i<=20; i++)
    {
        printf("TaskTwo out of CS, i = %d\n",i);
    }
    runState++;
}
```

TaskThree – Middle Priority

```
void taskThree()
{
    int i,j;

    semTake (syncSemId, WAIT_FOREVER);
    for (i=1; i<=20; i++)
    {
        printf("TaskThree running, i = %d\n",i);
        for (j=1; j<=10000; j++);
    }
    runState++;
}
```

Output



TaskThree running, i = 1
TaskThree running, i = 2
…
TaskThree running, i = 20
TaskTwo in CS, i = 1
…
TaskTwo in CS, i = 10
TaskOne in CS, i = 1
…
TaskOne in CS, i = 10
TaskOne out of CS, i = 11
…
TaskOne out of CS, i = 20
TaskTwo out of CS, i = 11
…
TaskTwo out of CS, i = 20

Fix: myproject\cobble.c

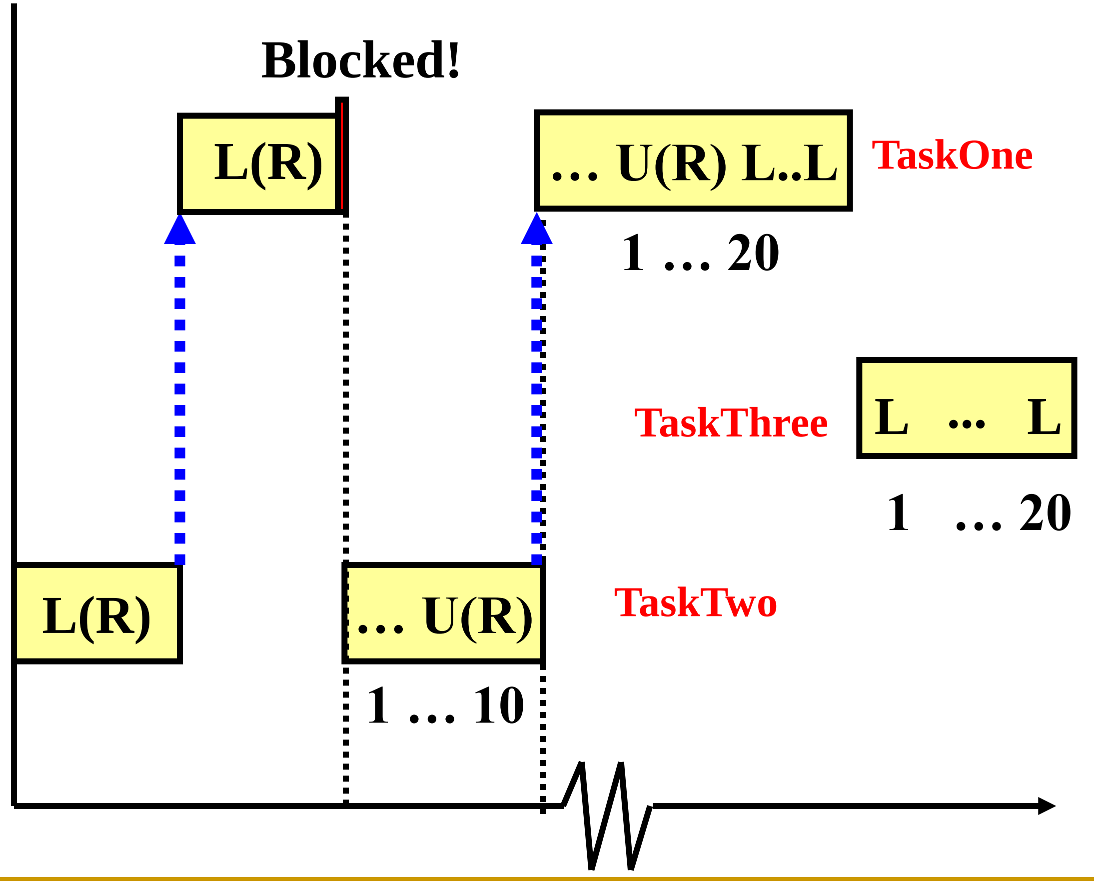
```
STATUS progStart (void)
{
    syncSemId = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    mutexSemId = semMCreate (   SEM_Q_PRIORITY
                              | SEM_INVERSION_SAFE
                              | SEM_DELETE_SAFE);
    runState = 0;

    taskOneId = taskSpawn("taskOne", 200, 0, STACK_SIZE,
        (FUNCPTR) taskOne,0,0,0,0,0,0,0,0,0,0);

    taskTwoId = taskSpawn("taskTwo", 220, 0, STACK_SIZE,
        (FUNCPTR) taskTwo,0,0,0,0,0,0,0,0,0,0);

