5.4 Interaktion

Literatur

Mellor-Crummey, J. M.; Scott, M. L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocesors. ACM Transactions on Computer Systems, Vol. 9, No. 1, February 1991, pp.21-65.

Für UMA und NUMA Architekturen sind zwei Klassen von Koordinierungsmechanismen bedeutsam:

- ❑ Blockierende Koordinierung
- ❑ Koordinierung durch aktives Warten

  Bedeutsam bei kurzen kritischen Abschnitten, zwei Ausprägungen

  - Spinlocks

    zur Erzielung gegenseitigen Ausschlusses
  - Barrieren (barriers)

    zur Koordinierung des Übergangs von parallelen in serielle Phasen

  Probleme

  - Bei großen Prozessorzahlen entstehen leicht ‘hot spots’
  - Untersuchungen von Agarwal und Cherian an Benchmarkprogrammen:
    - in Systemen mit Cache-Kohärenz ist mehr als die Hälfte der Fehlzugriffe durch Spinlocks bedingt
    - werden Spinlock-Variable nur im Arbeitsspeicher geführt, erzeugen die Spinlocks nahezu die Hälfte des Datentransports zwischen CPU und Arbeitsspeicher

  Folgerung: Entwicklung geeigneter Algorithmen

5.4.1 Spinlocks

- ❑ Teste-und setze
  - Bei alleiniger Führung der Variablen im Arbeitsspeicher Belastung der Übertragungswege durch wartende Fäden
  - Bei Caches mit Kohärenz häufige Invalidierungen der Variablen und damit Rückfall auf das vorige Problem
  - Bei Caches ohne automatische Kohärenz zusätzlicher Aufwand um für die Spinlock-Variable Kohärenz zu sichern
  - Verbesserung durch Erhöhen der Zeit bis zur nächsten Überprüfung, falls letzte Überprüfung negativ

```
#include <pthread.h>

#include <iostream.h>

#define NUMBER_OF_THREADS 5

extern "C" void clock_init(int *);
extern "C" void clock_lock(int *);
extern "C" void clock_unlock(int *);
extern "C" int clock_trylock(int *, int);

extern "C" void sleep(int);

int interruptible = 0;

int test_and_set(int *v)
{ int old;
  clock_lock(&interruptible);
  old = *v;
  *v = 1;
  interruptible = 1;
  clock_unlock(&interruptible);
  return old;
}
```



```
void spinning(int delay)
{ sleep(delay);
}

class lock {
  int value;
public:
  lock();
  void acquire();
  void release();
};

lock::lock()
{ value = 0;
};

void lock::acquire()
{ int delay = 1;
  while (test_and_set(&value)) {
    spinning(delay);
    delay = 2 * delay;
  }
}

void lock::release()
{ value = 0;
}

lock mutex;
```

```
int main(int argc, char **argv)
{ int i;
  void *task(void *);
  void scheduler(int);
  pthread_t tid[NUMBER_OF_THREADS];
    // array of thread IDs */
  int threadnumber[NUMBER_OF_THREADS];

  cout << "main here\n" << flush;
  for (i = 0; i < NUMBER_OF_THREADS; i++) {
    threadnumber[i] = i;
    pthread_create(&tid[i], NULL, &task,
                 &(threadnumber[i]));
  }
  for ( i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(tid[i], NULL);
  cout << "\nmain() reporting that all "
       << i << " threads have terminated\n";
}
```



```
void *task(void *x)
{ int arg = *((int *) x);
  int i;
  cout << "thread " << arg << " started\n"
       << flush;
  sleep(1);
  for(i = 0; i < 10; i++) {
    cout << char(97 + arg) << flush;
    mutex.acquire();
    cout << "[" << arg << flush;
    spinning(1);
    cout << arg << flush;
    mutex.release();
    cout << arg << "]" << flush;
    spinning(2);
  }
}
```



```
a.out
main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
b[1deca11][2b22]c[222]c[222]c[222][3c33]d[333]d[
333]d[333]d[333][2d22][3c33][2d22]c[222]c[222]c[
222]c[222][333]d[333]d[333]d[333][000][1a11]b[11
1]b[111]b[111]b[111]b[111]b[111]b[111]b[111][000
]a[000]a[000]a[000]a[000]a[000]a[000]a[000]a[000
][444]e[444]e[444]e[444]e[444]e[444]e[444]e[444]
e[444]e[444]
main() reporting that all 5 threads have termi-
nated
```


