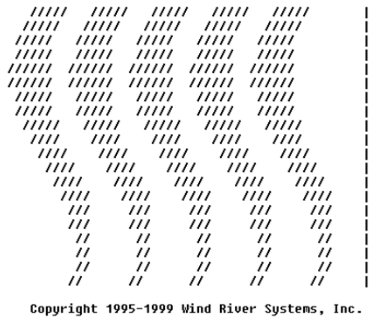
Revision 12.9 2011: added «Condition Variable» synchronization primitive

UNIVERSITY OF OSLO

FYS 4220 – 2011 / #3

Real Time and Embedded Data Systems and Computing

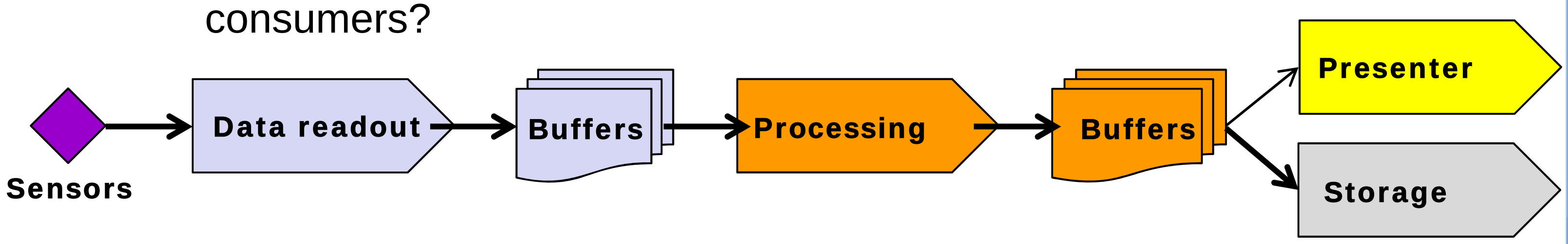
Process-Process Synchronization and Communication based on Shared Variables


```
T O R N A D O

Development System

Host Based Shell

Version 2.0.2

Copyright 1995-1999 Wind River Systems, Inc.
```

T.B. Skaali, Department of Physics, University of Oslo)

A kickstart: data flow through a basic DAQ system

- The figure shows schematically the data flow in a baseline DAQ system with three stages. Keep in mind the signal processing chain from Lecture 1. Let us here focus on the communication.
  - Data are exchanged via buffers. Each buffer has one *producer* and and in this case one or two *consumer* process(es)
    - The buffer comprises buffer information (address pointers, used space, free space, etc) and data
  - Synchronization is needed between the stages
  - A producer can only store data if there is free buffer space
    - What if the buffer is full?
  - A consumer can only fetch data if there is something in the buffer
    - What to do if the buffer is empty?
    - When can an input buffer be released when there are several consumers?



Process-process interaction

- Synchronization : no data transferred
- Messages : transfer of data but also a mean of synchronization: waiting process can not continue until sender has sent data
- Shared memory : as in tightly coupled systems
  - Note however that a shared memory does not necessarily imply that it is a local memory, in a cluster topology it may be located anywhere. More about this in a later lecture.
- In this lecture we will focus on shared variable based communication
  - As the name indicates, a ”shared variable” can be accessed from several processes. In a ”flat memory” system, i.e. no memory protection (vxWorks), any process can access any part of the local memory without going through a system call.
  - However, shared variables (semaphores) is only accessed through (VxWorks) system calls
  - In a memory protected system, shared memory areas can be mapped from user space through a system call

Borrowed some phrases from B&W for the lecture

Main Interprocess Communication methods (ref. Wikipedia)

- File - Most OS's
- Signal - Most OS's, some systems, such as Windows, only implement signals in the C run-time library and do not actually provide support for their use as an IPC technique.
- Socket - Most OS's
- Message queue - Most OS's
- Pipe / Named Pipe – All POSIX, Windows
- Semaphore – All POSIX, Windows
- Shared memory – All POSIX, Windows
- Message passing - Used in MPI paradigm and others
- Memory-mapped file - All POSIX systems, Windows. This technique may carry race condition risk if a temporary file is used

- On a higher level, there are many API's (Application Programmers Interface) and hardware interconnects for implementing IPCs

Mutual exclusion and critical sections

- The parts of processes that access shared variables must be executed indivisibly with respect to each other
- These parts are called critical sections
- The required protection is called mutual exclusion
- So if one uses shared variables for synchronization, how can they be accessed from a critical section, because one would use the same variables to create a critical section??

Accessing shared variables can go *vyer rwnog* !

```
•   /* (un)critical region demo case, B. Skaali, Jan. 2002 */

•   #include   "vxWorks.h"
•   #include   "sysLib.h"
•   #include   "time.h"
•   #include   "kernelLib.h"
•   #include   "taskLib.h"
•   #include   "tickLib.h"
•   #include   "stdio.h"

•   #define                     LOOPS          50000000
•   /* reduce LOOPS for slow MC68030 target */
•   #define                     INC            1
•   /* relative priorities */
•   #define                     P_PRI          200
•   #define                     M_PRI          200

•   ULONG   tickstart;
•   double  x;

•   /* plusx task */
•   int       plusx()
•   {
•     double i;
•     for (i=0; i<LOOPS; i++) {
•       x = x + INC;
•       x = x + INC;
•       x = x + INC;
•     }
•    printf("\n");
•    printf("- plusx exit = %ld\n", (long int)x);
•    printf("- plusx exit ticks = %ld\n", tickGet()-tickstart);
•    return (OK);
•   }
```