    taskThreeId = taskSpawn("taskThree", 210, 0, STACK_SIZE,
        (FUNCPTR) taskThree,0,0,0,0,0,0,0,0,0,0);
    return (OK);
}
```

High Priority

Low Priority

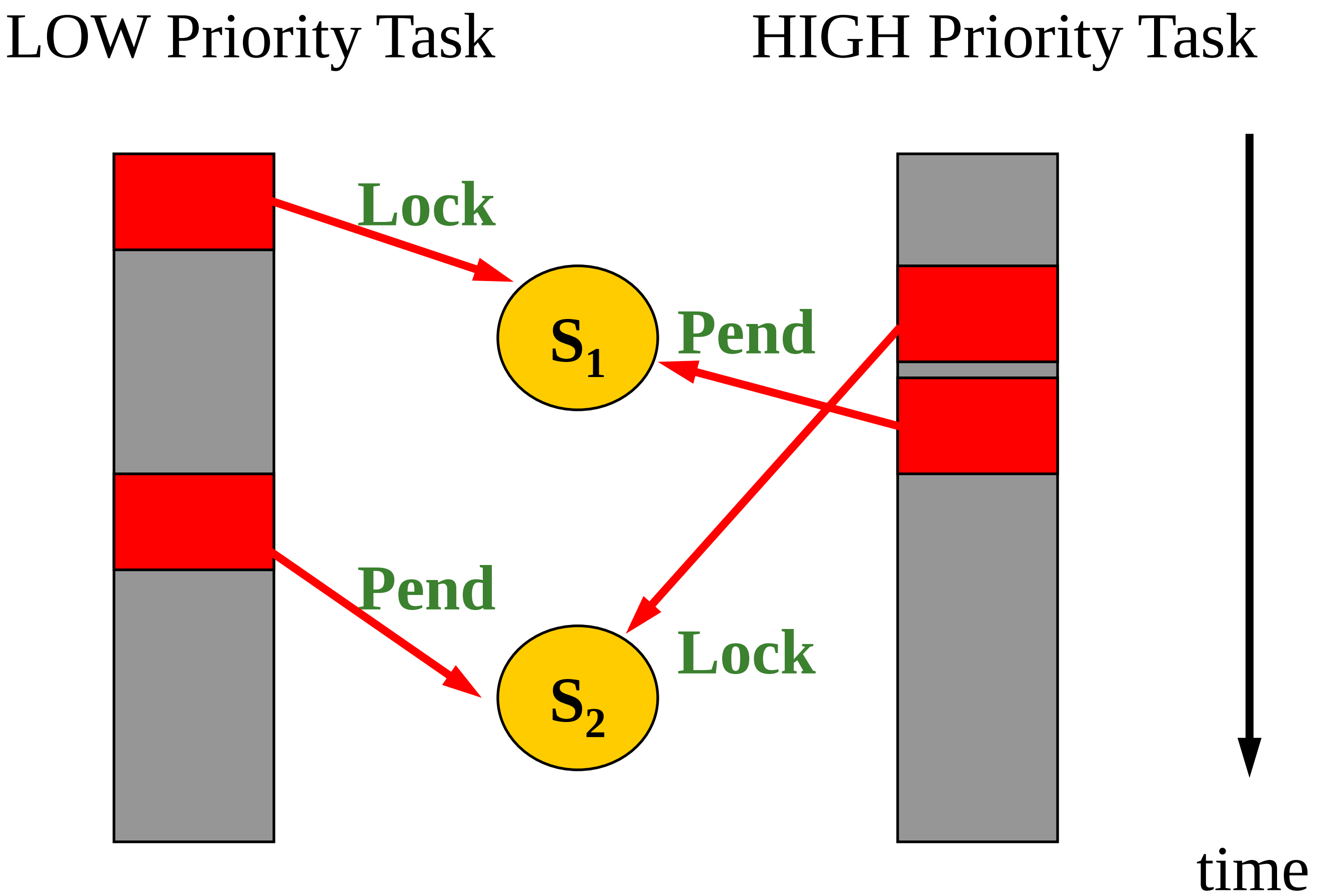
Medium Priority

Output



TaskTwo in CS, i = 1
...
TaskTwo in CS, i = 10
TaskOne in CS, i = 1
...
TaskOne in CS, i = 10
TaskOne out of CS, i = 11
...
TaskOne out of CS, i = 20
TaskThree running, i = 1
TaskThree running, i = 2
...
TaskThree running, i = 20
TaskTwo out of CS, i = 11
...
TaskTwo out of CS, i = 20

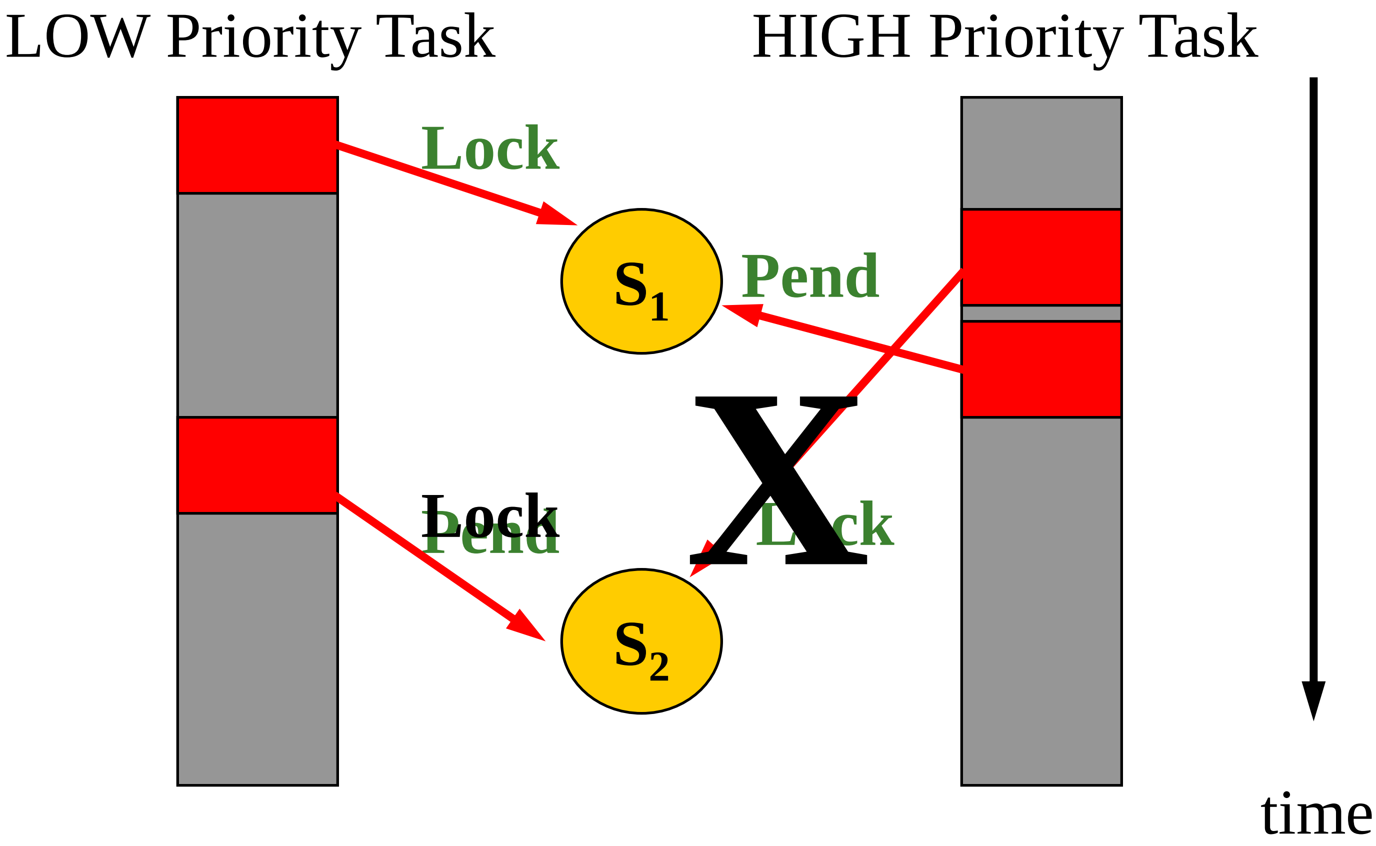
Priority Inheritance – Can Still Deadlock



Priority Ceiling Protocols

- Priority Ceiling Protocols prevent: Deadlock, Transitive Blocking, and Priority Inversion.
- Each task has a static priority.
- Each resource has a static ceiling value = the maximum priority of any task that may use it.
- Each task also has a dynamic priority = the maximum of its own static priority and the ceiling value of any resource it has locked.
- Max. blocking time B_i = max usage(k,i)*CS(k)

Priority Ceiling Protocols – No Deadlocks



VxWorks System Functions

- **Create and Activate a Task:**

```
taskSpawn ("tctrlSW", PRIO, 0, STACK_SIZE,
  (FUNCPTR) controlSoftware,0,0,0,0,0,0,0,0,0,0);
```

- **Create a Mutual Exclusion Semaphore**:

```
inverSemId =  semMCreate
  (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

- **Set Task Priority:**

```
taskPrioritySet(taskID, task_priority);
```

- **Measure the Execution Time of a Task:**

```
timexN((FUNCPTR) controlSoftware, NULL,NULL,
        NULL,NULL,NULL,NULL,NULL,NULL);
```