- Ticket-Lock

```
#include <pthread.h>

#include <iostream.h>

#define NUMBER_OF_THREADS 5

extern "C" void clock_init(int *);
extern "C" void clock_lock(int *);
extern "C" void clock_unlock(int *);
extern "C" int clock_trylock(int *, int);

extern "C" void sleep(int);

int interruptible = 0;

int fetch_and_increment(int *v)
{ int old;
  clock_lock(&interruptible);
  old = *v;
  (*v)++;
  clock_unlock(&interruptible);
  return old;
}

void spinning(int delay)
{ sleep(delay);
}
```

```
class lock {
  int next_ticket;
  int now_serving;
public:
  lock();
  void acquire();
  void release();
};

lock::lock()
{ next_ticket = 0;
  now_serving = 0;
};

void lock::acquire()
{ int my_ticket =
    fetch_and_increment(&next_ticket);
  do {
    spinning(my_ticket - now_serving);
    if (now_serving == my_ticket)
      return;
  } while (1);
}

void lock::release()
{ now_serving++;
}

lock mutex;
```



```
int main(int argc, char **argv)
{ int i;
  void *task(void *);
  void scheduler(int);
  pthread_t tid[NUMBER_OF_THREADS];      // array of thread IDs
  int threadnumber[NUMBER_OF_THREADS];

  cout << "main here\n" << flush;
  for (i = 0; i < NUMBER_OF_THREADS; i++) {
    threadnumber[i] = i;
    pthread_create(&tid[i], NULL, &task, &(threadnumber[i]));
  }
  for ( i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(tid[i], NULL);
  cout << "\nmain() reporting that all "
       << i << " threads have terminated\n";
}

void task(void *x)
{ int arg = *((int *) x);
  int i;
  cout << "thread " << arg << " started\n"
       << flush;
  sleep(1);
  for(i = 0; i < 10; i++) {
    cout << char(97+arg) << flush;
    mutex.acquire();
    cout << "[" << arg << flush;
    spinning(1);
    cout << arg << flush;
    mutex.release();
    cout << arg << "]" << flush;
    spinning(2);
  }
}
```



```
a.out
main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
c[2ebda22][444][0ce00][4a44][111]e[4b44][1e11][0
b00][3a33][1d11][222][3b33]c[2d22][1c11][3b33][4
d44][1e11][0b00][4a44][3e33][0d00][1a11][3b33][0
d00][1a11][3b33][0d00][4a44][3e33][2d22][4c44][0
e00][2a22][1c11][0b00][444][222][111]a[0cbe00][3
a33]d[333]d[333][222]c[222]c[222]c[222][444][0e0
0][444][111]
main() reporting that all 5 threads have termi-
nated
```


- ❏ Array-Based Queuing Lock

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <iostream.h>
#include <lwp/interfaces.h>
#define MAXPRIO      10
#define NUMBER_OF_THREADS 5

int interruptible = 1;

enum lockvalue {has_lock, must_wait};

void atomic_add(int *v1, int v2)
{ interruptible = 0;
  *v1 = *v1 + v2;
  interruptible = 1;
}

int fetch_and_increment(int *v)
{ int old;
  interruptible = 0;
  old = *v;
  (*v)++;
  interruptible = 1;
  return old;
}

void spinning(int delay)
{ for (int j = 0; j < delay * 500; j++) {
    float x, y, z;
    x = 2; y = 3; z = x / y;
  }
}
```