Accessing shared variables can go *vyer rwnog* !

```
/* minusx task */
int         minusx()
{
  double i;
  for (i=0; i<LOOPS; i++) {
    x = x - INC;
    x = x - INC;
    x = x - INC;
  }
 printf("\n");
 printf("- minusx exit= %ld\n", (long int)x);
 printf("- minusx exit ticks = %ld\n", tickGet()-tickstart);
 return (OK);
}

/* start here to run the demo */
void        plusminus()
{
  int Pid, Mid;
/* enable or disable Wind round-robin*/
  kernelTimeSlice (1);
/* launch the intelligent stuff */
  x = 0;
  tickSet(0);
  tickstart = tickGet();
  Pid = taskSpawn ("tP",P_PRI,0,1000,(FUNCPTR)plusx,0,0,0,0,0,0,0,0,0,0);
  Mid = taskSpawn ("tM",M_PRI,0,1000,(FUNCPTR)minusx,0,0,0,0,0,0,0,0,0,0);
}

/* show value */
void        showx()
{
  printf("current x = %ld\n", (long int)x);
}
```

Accessing shared variables goes rwnog

- Start from the shell
  - **-> plusminus**
  - **value = 78545560 = 0x4ae8298**
  - **->**
- Output:
  - - plusx exit = 46959291
  - - plusx exit ticks = 138

  - - minusx exit= 46051968
  - - minusx exit ticks = 138
- From showx():
  - **-> showx**
  - **current x = 46051968**
  - **value = 21 = 0x15**

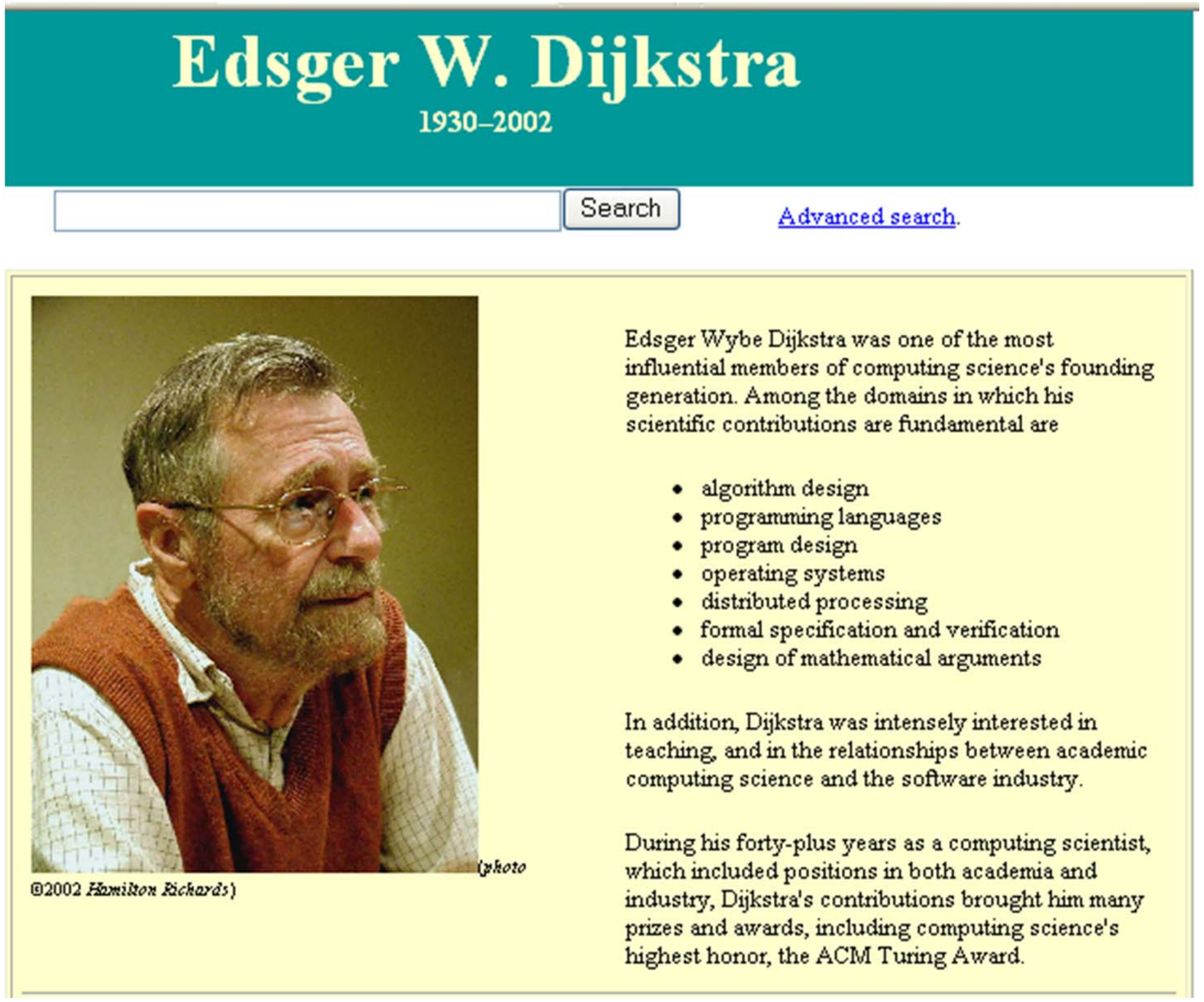
Busy waiting

- One way to implement synchronisation is to have processes set and check shared variables that are acting as flags
- This approach can work well for condition synchronisation but no simple method for mutual exclusion exists
  - However, if more than one process can change the value of the flag in a non-atomic action, then one is back to square one!
- Busy wait algorithms are in general inefficient; they involve processes using up processing cycles when they cannot perform useful work
- Busy waiting can create a livelock!

