PROGRAMOWANIE SYSTEMÓW CZASU RZECZYWISTEGO

LABORATORIUM

Temat: THREADS SYNCHRONIZATION METHODS

Mariusz Rudnicki

2016

1. Wstęp

W systemach wielowątkowych wspólne zasoby procesu takie, jak:

- pamięć:
  - kod programu;
  - dane;
- otwarte pliki;
- identyfikatory:
  - id użytkownika,
  - id grupy;
- timery,

są współdzielone przez wszystkie wątki należące do procesu. Wątki konkurują pomiędzy sobą o dostęp do poszczególnych zasobów. Może to prowadzić do niepożądanych sytuacji, w których wielokrotne zapisy tego samego obszaru pamięci mogą nadpisywać oczekiwane dane. W takich sytuacjach wątki czytające nie wiedzą kiedy dane są stabilne. Operowanie na niewłaściwych danych prowadzi do wypracowywania błędnych wyników, a w konsekwencji może być przyczyną podejmowania niewłaściwych decyzji, które mogą mieć katastrofalne skutki.

Rozwiązaniem w/w problemów jest stosowanie metod synchronizacji pracy wątków. QNX Neutrino dostarcza wielu metod synchronizacji wątków. Do najważniejszych możemy zaliczyć wymienione poniżej:

- mutex – wzajemne wykluczenie wątków;
- condvar – oczekiwanie na zmienną;
- semaphore – oczekiwanie na licznik;
- rwlock – synchronizacja wątków piszących i czytających;
- join – synchronizacja do zakończenia wątku;
- spinlock – oczekiwanie na alokację pamięci;
- sleepon – podobnie do condvars, z dynamiczną alokacją;
- barrier – oczekiwanie na określoną liczbę wątków.

Na laboratorium będziemy badać działanie trzech pierwszych metod.

2. Przygotowanie platformy i środowiska IDE

Uruchom QNX Neutrino za pomocą maszyny wirtualnej. Sprawdź poleceniem ***pidin*** czy jest uruchomiony proces ***qconn***. Jeśli nie ma go na liście pracujących procesów uruchom go poleceniem ***qconn #***.Sprawdź adres IP poleceniem ***ifconfig***. Następnie uruchom środowisko QNX Momentics IDE. Wybierz przestrzeń roboczą wskazaną przez prowadzącego.