```
class lock {
  lockvalue slots[NUMBER_OF_THREADS];
  int next_slot;
public:
  lock();
  void acquire(int *);
  void release(int *);
};

lock::lock()
{ slots[0] = has_lock;
  for (int i = 1; i < NUMBER_OF_THREADS; i++) {
    slots[i] = must_wait;
  }
  next_slot = 0;
};

void lock::acquire(int *my_place)
{ *my_place = fetch_and_increment(&next_slot);
  if (*my_place == NUMBER_OF_THREADS) {
    atomic_add(&next_slot, -NUMBER_OF_THREADS);
  }
  *my_place = *my_place % NUMBER_OF_THREADS;
  while (slots[*my_place] == must_wait);
  slots[*my_place] = must_wait;
}

void lock::release(int *my_place)
{ slots[(*my_place + 1) % NUMBER_OF_THREADS]
    = has_lock;
}

lock mutex;
```



```
int main(int argc, char **argv)
{ int i;
  void task(int);
  void scheduler();
  thread_t tid;
  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, NUMBER_OF_THREADS);
  (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0, lwp_newstk(),
                    0);
  for (i = 0; i < NUMBER_OF_THREADS; i++)
  lwp_create((thread_t *)0, task, MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}

void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    if (interruptible) {
      lwp_resched(MINPRIO);
    }
  }
}

void task(int arg)
{ int my_place;
  for(;;) {
    cout << char(97+arg) << flush;
    mutex.acquire(&my_place);
    cout << "[" << arg << flush;
    spinning(1);
    cout << arg << flush;
    mutex.release(&my_place);
    cout << arg << "]" << flush;
    spinning(2);
  }
}
```