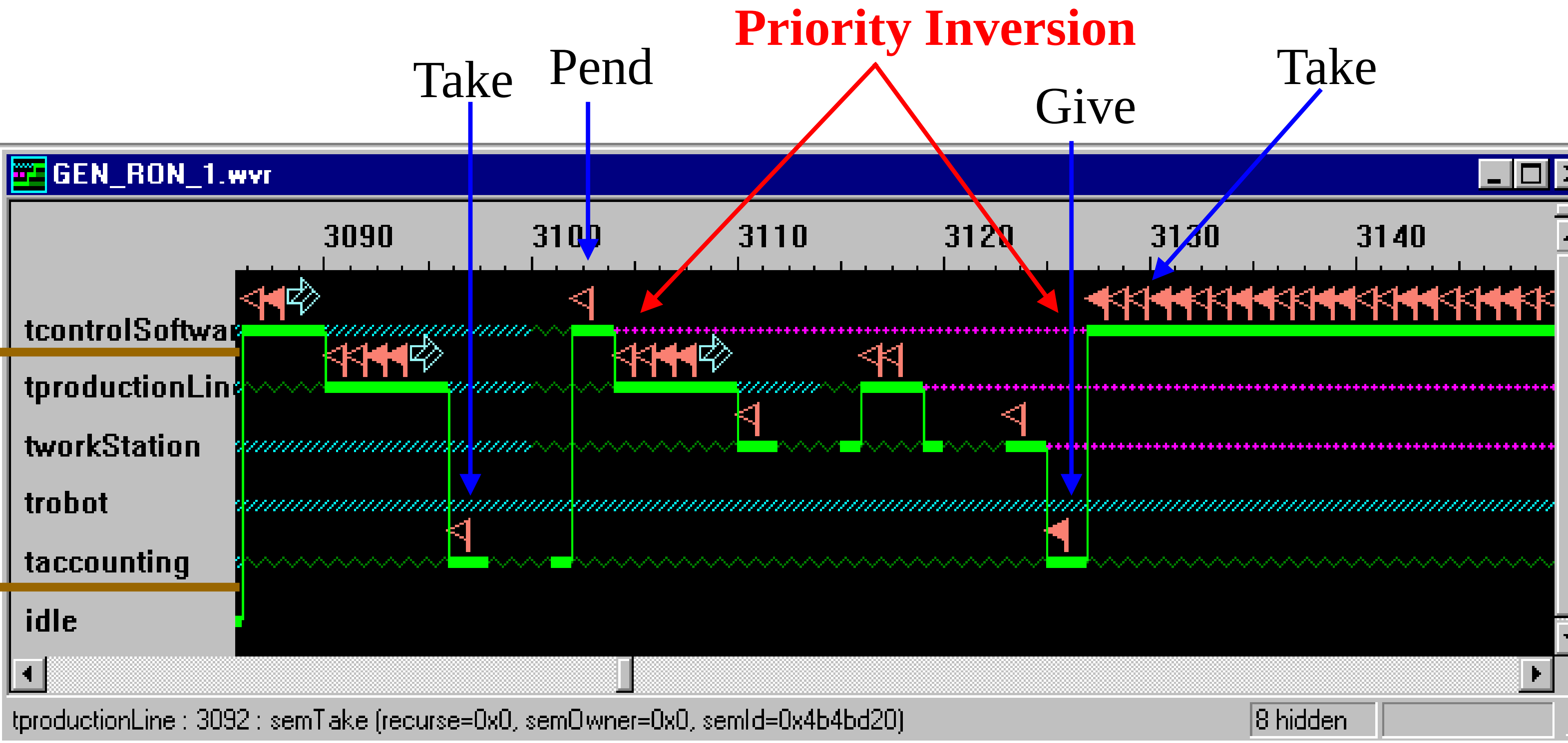
Example 2: Priority Inversion

```
void controlSoftware(void) {  //HIGH priority
     FOREVER {
      semTake(inverSemId, WAIT_FOREVER);
         for (long i=0; i < LONG_TIME; i++);  //wait for a while
      semGive(inverSemId);
}}
void accounting(void) {  //LOW priority
    FOREVER {
        semTake(inverSemId, WAIT_FOREVER);
           for (long i=0; i < 6 * LONG_TIME; i++);
        semGive(inverSemId);
}}
```

Suppose that the HIGH and LOW priority tasks compete for the same semaphore, and MEDIUM priority tasks run frequently.

Example 2: Priority Inversion

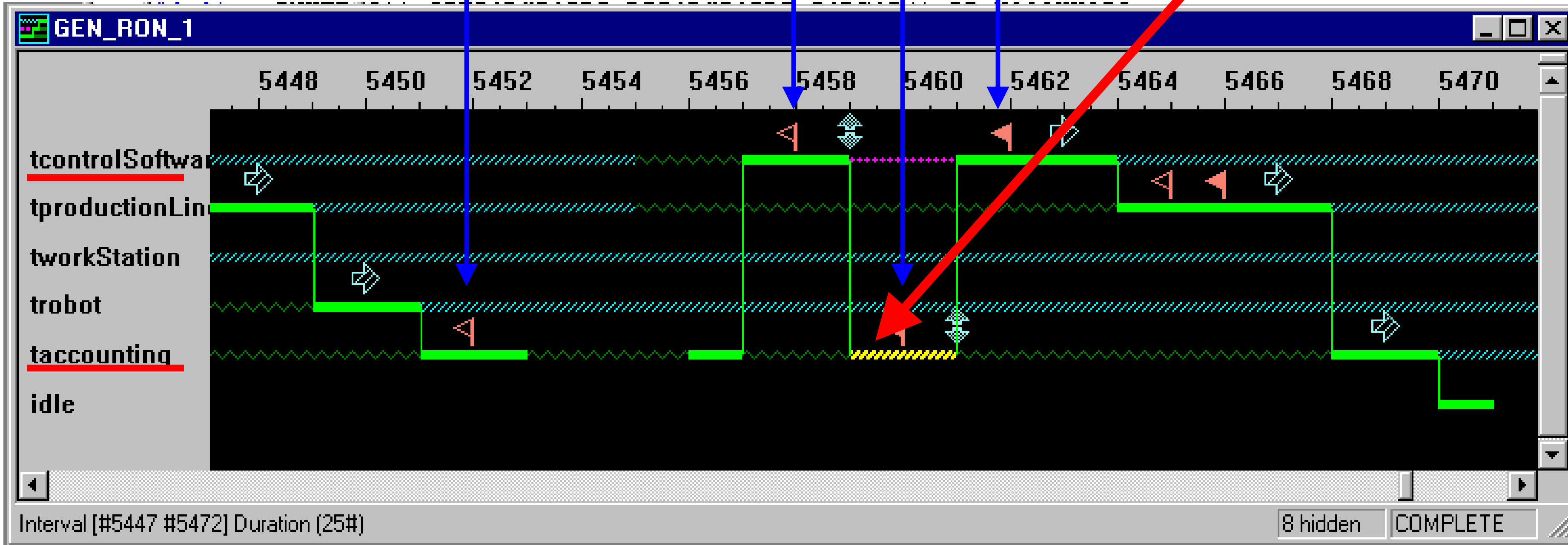
Semaphore Operations:



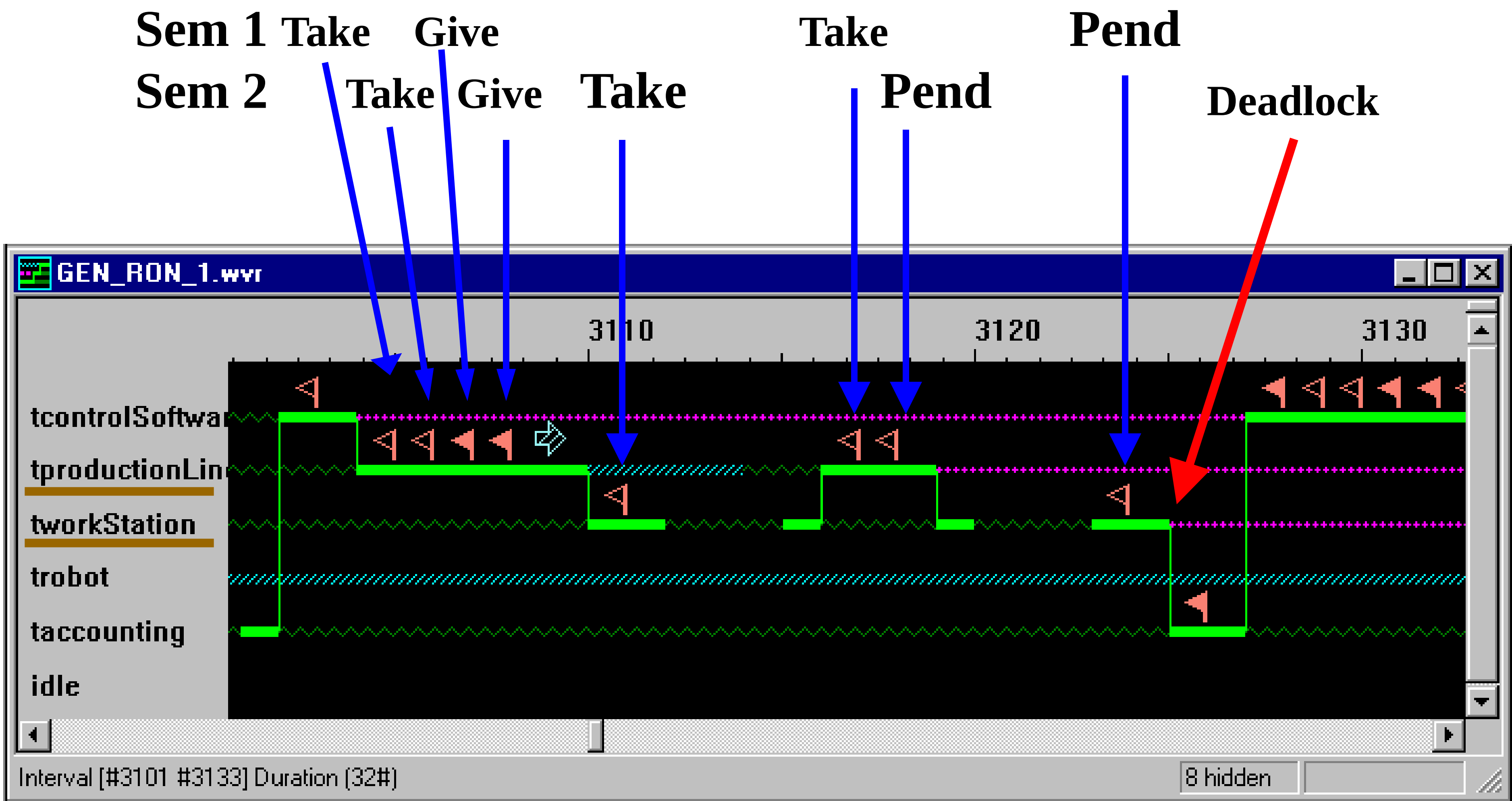
Example 2: Priority Inheritance

Example: inverSemId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);

Semaphore: Take Pend Give Give **Priority Inheritance**



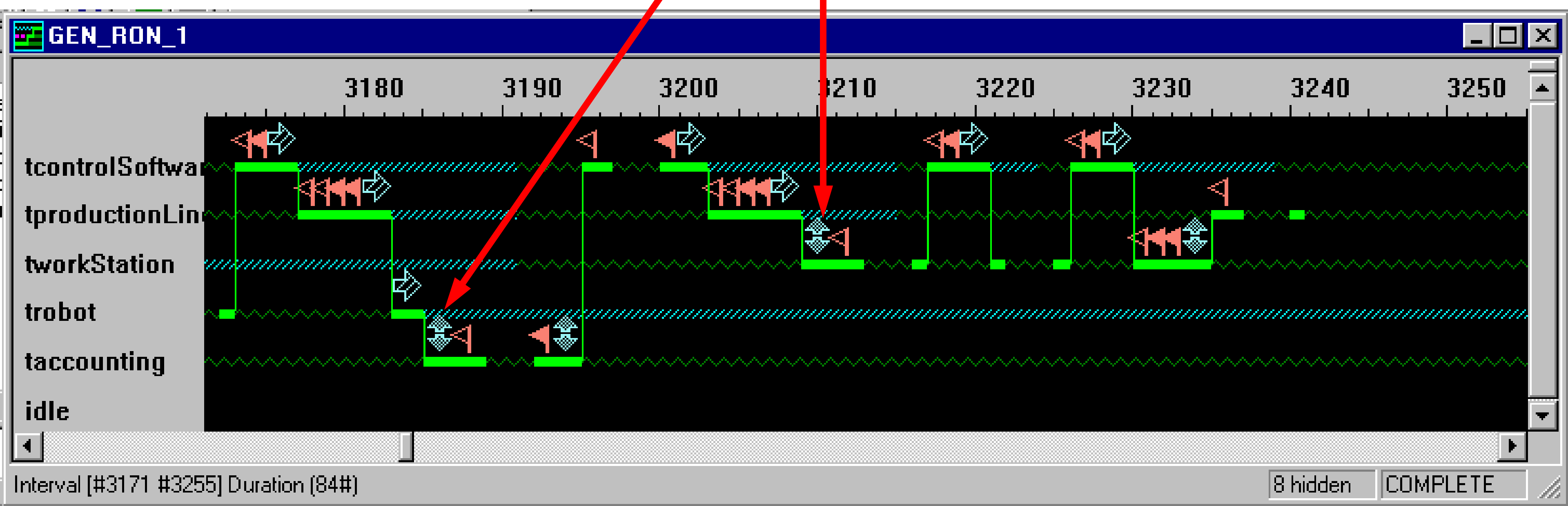
Priority Inheritance and Deadlock



Avoiding Deadlock: Emulating IPCP

```
void workStation(void) {
   taskPrioritySet(tidWs,MID1); /* IPCP dynamic priority */
   semTake(dlock2SemId, WAIT_FOREVER);   ...
   semTake(dlock1SemId, WAIT_FOREVER);   ...
   taskPrioritySet(tidWs,MID2); /* IPCP dynamic priority reset to original */  …}
```