Atomic actions

- An atomic action is a sequence of instructions which are executed as an entity
- Changing a condition flag implies that it is incremented or decremented
- If this operation is not carried out by a single machine (assembly) instruction it is not an atomic action if the sequence can be interrupted
  - Note however processors have multi-stage atomic instructions such as “test-and-set", "fetch-and-add“, "compare-and-swap“, etc
  - Example: http://www.ibm.com/developerworks/library/pa-atom
- So, is synchronization by means of shared variables possible, or is it a contradiction in terms?
- It is, but complicated, and the first solution was given by E.W. Dijkstra

The birth of the Semaphore concept


Edsger W. Dijkstra

1930–2002

(photo ©2002 *Hamilton Richards*)

Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are

- algorithm design
- programming languages
- program design
- operating systems
- distributed processing
- formal specification and verification
- design of mathematical arguments

In addition, Dijkstra was intensely interested in teaching, and in the relationships between academic computing science and the software industry.

During his forty-plus years as a computing scientist, which included positions in both academia and industry, Dijkstra's contributions brought him many prizes and awards, including computing science's highest honor, the ACM Turing Award.

E.W. Dijkstra: ”Computer Science is no more about computers than astronomy is about telescopes”.

Quotation #788 from Michael Moncur’s (Cynical) Quotations

Dijkstra’s Semaphores

- A semaphore is a non-negative integer variable that apart from initialization can only be acted upon by two procedures P (or WAIT) and V (or SIGNAL)
  - The canonical names P and V come from the initials of Dutch words. V stands for *verhogen* (“increase”), and P stands for *probeer* (“try”)
- WAIT(S)   If the value of S > 0 then decrement its value by one; otherwise **delay** the process until S > 0 (and then decrement its value).
- SIGNAL(S)   Increment the value of S by one. If a process is waiting for S, **dispatch** the process
- WAIT and SIGNAL are atomic (indivisible). Two processes both executing WAIT operations on the same semaphore cannot interfere with each other and cannot fail during the execution of a semaphore operation

Synchronization

```
(Some language)  var sem : semaphore (* init 0 *)
```

```
process P1;
  statement X;
  wait (sem)
  statement Y;
end P1;
```

```
process P2;
  statement A;
  signal (sem)
  statement B;
end  P2;
```

In what order will the statements execute?

Mutual exclusion

```
(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)
```

```
process P1;
  statement X
  wait (mutex);
    statement Y
  signal (mutex);
  statement Z
end P1;
```

```
process P2;
  statement A;
  wait (mutex);
    statement B;
  signal (mutex);
  statement C;
end P2;
```

In what order will the statements execute?

Problems with semaphores

- Semaphore are an elegant low-level synchronisation primitive, however, their use is error-prone
- If a semaphore is omitted or misplaced, the entire program will collapse. Mutual exclusion may not be assured and deadlock may appear just when the software is dealing with a rare but critical event
- A more structured synchronisation primitive is required
- No high-level concurrent programming language relies entirely on semaphores; they are important historically but are arguably not adequate for the real-time domain (well, I am not sure that I will fully buy this statement …)

Deadlock (or the deadly embrace)

- Two processes are deadlocked if each is holding a resource while waiting for a resource held by the other

```
type Sem is ...;
X : Sem := 1; Y : Sem := 1;
```

```
proc P1;
begin
     ...
     Wait(X);
     Wait(Y);
     ...
end P1;
```

```
process P2;
begin
     ...
     Wait(Y);
     Wait(X);
     ...
end P2;
```

Deadlock will occur when?

Binary and quantity semaphores

- A general semaphore is a non-negative integer; its value can rise to any supported positive number
- A binary semaphore only takes the value 0 and 1; the signalling of a semaphore which has the value 1 has no effect - the semaphore retains the value 1
- A general semaphore can be implemented by two binary semaphores and an integer. Try it!
- With a quantity semaphore the amount to be decremented by WAIT (and incremented by SIGNAL) is given as a parameter; e.g. WAIT (S, i)

UNIVERSITY OF OSLO

VxWorks Mutual Exclusion I

- While a shared address space simplifies exchange of data, interlocking access to memory is crucial to avoid contention.
- Method 1:
  - Interrupt locks with system calls intLock() and intUnLock()
    - Very powerful, no interference when interrupt disabled!
    - Latency problem, can not respond to interrupt during the lock

```
funcA ()
{
 int lock = intLock();
 .
 .   critical region that cannot be interrupted
 .
 intUnlock (lock);
}
```

VxWorks Mutual Exclusion II

- Method 2:
  - Preemptive lock with taskLock() and taskUnLock
    - Disabling preemption is somewhat less restrictive
    - Interrupt service routines (ISR) can continue to execute
    - However, tasks of higher priority can not execute

```
funcA ()
{
 taskLock ();
 .
 .   critical region that cannot be interrupted
 .
 taskUnlock ();
}
```