```
a[000]a[0bcde00]a[111]b[222]c[333]d[444][111][22
eb2]c[000][333][444]a[111][222]cde[000]b[222][33
ac3]d[444][111]e[000]b[22a2]c[333][444][111]de[0
00]b[222][333][4acd44]e[111][000]b[2a22]c[333]d[
444][11e1]b[000][22a2]c[333]d[444][11e1][00b0]a[
222]c[333]d[444]e[111][00b0][2a22]c[333][44d4]e[
111]b[000][222][333][44acd4]e[111][000]b[222][33
ac3]d[444][111]e[000]b[222][33ac3]d[444][111]e[0
00]b[22a2]c[333][44d4]e[111][000]b[222][33ac3]d[
444][111]e[000]b[2a22]c[333][444][111]de[00b0][2
22][333][44acd4]e[111][000]b[222][333][44acd4]e[
111][000]b[222][333][444]a[111]cde[000]b[222][33
ac3]d[444]e[111][000]b[222][33ac3]d[444][1e11]b[
000][222][3ac33]d[444]e[111][000]b[222][33ac3]d[
444][111]e[000]b[222][33ac3]d[444][111]e[000]b[2
22][33ac3]d[444][111]e[000]b[222][33ac3]d[444][1
11]e[000]b[22a2]c[333][444][111]de[000]b[222][33
ac3]d[444][111]e[000]b[222][33ac3]d[444][111]e[0
00]b[222][3ac33]d[444]e[111][000]b[222][33ac3]d[
444][111]e[000]b[222][33ac3]d[444][111]e[000]b[2
22][33ac3]d[444][00e0]a[111][22b2]c[333]
```


- ❏ List-Based Queuing Lock

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <iostream.h>
#include <lwp/interfaces.h>
#define MAXPRIO      10
#define NUMBER_OF_THREADS 5

typedef int boolean;
class qnode;
class lock;

int interruptible = 1;

qnode *fetch_and_store(qnode **l, qnode *i)
{ qnode *tmp;
  interruptible = 0;
  tmp = *l;
  *l = i;
  interruptible = 1;
  return tmp;
}

boolean compare_and_swap(qnode **l,
                         qnode *old_qnode,
                         qnode *new_qnode)
{ if (*l != old_qnode)
    return false;
  *l = new_qnode;
  return true;
}

void spinning(int delay)
{ for (int j = 0; j < delay * 500; j++) {
    float x, y, z;
    x = 2; y = 3; z = x / y;
  }
}
```



```
class qnode {
public:
  qnode *next;
  boolean locked;
};

class lock {
  qnode *qnode_ptr;
public:
  lock();
  void acquire(qnode *);
  void release(qnode *);
  friend qnode *fetch_and_store(qnode **l,
                               qnode *i);
  friend
    boolean compare_and_swap(qnode **l,
                            qnode *old_qnode,
                            qnode *new_qnode);
};
```



```
lock::lock()
{ qnode_ptr = 0;
};

void lock::acquire(qnode *i)
{ i->next = 0;
  qnode *predecessor
    = fetch_and_store(&qnode_ptr, i);
  if (predecessor != 0) {
    i->locked = true;
    predecessor->next = i;
    while (i->locked) ;
  }
}

void lock::release(qnode *i)
{ if (i->next == 0) {
    if (compare_and_swap(&qnode_ptr, i, 0)) {
      return;
    }
    while (i->next == 0) ;
  }
  i->next->locked = false;
}

lock mutex;
```



```
int main(int argc, char **argv)
{ int i;
  void task(int);
  void scheduler();
  thread_t tid;
  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, NUMBER_OF_THREADS);
  (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0, lwp_newstk(),
0);
  for (i = 0; i < NUMBER_OF_THREADS; i++)
  lwp_create((thread_t *)0, task, MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}

void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    if (interruptible) {
      lwp_resched(MINPRIO);
    }
  }
}

void task(int arg)
{ int my_place;
  qnode my_qnode;
  for(;;) {
    cout << char(97+arg) << flush;
    mutex.acquire(&my_qnode);
    cout << "[" << arg << flush;
    spinning(1);
    cout << arg << flush;
    mutex.release(&my_qnode);
    cout << arg << "]" << flush;
    spinning(2);
  }
}
```



```
a[000]a[000]a[0bcde00]a[111]b[222]c[333]d[444]e[
000]a[111]b[222]c[333]d[444][000][11ea1]b[222][3
3c3]d[444][00e0]a[111][22b2]c[333][44d4]e[000][1
11][222][333][4abcd44]e[000]a[111][22b2]c[333][4
4d4]e[000][11a1]b[222][33c3]d[444][000][111][2ea
b22]c[333]d[444][00e0]a[111][22b2]c[333][44d4]e[
000][111]b[222]c[333]d[444]ea[111][222][33bc3]d[
444][00e0]a[111][22b2]c[333][44d4]e[000][11a1]b[
222][333][4cd44]e[000][11a1]b[222][33c3][444][00
de0]a[111][22b2]c[333][44d4]e[000][111][222][3ab
c33]d[444]e[000][11a1]b[222][33c3]d[444][00e0]a[
111]b[222][33c3]d[444][00e0]a[111][222][33bc3]d[
444][00e0]a[111][22b2]c[333][44d4][0e00]a[111]b[
222]c[333][444][00de0][111][222][333][444]ab-
cde[000]a[111][22b2]c[333]d[444]e[000][11a1]b[22
2][3c33]d[444][00e0]a[111][22b2]c[333][44d4]e[00
0][11a1]b[222][33c3]d[444][000][1ea11]b[222][33c
3]d[444][00e0]a[111][22b2]c[333][444]e[000]a[1d1
1]b[222][444][000]c[33ea3]d
```

5.4.2 Barrieren

- ❏ Zentralisierte Barrieren

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <iostream.h>
#include <lwp/interfaces.h>
#define MAXPRIO      10
#define NUMBER_OF_THREADS 3

typedef int boolean;
int interruptible = 1;

int fetch_and_decrement(int *v)
{ int old;
  interruptible = 0;
  old = *v;
  (*v)--;
  interruptible = 1;
  return old;
}

void spinning(int delay)
{ for (int j = 0; j < delay * 500; j++) {
    float x, y, z;
    x = 2; y = 3; z = x / y;
  }
}

int count = NUMBER_OF_THREADS;
boolean sense = true;

void central_barrier(boolean *local_sense, int
arg)
{ *local_sense = !(*local_sense);
  if (fetch_and_decrement(&count) == 1) {
    count = NUMBER_OF_THREADS;
```



```
    sense = *local_sense;
  } else {
    while (sense != *local_sense) ;
  }
}

int main(int argc, char **argv)
{ int i;
  void task(int);
  void scheduler();
  thread_t tid;
  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, NUMBER_OF_THREADS);
  (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0, lwp_newstk(),
0);
  for (i = 0; i < NUMBER_OF_THREADS; i++)
  lwp_create((thread_t *)0, task, MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}

void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    if (interruptible) {
      lwp_resched(MINPRIO);
    }
  }
}
```