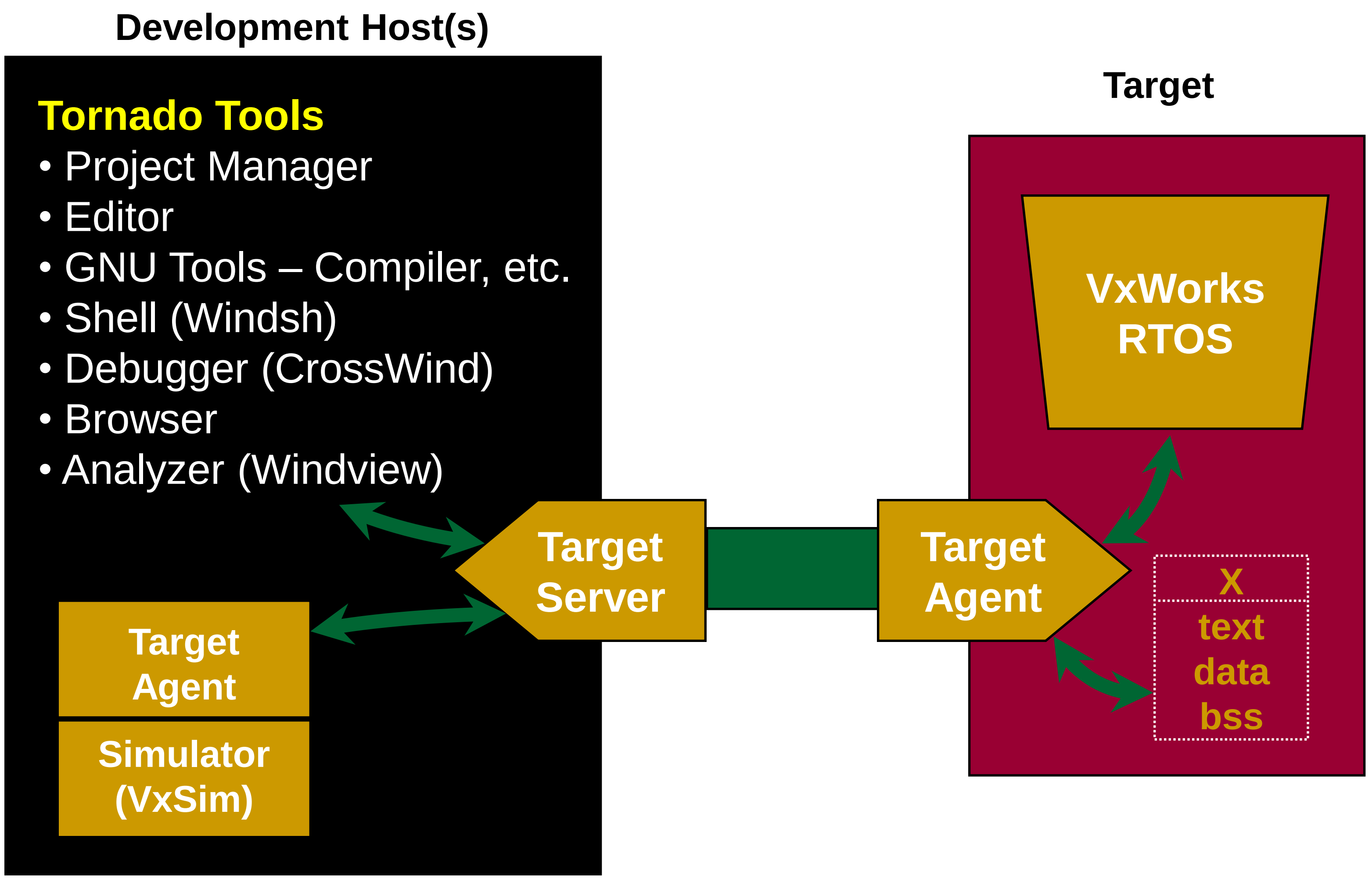
Priority Set



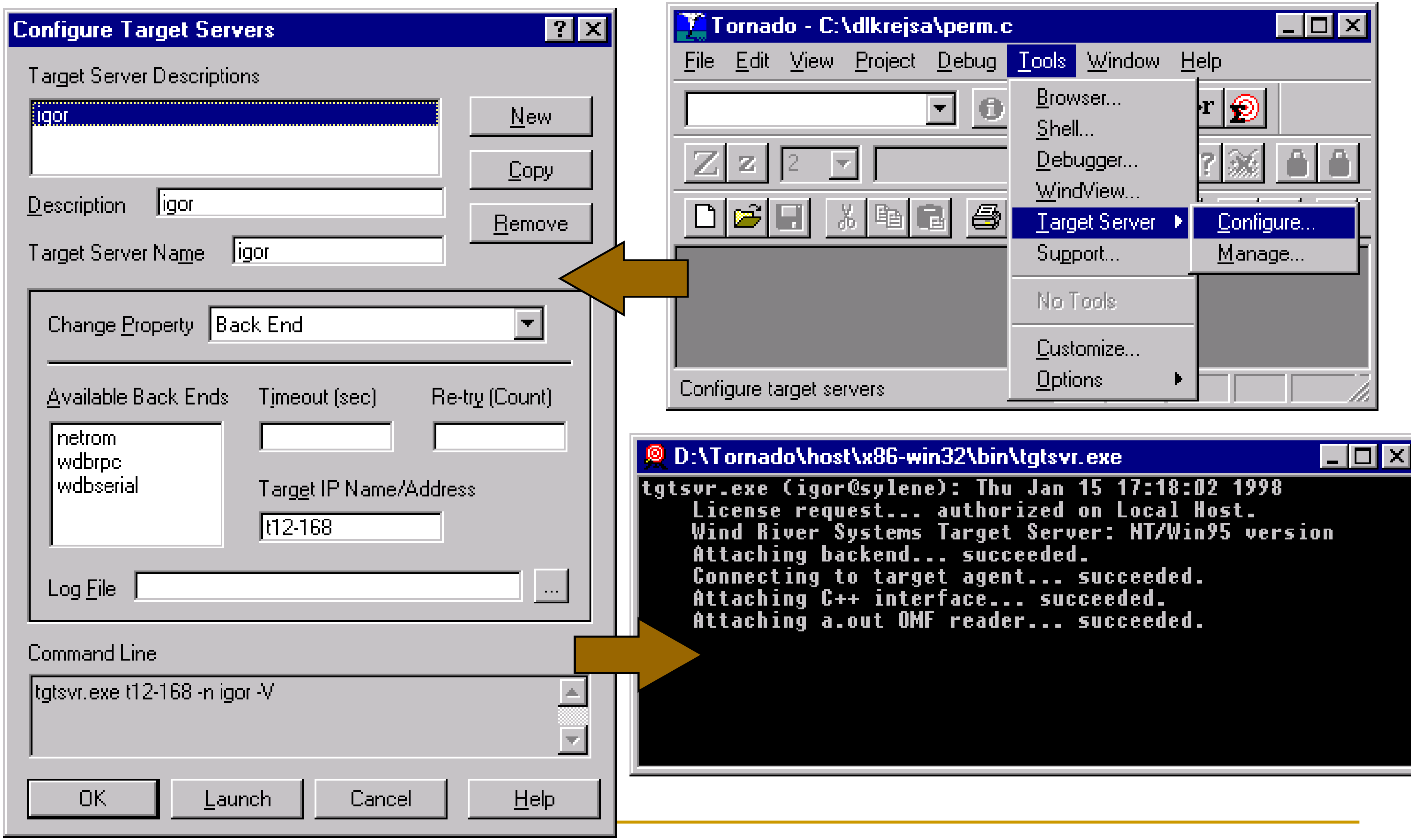
Priority Ceiling Protocols

- Assumptions
  - Fixed priorities
  - Set of resources to be accessed is fixed in advance
- Advantages
  - No deadlocks
  - No transitive blocking
  - A higher priority task can be blocked at most once by a lower priority task

Tornado Development Tools



Launching a Target Server

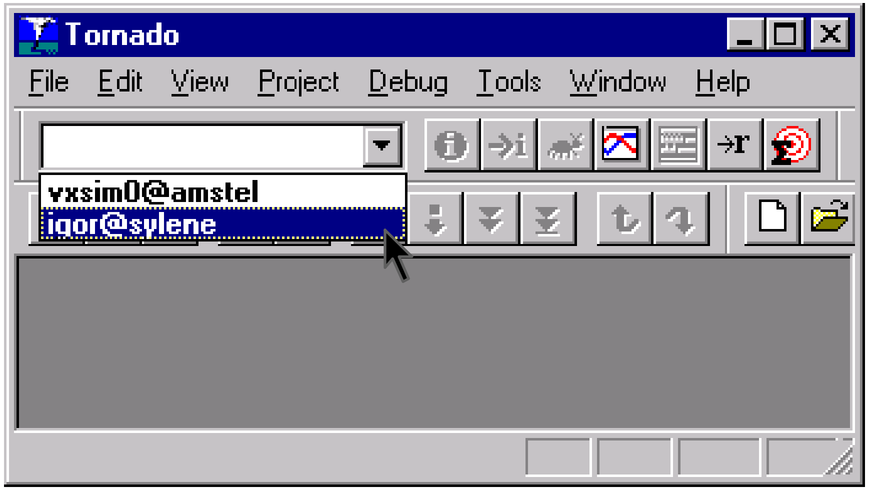


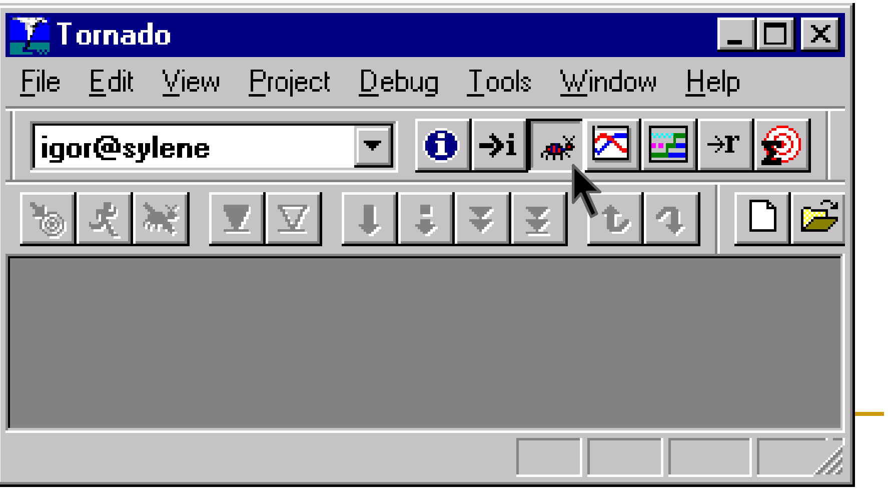
Starting Tornado Tools

1. Select target server from registry drop-down list.

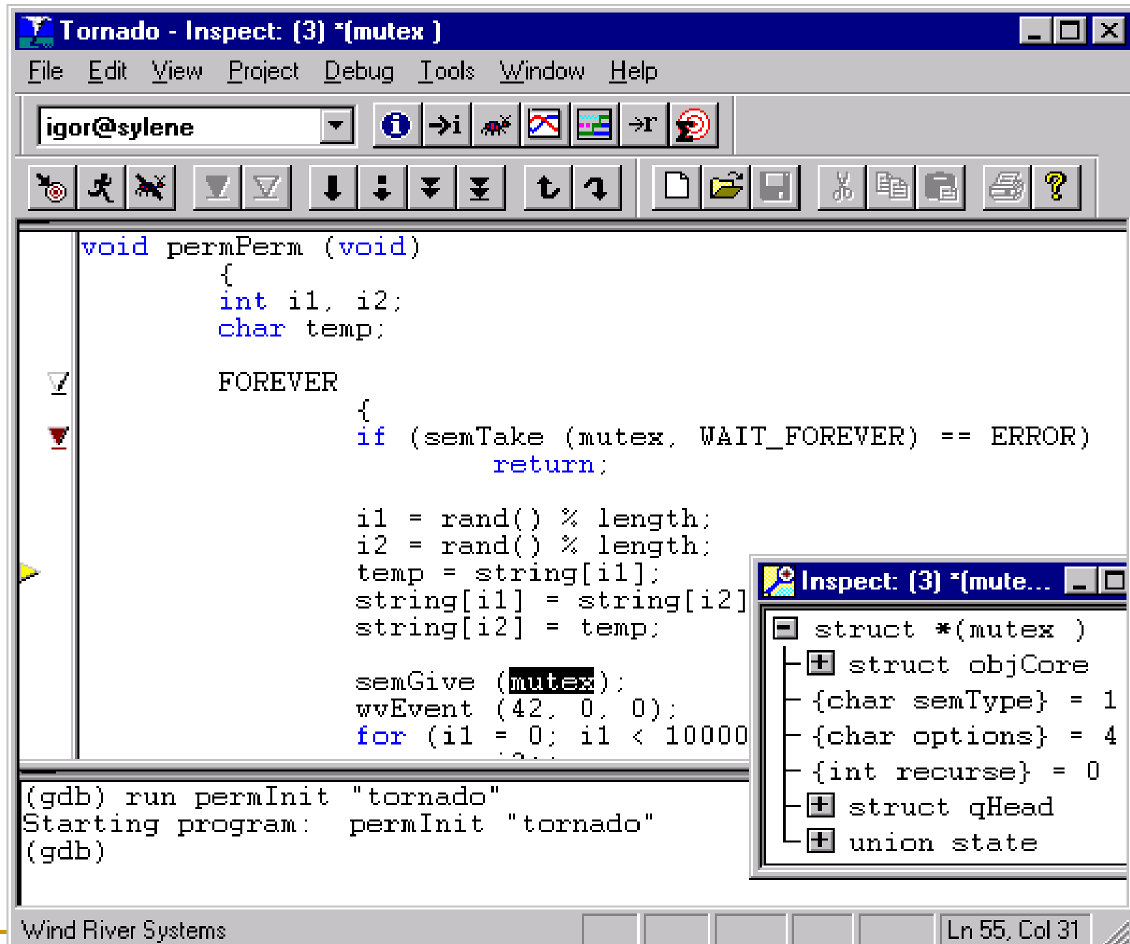
2. Click on tool icon in launch toolbar.

3. Select de-”bug”.





CrossWind Debugger

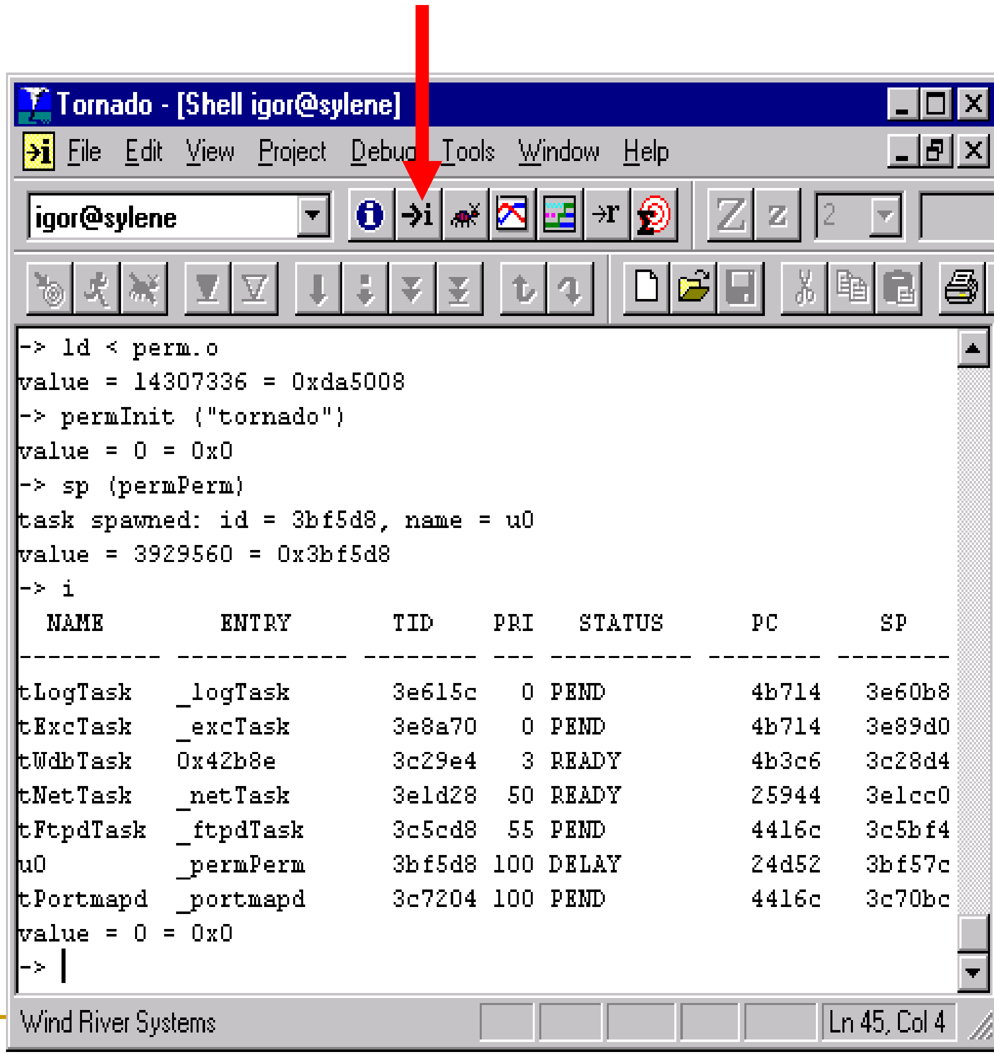


A graphical, source-level debugger built on the GNU debugger GDB.

Provides task level and system level debugging.

Use either the graphical or the command line interface.