VxWorks Mutual Exclusion III

- Method 3 – Semaphores
  - For *mutual exclusion*, semaphores interlock access to shared resources
  - For *synchronization*, semaphores coordinate a task's execution with external events
  - Wind semaphores:
    - ***binary*** - the fastest, most general-purpose semaphore. Optimized for synchronization or *mutual exclusion*
    - ***mutual exclusion*** - a special binary semaphore optimized for problems inherent in mutual exclusion: priority inheritance, deletion safety, and recursion
    - ***counting*** - like the binary semaphore, but keeps track of the number of times a semaphore is given. Optimized for guarding multiple instances of a resource
  - Vxworks also provides POSIX semaphores

VxWorks ”Wind” semaphore library

- ***semBCreate*( )** Allocate and initialize a binary semaphore.
- ***semMCreate*( )** Allocate and initialize a mutual-exclusion semaphore.
- ***semCCreate*( )** Allocate and initialize a counting semaphore.
- ***semDelete*( )** Terminate and free a semaphore.
- ***semTake*( )** Take a semaphore.
- ***semGive*( )** Give a semaphore.
- ***semFlush*( )** Unblock all tasks that are waiting for a semaphore.
- The ***semBCreate*( )**, ***semMCreate*( )**, and ***semCCreate*( )** routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by the other semaphore-control routines. When a semaphore is created, the queue type is specified. Tasks pending on a semaphore can be queued in priority order (**SEM_Q_PRIORITY**) or in first-in first-out order (**SEM_Q_FIFO**).
  - **WARNING:** The ***semDelete*( )** call terminates a semaphore and deallocates any associated memory. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore that another task still requires. Do not delete a semaphore unless the same task first succeeds in taking it.

Accessing shared variables goes OK

```
•   /* binary semaphore critical region demo , B. Skaali, Sep. 2007 */

•   #include            "vxWorks.h"
•   #include            "sysLib.h"
•   #include            "time.h"
•   #include            "kernelLib.h"
•   #include            "taskLib.h"
•   #include            "tickLib.h"
•   #include            "stdio.h"

•   #define                         LOOPS     500000
•   /* reduce LOOPS for slow MC68030 target */
•   #define                         INC       1
•   /* relative priorities */
•   #define                         P_PRI     200
•   #define                         M_PRI     200

•   /* mutex */
•   SEM_ID                          semMutex;

•   ULONG   tickstart;
•   double x;
```

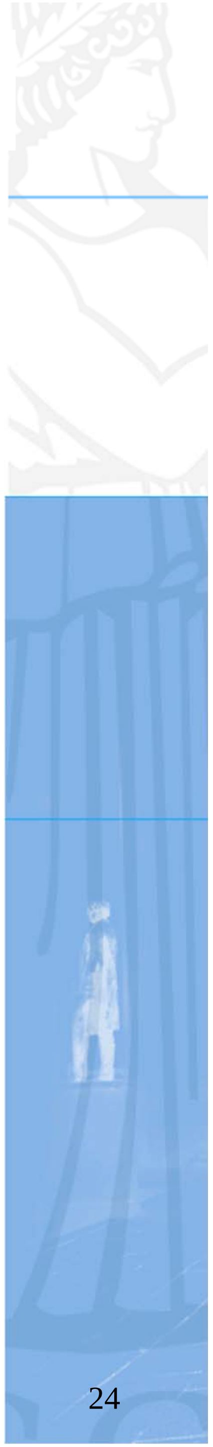
UNIVERSITY OF OSLO

Accessing shared variables goes OK

```
/* plusx */
int            plusx()
{
  double i;
  for (i=0; i<LOOPS; i++) {
    semTake (semMutex, WAIT_FOREVER);
    x = x + INC;
    x = x + INC;
    x = x + INC;
    semGive (semMutex);
  }
 printf("\n");
 printf("- plusx exit = %ld\n", (long int)x);
 printf("- plusx exit ticks = %ld\n", tickGet()-tickstart);
 return (OK);
}

/* minusx */
int            minusx()
{
  double i;
  for (i=0; i<LOOPS; i++) {
    semTake (semMutex, WAIT_FOREVER);
    x = x - INC;
    x = x - INC;
    x = x - INC;
    semGive (semMutex);
  }
 printf("\n");
 printf("- minusx exit= %ld\n", (long int)x);
 printf("- minusx exit ticks = %ld\n", tickGet()-tickstart);
 return (OK);
}
```

Accessing shared variables goes OK

```
•       /* run the demo */
•       STATUS plusminus()
•       {
•         int Pid, Mid;

•       /* enable or disable Wind round-robin*/
•         kernelTimeSlice (1);

•       /* Create a binary semaphore that is initially full. Tasks *
•        * blocked on semaphore wait in priority order.            */
•          semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
•          if (semMutex == NULL) return(-1);

•       /* launch the intelligent stuff */
•         x = 0;
•         tickSet(0);
•         tickstart = tickGet();
•         Pid = taskSpawn ("tP",P_PRI,0,1000,(FUNCPTR)plusx,0,0,0,0,0,0,0,0,0,0);
•         Mid = taskSpawn ("tM",M_PRI,0,1000,(FUNCPTR)minusx,0,0,0,0,0,0,0,0,0,0);
•         return (OK);
•       }
•       /* what is the status of semMutex after plusx and minusx have run? *
•        * use shell command semShow(semMutex,1) */
•
•       /* show value */
•       void        showx()
•       {
•         printf("current x = %ld\n", (long int)x);
•       }
```