```
void task(int arg)
{ boolean local_sense = true;
  int round = 0;
  for(;;) {
    round++;
    cout << "\nthread " << arg
         << " begins round " << round << flush;
    spinning(1);
    central_barrier(&local_sense, arg);
  }
}
```



```
thread 0 begins round 1
thread 1 begins round 1
thread 2 begins round 1
thread 2 begins round 2
thread 0 begins round 2
thread 1 begins round 2
thread 2 begins round 3
thread 0 begins round 3
thread 1 begins round 3
thread 1 begins round 4
thread 2 begins round 4
thread 0 begins round 4
thread 2 begins round 5
thread 0 begins round 5
thread 1 begins round 5
thread 1 begins round 6
thread 2 begins round 6
thread 0 begins round 6
thread 2 begins round 7
thread 0 begins round 7
thread 1 begins round 7
thread 1 begins round 8
thread 2 begins round 8
thread 0 begins round 8
thread 0 begins round 9
thread 1 begins round 9
thread 2 begins round 9
thread 1 begins round 10
thread 2 begins round 10
thread 0 begins round 10
thread 0 begins round 11
```

❑ Dissemination Barrier

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <iostream.h>
#include <lwp/interfaces.h>
#define nil 0
#define MAXPRIO      10

typedef int boolean;
int interruptible = 1;

void spinning(int delay)
{ for (int j = 0; j < delay * 500; j++) {
    float x, y, z;
    x = 2; y = 3; z = x / y;
  }
}

// LogP is ceiling(ld(NUMBER_OF_THREADS))
#define LogP 2
#define NUMBER_OF_THREADS 4

class flags {
public:
  boolean myflags[2][LogP];
  boolean *partnerflags[2][LogP];
  flags();
};
```



```
flags allnodes[NUMBER_OF_THREADS];

flags::flags()
{ int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < LogP; j++) {
      myflags[i][j] = false;
    }
  }
}

void dissemination_barrier(flags *localflags,
boolean *sense, int *parity)
{ for (int instance = 0; instance < LogP;
       instance++) {
    *(localflags
        ->partnerflags[*parity][instance])
      = *sense;
    while (localflags
             ->myflags[*parity][instance]
           != *sense) ;
  }
  if (*parity == 1) {
    *sense = !(*sense);
  }
  *parity = 1 - *parity;
}
```



```
int main(int argc, char **argv)
{ int i, j, r, k;
  void task(int);
  void scheduler();
  thread_t tid;

  for (i = 0; i < NUMBER_OF_THREADS; i++) {
    for (j = 0; j < NUMBER_OF_THREADS; j++) {
      for (k = 0; k < LogP; k++) {
        if (j == (i + (1 << k))
                   % NUMBER_OF_THREADS) {
          for (r = 0; r < 2; r++) {
            allnodes[i].partnerflags[r][k]
              = &(allnodes[j].myflags[r][k]);
          }
        }
      }
    }
  }

  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, NUMBER_OF_THREADS);
  (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0, lwp_newstk(),
                    0);
  for (i = 0; i < NUMBER_OF_THREADS; i++)
  lwp_create((thread_t *)0, task, MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}

void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    if (interruptible) {
      lwp_resched(MINPRIO);
    }
  }
}
```



```
void task(int arg)
{ int parity = 0;
  boolean sense = true;
  flags *localflags = &(allnodes[arg]);
  int round = 0;
  for(;;) {
    interruptible = 0;
    round++;
    cout << "thread " << arg << " begins round "
<< round << "\n" << flush;
    interruptible = 1;
    spinning(1);
    dissemination_barrier(localflags, &sense,
&parity);
  }
}
```

```
thread 0 begins round 1
thread 1 begins round 1
thread 2 begins round 1
thread 3 begins round 1
thread 3 begins round 2
thread 0 begins round 2
thread 1 begins round 2
thread 2 begins round 2
thread 2 begins round 3
thread 3 begins round 3
thread 0 begins round 3
thread 1 begins round 3
thread 1 begins round 4
thread 2 begins round 4
thread 3 begins round 4
thread 0 begins round 4
thread 0 begins round 5
thread 1 begins round 5
thread 2 begins round 5
thread 3 begins round 5
thread 3 begins round 6
```