3. Przeanalizuj poniższy kod programu.

```
/*
The exercise is to use the mutex construct that we learned about
to modify the source to prevent our access problem.
*/

#include <stdio.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <sched.h>
/*
The number of threads that we want to have running
simultaneously.
*/

#define NumThreads      4

/*
The global variables that the threads compete for. To demonstrate
contention, there are two variables that have to be updated
"atomically". With RR scheduling, there is a possibility that one
thread will update one of the variables, and get preempted by
another thread, which will update both.  When our original thread
runs again, it will continue the update, and discover that the
variables are out of sync.
*/

volatile int     var1;
volatile int     var2;

void    *update_thread (void *);

char    *progname = "mutex";
pthread_mutex_t var_mutex;

main (){
  pthread_t           threadID [NumThreads];  // a place to hold
                                              // the thread ID's
  pthread_attr_t      attrib;                 // scheduling
                                              // attributes
  struct sched_param  param;                  // for setting
                                              // priority
  int                 i, policy;

  setvbuf (stdout, NULL, _IOLBF, 0);
```

```
  var1 = var2 = 0;         /* initialize to known values */

  printf ("%s:  starting; creating threads\n", progname);
/*
Mutex initialization
*/
  pthread_mutex_init(&var_mutex, NULL );

/*
We want to create the new threads using Round Robin scheduling,
and a lowered priority, so set up a thread attributes structure.
We use a lower priority since these threads will be hogging the
CPU.
*/

  pthread_getschedparam( pthread_self(), &policy, &param );
  pthread_attr_init (&attrib);
  pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attrib, SCHED_RR);
  param.sched_priority -= 2;       // drop priority a couple
                                   // levels
  pthread_attr_setschedparam (&attrib, &param);
  attrib.flags |= PTHREAD_CANCEL_ASYNCHRONOUS;

/*
Create the threads.  As soon as each pthread_create call is done,
the thread has been started.
*/
  for(i = 0; i < NumThreads; i++) {
    pthread_create(&threadID [i], &attrib, &update_thread,
                   (void*)i);
  }
/*
Let the other threads run for a while
*/

  sleep(20);

/*
And then kill them all
*/


  printf("%s:  stopping; cancelling threads\n", progname);
  for(i = 0; i < NumThreads; i++) {
    pthread_cancel (threadID [i]);
  }
```

```
  printf("%s:  all done, var1 is %d, var2 is %d\n", progname,
         var1, var2);
  fflush (stdout);
  sleep(1);
  exit(0);
}

/*
The actual thread.

The thread ensures that var1 == var2.  If this is not the case,
the thread sets var1 = var2, and prints a message.
Var1 and Var2 are incremented.
Looking at the source, if there were no "synchronization"
problems, then var1 would always be equal to var2.  Run this
program and see what the actual result is...
*/

void do_work(){
  static int var3;

  var3++;

/*
For faster/slower processors, may need to tune this program by
modifying the frequency of this printf -- add/remove a 0
*/

  if( !(var3 % 10000000) )
    printf ("%s: thread %d did some work\n", progname,
            pthread_self());
}

void *
update_thread (void *i){
  while (1) {
    pthread_mutex_lock(&var_mutex);
    if (var1 != var2) {
      int lvar1, lvar2;
      lvar1 = var1;
      lvar2 = var2;
      var1 = var2;
      pthread_mutex_unlock(&var_mutex);
      printf("%s:  thread %d, var1 (%d) is not equal to var2
             (%d)!\n", progname, (int) i, lvar1, lvar2);
    }else
```

```
      pthread_mutex_unlock(&var_mutex);
/* do some work here */
    do_work();
     pthread_mutex_lock(&var_mutex);
    var1++;
    var2++;
    pthread_mutex_unlock(&var_mutex);
  }
  return (NULL);
}
```

4. Uruchom program z projektu **Mutex1**. Sprawdź jego działanie. Jaki jest wynik działania programu?
5. Uzupełnij kod projektu **Mutex1**, tak aby praca wątków była synchronizowana przy pomocy mutexa.
6. Sprawdź wynik działania zmodyfikowanego kodu. Sprawdź w jakim stanie są inne wątki, gdy jeden z nich wykonuje kod sekcji krytycznej. Do sprawdzeń użyj **perspektywy QNX System Information.** Wynik działania programu przedstaw prowadzącemu.
7. **Przeanalizuj poniższy kod programu.**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <sched.h>

/
The number of threads that we want to have running
simultaneously.
*/

#define NumThreads      4
```

```
/*
Global variables
*/
volatile char Text[3][10];
volatile unsigned int k=0;

void  *update_thread (void *);

char  *progname = "mutex";
pthread_mutex_t var_mutex;

int main(int argc, char *argv[]) {
  pthread_t           threadID [NumThreads];  // a place to hold
                                              // the thread ID's
  pthread_attr_t      attrib;                 // scheduling
                                              // attributes
  struct sched_param  param;                  // for setting
                                              // priority
  int                 i, policy;

  int oldcancel;
  setvbuf (stdout, NULL, _IOLBF, 0);

  memcpy((void *)&Text[0][0],"Test \0", 6);
  memcpy((void *)&Text[1][0],"Mutex \0", 7);
  memcpy((void *)&Text[2][0],"Nomutex \0", 9);

  printf ("%s:  starting; creating threads\n", progname);
  pthread_mutex_init(&var_mutex, NULL );