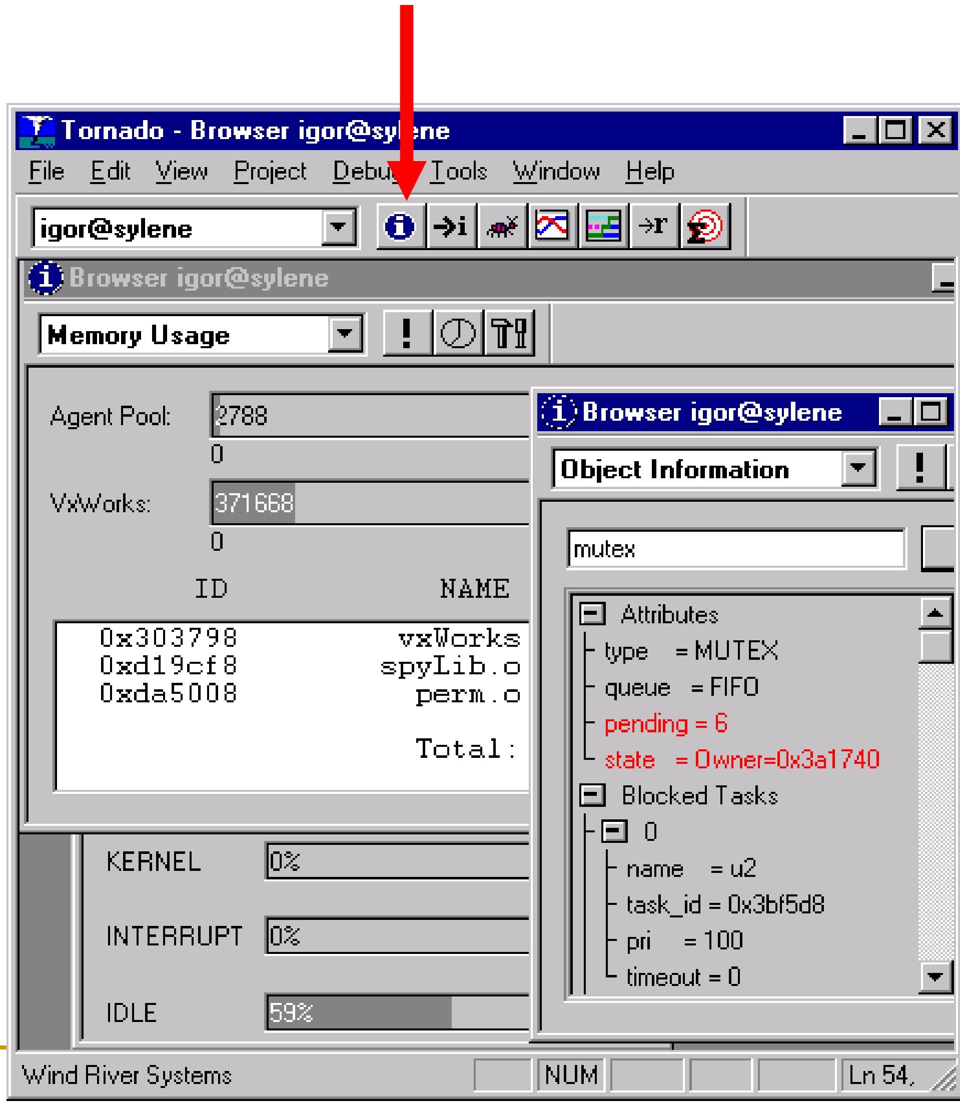
WindSh - The Tornado Shell



A C expression interpreter and an associated Tcl interpreter.

Download code to the target; spawn tasks to execute functions; modify existing variables or create new ones.

The Browser



Graphical tool which lets you monitor the state of the target, and display information on particular VxWorks system objects.

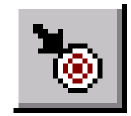
Information displayed may be updated on demand ! or periodically .

Downloading Object Modules

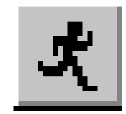
- Object modules may be downloaded dynamically and linked with modules already present on the target.
- To download from the shell:
  - → ld < myProg.o
- Debug => Download also loads debugging information used by CrossWind.

Debugging Tasks

- To download additional modules:

  Debug => Download or  Toolbar button

- To create a new task to run a loaded function:

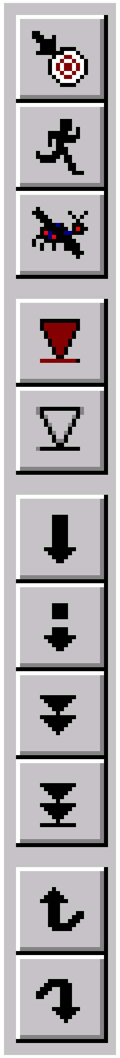
  Debug => Run or  Toolbar button

- To debug an already running task:

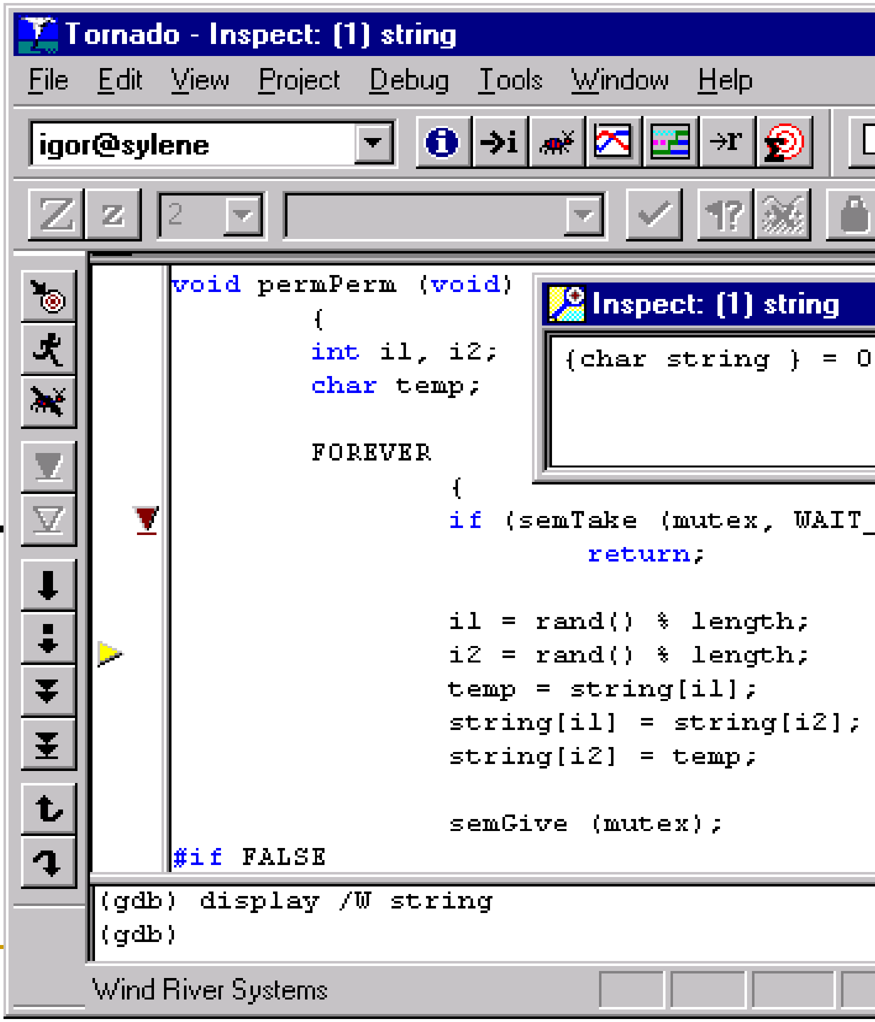
  Debug => Attach

- To debug multiple tasks independently, you may start multiple Tornado sessions.