- Tree-Based Barrier

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <iostream.h>
#include <lwp/interfaces.h>
#define nil 0
#define MAXPRIO      10

#define NUMBER_OF_THREADS 5

typedef int boolean;
int interruptible = 1;

void spinning(int delay)
{ for (int j = 0; j < delay * 500; j++) {
    float x, y, z;
    x = 2; y = 3; z = x / y;
  }
}
```



```
class treenode {
public:
  boolean parentsense;
  boolean *parentpointer;
  boolean *childpointers[2];
  boolean havechild[4];
  boolean childnotready[4];
  boolean dummy;
};

treenode nodes[NUMBER_OF_THREADS];

void tree_barrier(int vpid, boolean *sense)
{ int i;
  treenode *mynode = &(nodes[vpid]);
  while (mynode->childnotready[0] == true
         || mynode->childnotready[1] == true
         || mynode->childnotready[2] == true
         || mynode->childnotready[3] == true) ;
  for (i = 0; i < 4; i++) {
    mynode->childnotready[i]
      = mynode->havechild[i];
  }
  *(mynode->parentpointer) = false;
  if (vpid != 0) {
    while (mynode->parentsense != *sense) ;
  }
  *(mynode->childpointers[0]) = *sense;
  *(mynode->childpointers[1]) = *sense;
  *sense = !(*sense);
}
```



```
int main(int argc, char **argv)
{ int i, j, r, k;
  void task(int);
  void scheduler();
  thread_t tid;

  { int i, j;
    for (i = 0; i < NUMBER_OF_THREADS; i++) {
      for (j = 0; j < 4; j++) {
        nodes[i].childnotready[j] =
          (nodes[i].havechild[j]
             = (4*i+j+1 < NUMBER_OF_THREADS ?
                                   true : false));
      }
      nodes[i].parentpointer
        = (i == 0 ?
            &(nodes[i].dummy) :
            &(nodes[(i - 1) / 4]
                .childnotready[(i - 1) % 4]));
      nodes[i].childpointers[0]
        = (2 * i + 1 >= NUMBER_OF_THREADS ?
              &(nodes[i].dummy) :
              &(nodes[2 * i + 1].parentsense));
      nodes[i].childpointers[1]
        = (2 * i + 2 >= NUMBER_OF_THREADS ?
              &(nodes[i].dummy) :
              &(nodes[2 * i + 2].parentsense));
      nodes[i].parentsense = false;
    }
  }
```

```
  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, NUMBER_OF_THREADS);
  (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0, lwp_newstk(),
                    0);
  for (i = 0; i < NUMBER_OF_THREADS; i++)
  lwp_create((thread_t *)0, task, MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}

void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    if (interruptible) {
      lwp_resched(MINPRIO);
    }
  }
}

void task(int arg)
{ boolean sense = true;
  int round = 0;
  for(;;) {
    interruptible = 0;
    round++;
    cout << "thread " << arg
         << " begins round " << round
         << "\n" << flush;
    interruptible = 1;
    spinning(1);
    tree_barrier(arg, &sense);
  }
}
```



```
thread 0 begins round 1
thread 1 begins round 1
thread 2 begins round 1
thread 3 begins round 1
thread 4 begins round 1
thread 0 begins round 2
thread 1 begins round 2
thread 2 begins round 2
thread 3 begins round 2
thread 4 begins round 2
thread 0 begins round 3
thread 1 begins round 3
thread 2 begins round 3
thread 3 begins round 3
thread 4 begins round 3
thread 0 begins round 4
thread 1 begins round 4
thread 2 begins round 4
thread 3 begins round 4
thread 4 begins round 4
thread 0 begins round 5
thread 1 begins round 5
thread 2 begins round 5
thread 3 begins round 5
thread 4 begins round 5
```