Accessing shared variables goes OK

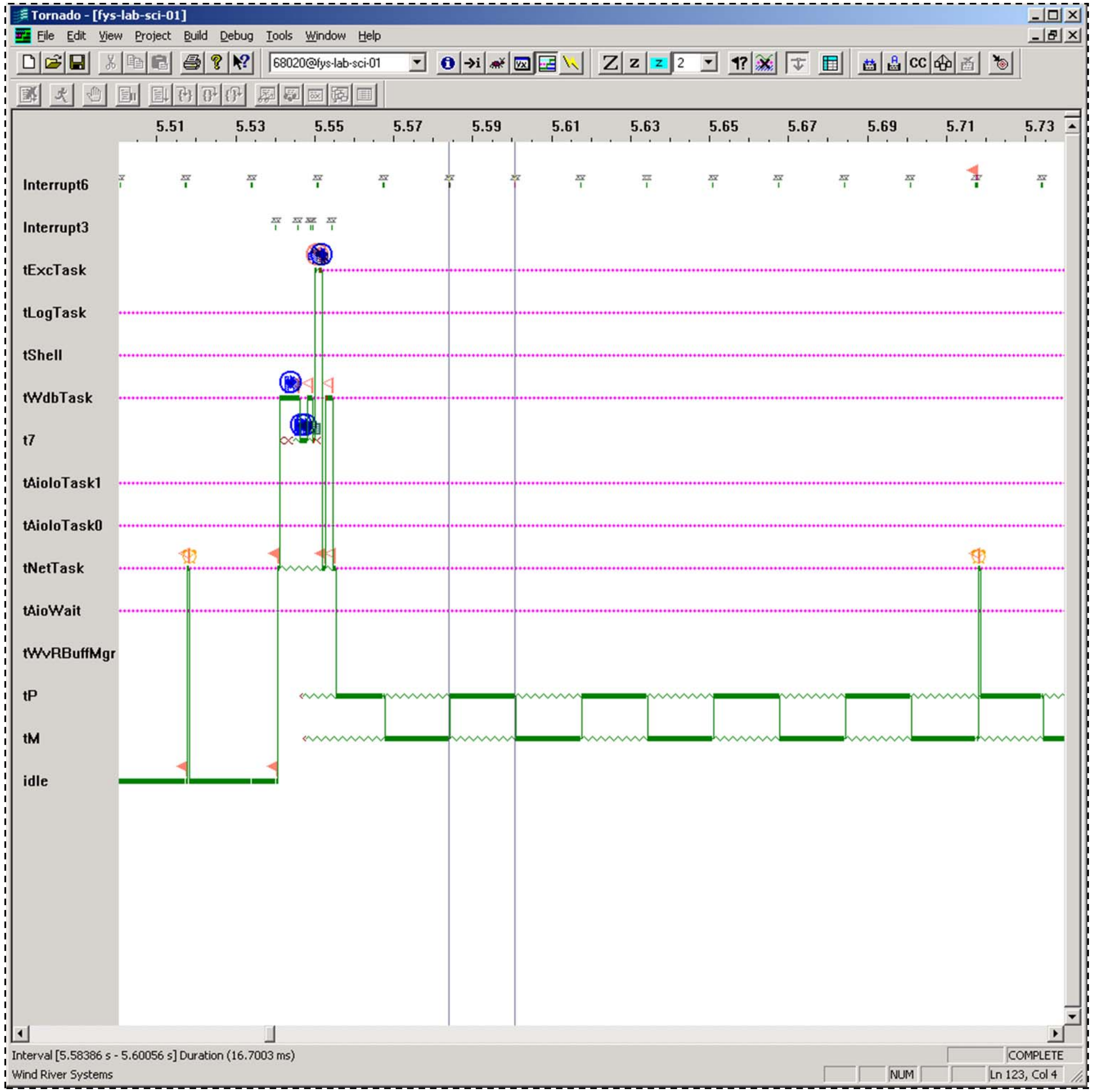
- **-> plusminus**
- **value = 0 = 0x0**

- **-> showx**
- **current x = 0**
- **value = 14 = 0xe**

- **-> semShow(semMutex,1)**
- **Semaphore Id   : 0x4b2dce0**
- **Semapho re Type : BINARY**
- **Task Queueing   : PRIORITY**
- **Pended Tasks   : 0**
- **State   : FULL**

- - plusx exit = 1500000
- - plusx exit ticks = 67

- - minusx exit= 0
- - minusx exit ticks = 129



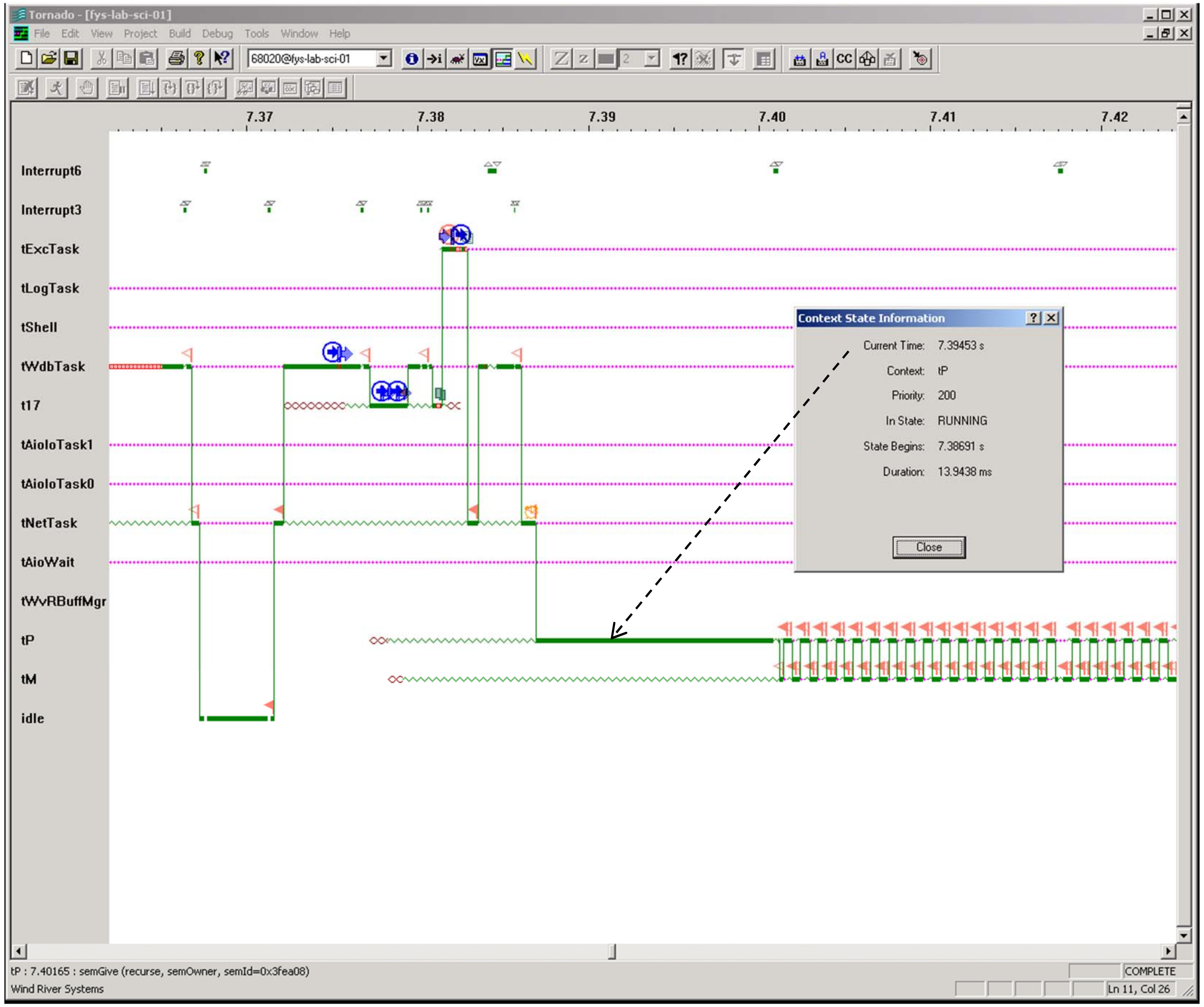
Interval [5.58386 s - 5.60056 s] Duration (16.7003 ms)

Wind River Systems

UNIVERSITY OF OSLO

Without mutual exclusion or interaction between plusx() and minusx().

After tP and tM have been dispatched from process interactive process t7 through tExcTask they execute on a round-robin basis with time slice 16.67ms, which is the period of the Real-Time clock

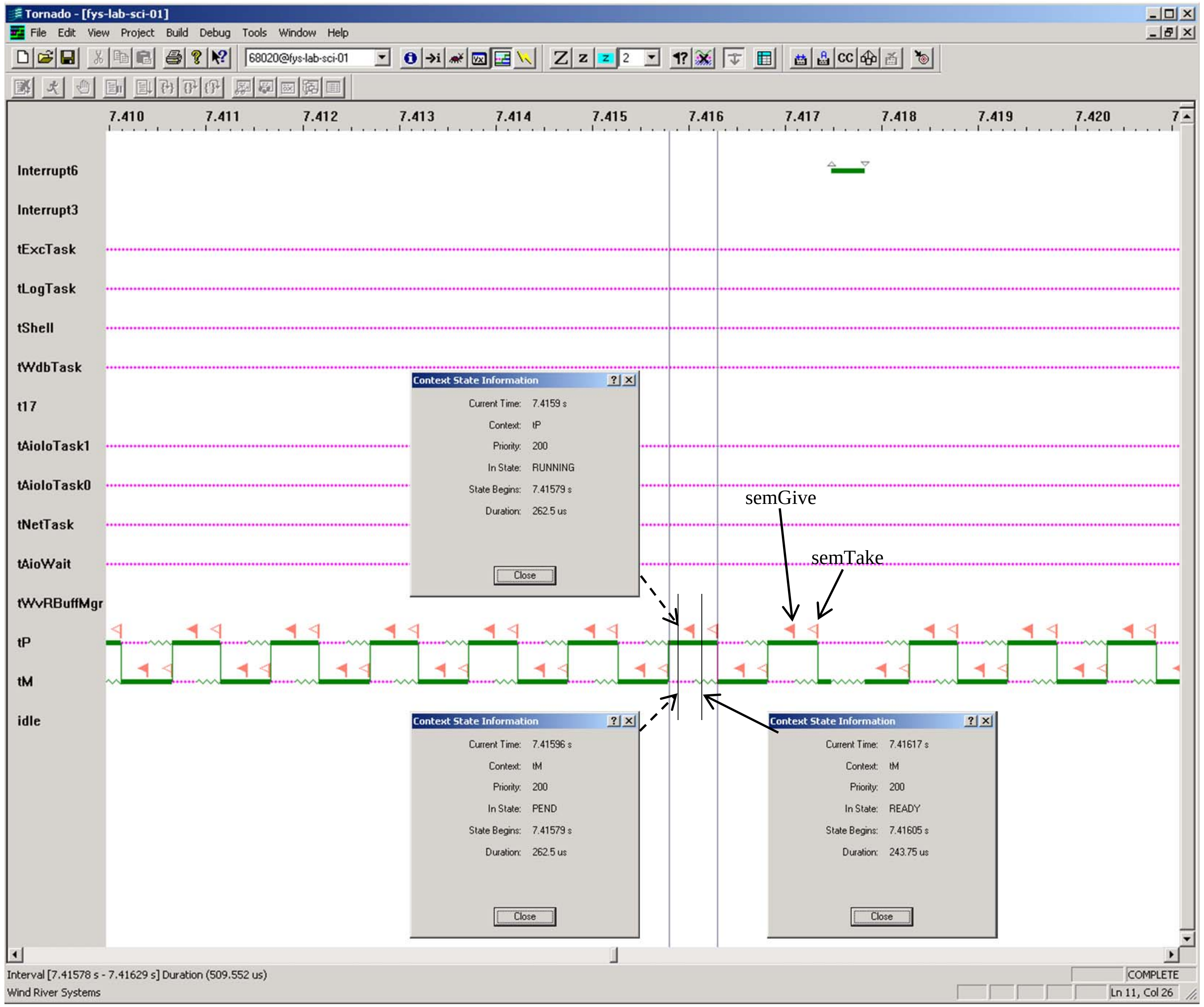


Context State Information

Current Time:	7.39453 s
Context:	tP
Priority:	200
In State:	RUNNING
State Begins:	7.38691 s
Duration:	13.9438 ms

```
tP : 7.40165 : semGive (recurse, semOwner, semId=0x3fea08)
```

With mutual exclusion between plusx() and minusx(). Semaphore operations are shown by flags.



Context State Information

Current Time:	7.4159 s
Context:	tP
Priority:	200
In State:	RUNNING
State Begins:	7.41579 s
Duration:	262.5 us

Close

Context State Information

Current Time:	7.41596 s
Context:	tM
Priority:	200
In State:	PEND
State Begins:	7.41579 s
Duration:	262.5 us

Close

Context State Information

Current Time:	7.41617 s
Context:	tM
Priority:	200
In State:	READY
State Begins:	7.41605 s
Duration:	243.75 us

Close

Interval [7.41578 s - 7.41629 s] Duration (509.552 us)
Wind River Systems
COMPLETE
Ln 11, Col 26

With mutual exclusion between plusx() and minusx(). Semaphore operations are shown by flags. Note that timeslice is now ~500µs!

VxWorks POSIX semaphores

- POSIX defines both *named* and *unnamed* semaphores, which have the same properties, but use slightly different interfaces. The POSIX semaphore library provides routines for creating, opening, and destroying both named and unnamed semaphores
- The POSIX terms *wait* (or *lock*) and *post* (or *unlock*) correspond to the VxWorks terms *take* and *give*, respectively
- POSIX semaphores are *counting* semaphores; that is, they keep track of the number of times they are given
- The Wind semaphore mechanism is similar to that specified by POSIX, except that Wind semaphores offer additional features: priority inheritance, task-deletion safety, the ability for a single task to take a semaphore multiple times, ownership of mutual-exclusion semaphores, semaphore timeouts, and the choice of queuing mechanism. When these features are important, Wind semaphores are preferable.

VxWorks POSIX semaphore library

• ***semPxLibInit*( )**	Initialize the POSIX semaphore library (non-POSIX).
• ***sem_init*( )**	Initialize an unnamed semaphore.
• ***sem_destroy*( )**	Destroy an unnamed semaphore.
• ***sem_open*( )**	Initialize/open a named semaphore.
• ***sem_close*( )**	Close a named semaphore.
• ***sem_unlink*( )**	Remove a named semaphore.
• ***sem_wait*( )**	Lock a semaphore.
• ***sem_trywait*( )**	Lock a semaphore only if it is not already locked.
• ***sem_post*( )**	Unlock a semaphore.
• ***sem_getvalue*( )**	Get the value of a semaphore.

- **WARNING:** The ***sem_destroy*( )** call terminates an unnamed semaphore and deallocates any associated memory; the combination of ***sem_close*( )** and ***sem_unlink*( )** has the same effect for named semaphores. Take care when deleting semaphores, particularly mutual exclusion semaphores, to avoid deleting a semaphore still required by another task. Do not delete a semaphore unless the deleting task first succeeds in locking that semaphore. (Likewise, for named semaphores, close semaphores only from the same task that opens them.)

POSIX mutexes I

- The POSIX thread library contains several synchronization constructs.
- The simplest of these is the mutex lock.
- A mutex, or mutex lock, is a special variable that can be either in the locked state or the unlocked state.
- If the mutex is locked, it has a distinguished thread that *holds* or *own* the mutex.
- If the mutex is unlocked, we say that the mutex is *free* or *available*.
- The mutex also has a queue of threads that are waiting to hold the mutex.
- POSIX does not require that this queue be accessed FIFO.
- A mutex is meant to be held for only a short period of time.
- More about POSIX mutexes and condition variables in a later lecture

The Barrier synchonization concept

- In parallell computing, a **barrier** is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
  - A barrier can be implemented by means of Wind semaphores. When all process have reached the barrier and are waiting for the semaphore, *semFlush*( ) will unblock all tasks that are waiting.
  - What about operations on the common variable `barrierCounter` !!??

```
/* Use a barrier for thread synchronization */
  semTake(barrierMutex, WAIT_FOREVER);

 if (++barrierCounter == 3)
 {
                    /* All threads have reached waypoint - Continue */
                    semFlush(barrierWait);
                    barrierCounter = 0;
                    semGive(barrierMutex);
  }
                    else
{
                    /* Wait for other threads */
                     semGive(barrierMutex);
                    semTake(barrierWait, WAIT_FOREVER);
  }
```

Condition Variables, Joining

- Condition Variables are a second kind of synchronization primitive (Mutexes being the first). They are useful when you have a thread that needs to wait for a certain condition to be true. In **pthreads**, there are three relevant procedures involving condition variables:
  - **pthread_cond_init(pthread_cond_t *cv);**
  - **pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);**
  - **pthread_cond_signal(pthread_cond_t *cv)**
- The first of these simply initializes a condition variable. The second two are related. **Pthread_cond_wait()** is called by a thread when it wants to block and wait for a condition to be true. It is assumed that the thread has locked the mutex indicated by the second parameter. The thread releases the mutex, and blocks until awakened by a **pthread_cond_signal()** call from another thread. When it is awakened, it waits until it can acquire the mutex, and once acquired, it returns from the **pthread_cond_wait()** call.
- **Pthread_cond_signal()** checks to see if there are any threads waiting on the specified condition variable. If not, then it simply returns. If there are threads waiting, then one is awakened. It is not specified whether the thread that calls **pthread_cond_signal()** should own the locked mutex specified by the **pthread_cond_wait()** call of the thread that it is waking up.
- Note, you should not assume anything about the order in which threads are awakened by **pthread_cond_signal()** calls. It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case. Program accordingly.

Ref. Jim Plank, Lecture notes: http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/CondVar/lecture.html

Condition Variables, Joining

- A simple example of using condition variables is in the program **barrier.c**. Here, we have 5 threads, and we want to make sure that they all synchronize at a particular point. Often this is called a ``barrier'', since all the threads stop at this barrier before proceeding. In **barrier.c** the number of threads waiting is held in the variable **ndone**, and if a thread reaches the barrier before **ndone** equals **NTHREADS**, it waits on the condition variable **ts->cv**. When the last thread reaches the barrier, it wakes all the others up using **pthread_cond_signal**. The output of **barrier.c** shows that they all block until the last thread reaches the barrier:
  - The barrier.c source is shown on next page

```
barrier
Thread 0 -- waiting for barrier
Thread 1 -- waiting for barrier
Thread 2 -- waiting for barrier
Thread 3 -- waiting for barrier
Thread 4 -- waiting for barrier
Thread 4 -- after barrier
Thread 0 -- after barrier
Thread 1 -- after barrier
Thread 2 -- after barrier
Thread 3 -- after barrier
done
```

Ref. Jim Plank, Lecture notes: http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/CondVar/lecture.html

```
#include "vxWorks.h"
#include "taskLib.h"
#include "sysLib.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
#include "time.h"
#include "tickLib.h"
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "logLib.h"

typedef struct {
    pthread_mutex_t *lock;
    pthread_cond_t *cv;
    int *ndone;
    int id;
} TStruct;

#define NTHREADS 5

void *barrier(void *arg)
{
    TStruct *ts;
    int i;
    ts = (TStruct *) arg;
    printf("Thread %d -- waiting for barrier\n", ts->id);
    pthread_mutex_lock(ts->lock);
    *ts->ndone = *ts->ndone + 1;
    if (*ts->ndone < NTHREADS) {
        pthread_cond_wait(ts->cv, ts->lock);
    } else {
       for (i = 1; i < NTHREADS; i++) pthread_cond_signal(ts-
>cv);
    }
    pthread_mutex_unlock(ts->lock);

    printf("Thread %d -- after barrier\n", ts->id);
  }
```

```
main()
{
     TStruct ts[NTHREADS];
     pthread_t tids[NTHREADS];
     int i, ndone;
     pthread_mutex_t lock;
     pthread_cond_t cv;
     void *retval;

     pthread_mutex_init(&lock, NULL);
     pthread_cond_init(&cv, NULL);
     ndone = 0;

    for (i = 0; i < NTHREADS; i++) {
        ts[i].lock = &lock;
        ts[i].cv = &cv;
        ts[i].ndone = &ndone;
        ts[i].id = i;
    }

    for (i = 0; i < NTHREADS; i++) {
       pthread_create(tids+i, NULL, barrier, ts+i);
    }

   for (i = 0; i < NTHREADS; i++) {
      pthread_join(tids[i], &retval);
    }
    printf("done\n");
}
```