CrossWind Debugger GUI

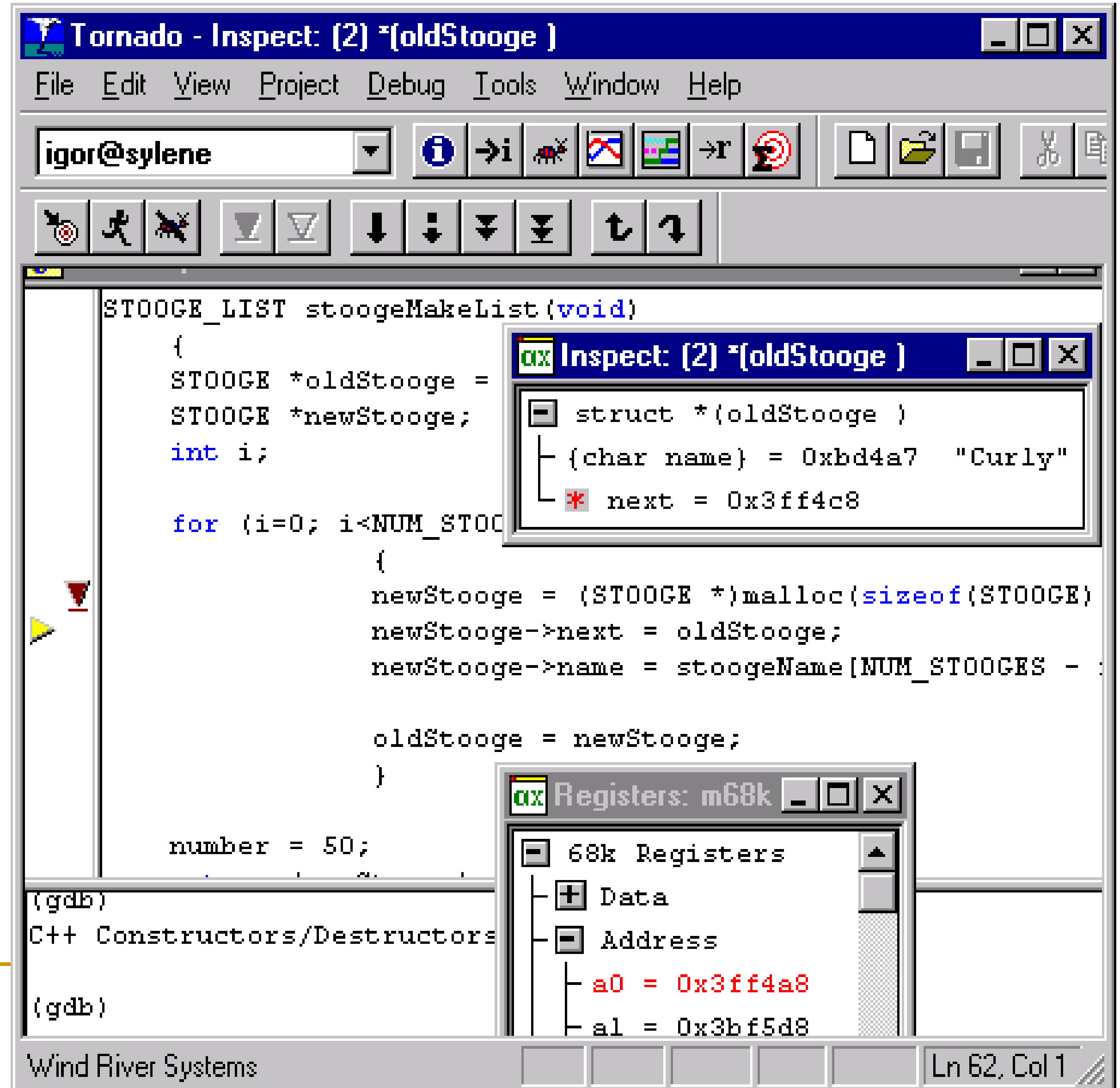


Download object module.
Spawn task to run function.
Stop debugging.
Toggle breakpoint.
Temporary breakpoint.
Step.
Next (step over function call).
Continue.
Finish current subroutine.
Up one stack frame.
Down one stack frame.



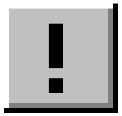
CrossWind Debugger Displays

- Application expressions
- Current local variables
- CPU registers
- Memory dump
- Stack call chain
- Displays are updated when control returns to the debugger.

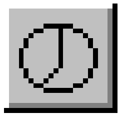


Browser Information

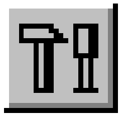
- Target Information
- Memory Usage (tools / application)
- Module Information (segments & symbols)
- Object Information (particular system objects)
- Spy Chart (CPU utilization)
- Stack Check
- Tasks



Update Now



Update Periodically

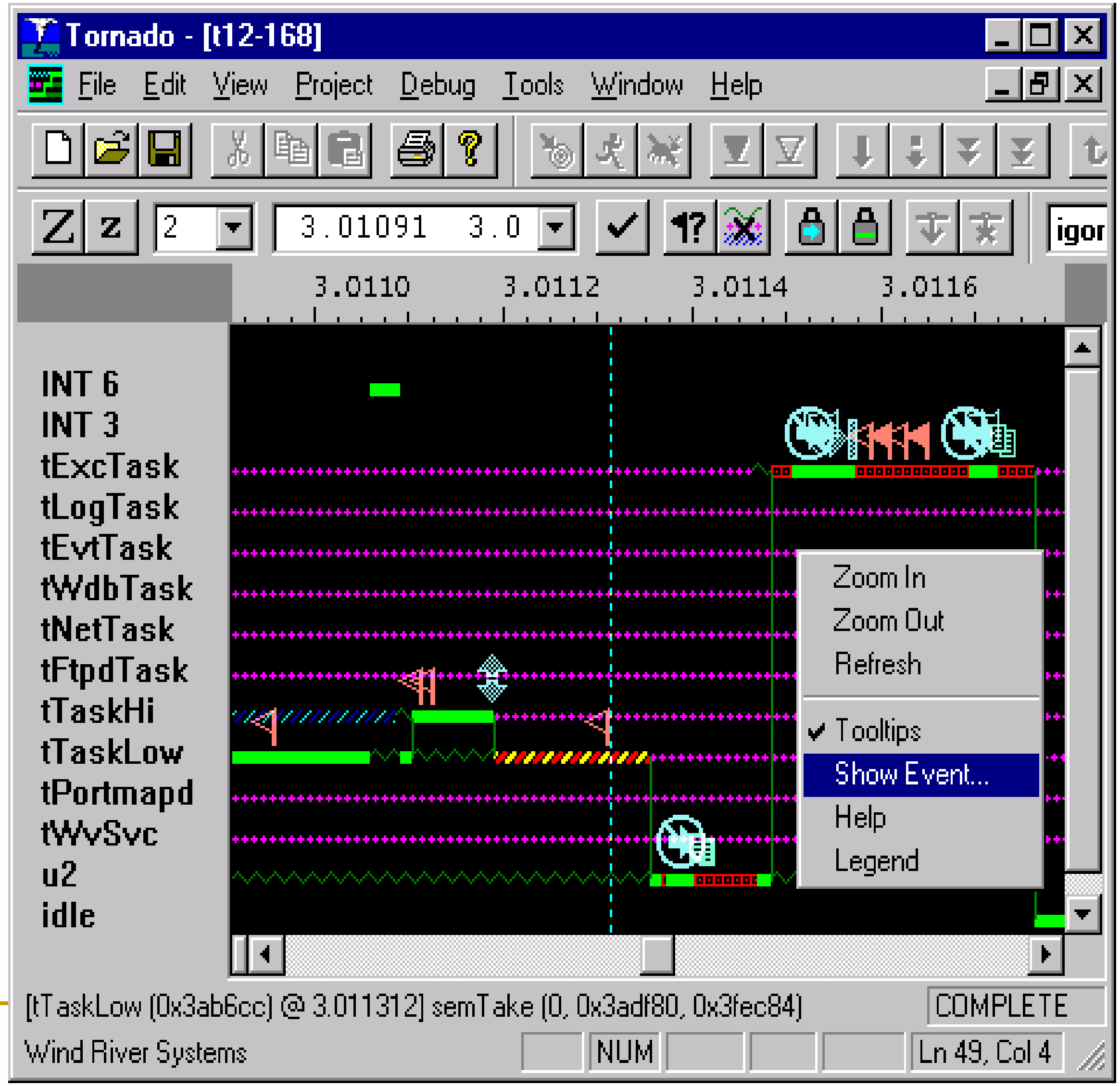


Configure Browser

WindView (Post-Processing Analysis)

WindView instruments the VxWorks kernel to record information on system (or user) events as they occur. If a high resolution timer (≈ 1µs) is available, events are assigned a timestamp.

This WindView graph illustrates a deadlock between *tTaskHi* and *tTaskLow*, and also shows task u2 exiting (with help from the task *tExcTask*).



Summary

- Read Ch. 8.
- Read Sha’s paper on priority inheritance protocols.