/*
We want to create the new threads using Round Robin scheduling,
and a lowered priority, so set up a thread attributes structure.
We use a lower priority since these threads will be hogging the
CPU
*/

  pthread_getschedparam( pthread_self(), &policy, &param );
  pthread_attr_init (&attrib);
  pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attrib, SCHED_RR);
  param.sched_priority -= 2;      // drop priority a couple
                                  // levels
  pthread_attr_setschedparam (&attrib, &param);
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldcancel);
/*
```

```
Create the threads.  As soon as each pthread_create call is done,
the thread has been started.
*/

  for(i = 0; i < NumThreads; i++) {
    pthread_create(&threadID [i], &attrib, &update_thread,
                   (void *)i);
  }
/*
Let the other threads run for a while
*/
  sleep (10);
/*
And then kill them all
*/
  printf ("%s:  stopping; cancelling threads\n", progname);

  for(i = 0; i < NumThreads; i++){
    pthread_cancel (threadID [i]);
  }
  fflush (stdout);
  sleep (1);
  return EXIT_SUCCESS;
}

void *
update_thread (void *i){
  while(1){
    pthread_mutex_lock(&var_mutex);
    for(k=0;k<3;k++){
      if((k%3)<2) printf("%s %d Thread.\n", &Text[(k%3)][0],
                         (int)i);
      else printf("%s %d. Thread.\n", &Text[(k%3)][0], (int)i);
    }
     delay(1000);
     pthread_mutex_unlock(&var_mutex);
    }
    return (NULL);
}
```

8. Uruchom program z projektu **Mutex**. Sprawdź jego działanie. Jaki jest wynik działania programu?

9. Uzupełnij kod projektu **Mutex**, tak aby praca wątków była synchronizowana przy pomocy mutexa. Sprawdź wynik działania programu.

10. **Przeanalizuj poniższy kod programu.**

```
/*
Objectives:
This code was a two-state example.  The producer (or state 0) did
something which caused the consumer (or state 1) to run.  State 1
did something which caused a return to state 0.  Each thread
implemented one of the states.

This example will have 4 states in its state machine with the
following state transitions:
State 0 -> State 1
State 1 -> State 2 if state 1's internal variable indicates
"even"
State 1 -> State 3 if state 1's internal variable indicates "odd"
State 2 -> State 0
State 3 -> State 0

And, of course, one thread implementing each state, sharing the
same state variable and condition variable for notification of
change in the state variable.
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <sched.h>

/*
Our global variables.
*/
volatile int        state;          // which state we are in

/*
Our mutex and condition variable
*/

pthread_mutex_t     mutex ;
pthread_cond_t      cond  ;

void    *state_0 (void *);
void    *state_1 (void *);
void    *state_2 (void *);
```

```
void    *state_3 (void *);

char    *progname = "condvar";

main (){
  setvbuf (stdout, NULL, _IOLBF, 0);

  pthread_mutex_init( &mutex, NULL );
  pthread_cond_init( &cond, NULL );
  state = 0;

  pthread_create (NULL, NULL, state_1, NULL);
  pthread_create (NULL, NULL, state_0, NULL);
  pthread_create (NULL, NULL, state_2, NULL);
  pthread_create (NULL, NULL, state_3, NULL);



  sleep (20);     // let the threads run
  printf ("%s:  main, exiting\n", progname);
}

/*
State 0 handler (was producer)
*/

void *
state_0 (void *arg){
  while (1) {
    pthread_mutex_lock (&mutex);
    while (state != 0) {
      pthread_cond_wait (&cond, &mutex);
    }
    printf ("%s:  transit 0 -> 1\n", progname);
    state = 1;
    /* don't chew all the CPU time */
    delay(100);
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock (&mutex);
  }
  return (NULL);
}

/*
State 1 handler (was consumer)
*/
void *
```

```
state_1 (void *arg){
  int i;
//int new_state = 2;
  while (1) {
    pthread_mutex_lock (&mutex);
    while (state != 1) {
      pthread_cond_wait (&cond, &mutex);
    }
    if( ++i & 0x1 ){
      state = 3;
    }
    else{
      state = 2;
    }

// "clever" ways to set next state
//        state = (++i & 0x1)? 3:2;
//
//        state = new_state ^= 0x1;
//

    printf ("%s:  transit 1 -> %d\n", progname, state);
    pthread_cond_broadcast (&cond);
    pthread_mutex_unlock (&mutex);
  }
  return (NULL);
}

void *
state_2 (void *arg){
  while (1) {
    pthread_mutex_lock (&mutex);
    while (state != 2) {
      pthread_cond_wait (&cond, &mutex);
    }
    printf ("%s:  transit 2 -> 0\n", progname);
    state = 0;
    pthread_cond_broadcast (&cond);
    pthread_mutex_unlock (&mutex);
  }
  return (NULL);
}

void *
state_3 (void *arg){
  while (1) {
    pthread_mutex_lock (&mutex);
```

```
    while (state != 3) {
      pthread_cond_wait (&cond, &mutex);
    }
    printf ("%s:  transit 3 -> 0\n", progname);
    state = 0;
    pthread_cond_broadcast (&cond);
    pthread_mutex_unlock (&mutex);
  }
  return (NULL);
}
```

11. Uruchom program z projektu **Condvar**. Sprawdź jego działanie. Jaki jest wynik działania programu?
12. Zmodyfikuj kod programu **Condvar** w taki sposób, aby wszystkie wątki consumer, wykonały pracę po zmianie stanu zmiennej warunkowej.
13. Sprawdź wynik działania uruchomionego kodu z pkt 11. Do sprawdzeń użyj **perspektywy QNX System Information.** Wynik działania programu przedstaw prowadzącemu.
14. **Przeanalizuj poniższy kod programu.**

```
/*
This module demonstrates POSIX semaphores.
Operation:
A counting semaphore is created, primed with 0 counts. Five
consumer threads are started, each trying to obtain the
semaphore. A producer thread is created, which periodically posts
the semaphore, unblocking one of the consumer threads.
*/

#include <stdio.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <malloc.h>
```

```c
/*
Our global variables, and forward references
*/

sem_t  *mySemaphore;

void   *producer (void *);
void   *consumer (void *);

char   *progname = "semex";

#define SEM_NAME "/Semex"

main (){
  int     i;

  setvbuf (stdout, NULL, _IOLBF, 0);

// #define  Named
#ifdef  Named
    mySemaphore = sem_open (SEM_NAME, O_CREAT, S_IRWXU, 0);
    /* not sharing with other process, so immediately unlink */
    sem_unlink( SEM_NAME );
#else   // Named
    mySemaphore = malloc (sizeof (sem_t));
    sem_init (mySemaphore, 1, 0);
#endif  // Named

  for(i = 0; i < 5; i++){
    pthread_create (NULL, NULL, consumer, (void *) i);
  }
  pthread_create (NULL, NULL, producer, (void *) 1);
  sleep (20);     // let the threads run
  printf ("%s:  main, exiting\n", progname);
}
/*
Producer
*/

void *
producer (void *i){
  while (1) {
    sleep (1);
    printf ("%s: (producer %d), posted semaphore\n", progname,
            (int) i);
    sem_post (mySemaphore);
```

```
  }
  return (NULL);
}

/*
Consumer
*/

void *
consumer (void *i){
  while (1) {
    sem_wait (mySemaphore);
    printf ("%s: (consumer %d) got semaphore\n", progname,
            (int) i);
  }
  return (NULL);
}
```

15. Uruchom program z projektu **Semex**. Sprawdź jego działanie. Jaki jest wynik działania programu? Jakiego rodzaju semafora używa program?
16. Zmodyfikuj kod programu **Semex** w taki sposób, aby wątki do synchronizacji swojej pracy używały innego rodzaju semafora.
17. Uruchom i sprawdź działanie zmodyfikowanego kodu.

18. Sprawozdanie.

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … … … …

Grupa: