4.4 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

Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

4.4.1 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 interruptible = 0;
void begin_of_atomic()
{ while (clock_trylock(&interruptible, 1) == 0) {
    sleep(1);
  }
}
void end_of_atomic()
{ clock_unlock(&interruptible);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int test_and_set(int *v)
{ int old;
  begin_of_atomic();
  old = *v;
  *v = 1;
  interruptible = 1;
  end_of_atomic;
  return old;
}

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


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
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;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ int i;
  void *task(void *);
  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";
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
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);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
c[2bdae22][111][0c00]b[111]a[000]b[111]a[000][2b
22]a[000]c[222][4ac44][000][1e11]a[0b00][111]a[0
00][2b22]a[0c00][222]a[000]c[222]a[000]c[222]c[2
22]c[222]c[222][444]e[444]e[444]e[444]e[444][333
]e[444]d[333]e[444]d[3e33][444]d[333]e[444]d[333
]d[333]d[333]d[333]d[333]d[333][111]b[111]b[111]
b[111]b[111]
main() reporting that all 5 threads have termi-
nated

main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
d[3bcae33][222]d[3c33][2d22][1c11][3b33][2d22][1
c11][4b44][2e22][444]c[222]e[444]c[222]e[444]c[2
22][3e33][4cd44][222][3e33]c[222]d[333][444]c[22
2]d[3e33][444]d[333]e[4d44][333]e[444]d[333]e[44
4][000]a[000]a[000]a[000]a[000]a[000][111]a[000]
b[111]a[000]b[111]a[000]b[111]a[000]b[111]b[111]
b[111]b[111]
main() reporting that all 5 threads have termi-
nated
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

❑ 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;
void begin_of_atomic()
{ while (clock_trylock(&interruptible, 1) == 0) {
    sleep(1);
  }
}
void end_of_atomic()
{ clock_unlock(&interruptible);
}

int fetch_and_increment(int *v)
{ int old;
  begin_of_atomic();
  old = *v;
  (*v)++;
  end_of_atomic();
  return old;
}


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


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
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;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ int i;
  void *task(void *);
  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);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
a.out
main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
c[2dbea22][333][111]c[4db44][0e00][2a22][333]c[1
d11][4b44][0e00][2a22][3c33][1d11][4b44][0e00][2
a22][3c33][1d11][4b44][0e00][2a22][3c33][1d11][4
b44][0e00][2a22][3c33][1d11][4b44][0e00][2a22][3
c33][1d11][4b44][000]e[222][3ac33][1d11][4b44][0
e00][2a22][3c33][1d11][444]b[0e00][2a22][333][11
1][444][000]
main() reporting that all 5 threads have termi-
nated
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

❑ Array-Based Queuing 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;
void begin_of_atomic()
{ while (clock_trylock(&interruptible, 1) == 0) {
    sleep(1);
  }
}
void end_of_atomic()
{ clock_unlock(&interruptible);
}

void atomic_add(int *v1, int v2)
{ begin_of_atomic();
  *v1 = *v1 + v2;
  end_of_atomic();
}

int fetch_and_increment(int *v)
{ int old;
  begin_of_atomic();
  old = *v;
  (*v)++;
  end_of_atomic();
  return old;
}

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


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
enum lockvalue {has_lock, must_wait};

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;
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
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)
    sleep(1);
  slots[*my_place] = must_wait;
}

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

lock mutex;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ int i;
  void *task(void *);
  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 terminat-
ed\n";
}

void task(void *x)
{ int my_place;
  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(&my_place);
    cout << "[" << arg << flush;
    spinning(1);
    cout << arg << flush;
    mutex.release(&my_place);
    cout << arg << "]" << flush;
    spinning(2);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
main here
thread 0 started
thread 1 started
thread 2 started
thread 3 started
thread 4 started
d[3cbae33][222]d[1c11][0b00][444][333]a[2e22][1d
11][0c00][4b44][3a33][222]e[1d11][000]c[444]b[33
3]a[2e22][111]d[000]c[4b44][3a33][222]e[111]d[0c
00][4b44][333]a[222]e[1d11][000]c[444]b[3a33][2e
22][111]d[000]c[444]b[3a33][2e22][1d11][0c00][4b
44][333]a[2e22][1d11][000]c[444][3b33]a[222]e[11
1][000][444]
main() reporting that all 5 threads have termi-
nated
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

❑ List-Based Queuing Lock

```
#define NUMBER_OF_PROCESSES 5

#include <iostream.h>
#include <sys/types.h>
union semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
};
#include <sys/ipc.h>

#include <sysent.h>
extern "C" void exit(int);
#include <sys/wait.h>

#include <errno.h>
extern char* sys_errlist[];

// Definitionen fuer gemeinsame Segmente
#include <sys/shm.h>
char **shm_free;

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);

int *interruptible;

void begin_atomic()
{ clock_lock(interruptible);
}

void end_atomic()
{ clock_unlock(interruptible);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
typedef int boolean;
class t_qnode;
class lock;

t_qnode *fetch_and_store(t_qnode **l,
                         t_qnode *i)
{ t_qnode *tmp;
  begin_atomic();
  tmp = *l;
  *l = i;
  end_atomic();
  return tmp;
}

boolean compare_and_swap(t_qnode **l,
                         t_qnode *old_qnode,
                         t_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;
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
class t_qnode {
  t_qnode *next;
  boolean locked;
  friend class lock;
public:
  void *operator new[](size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
  t_qnode()
    { next = NULL;
    };
};

class lock {
  t_qnode *qnode_ptr;
public:
  void *operator new(size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
  lock();
  void acquire(t_qnode *);
  void release(t_qnode *);
  friend
    t_qnode *fetch_and_store(t_qnode **l,
                             t_qnode *i);
  friend
    boolean compare_and_swap
                          (t_qnode **l,
                           t_qnode *old_qnode,
                           t_qnode *new_qnode);
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

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

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

void lock::release(t_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;
t_qnode *qnode;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ int i, meine_prozessnummer, semid,
      shmid, status;
  char *local_shm;
  semun seminit;
  void task(int);

  cout << "Beginn des Wurzelprogramms\n"
       << flush;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
// Einrichten eines `shared memory'-Segments
//    1. Tilgen, falls schon vorhanden
  shmid = shmget(1, 1, 0600 | IPC_CREAT);
  if ((status = shmctl(shmid, IPC_RMID, 0))
      == -1) {
    cout << "***** Fehler beim Tilgen des"
         << " gemeinsamen Sgementes: "
         << sys_errlist[errno] << "\n";
  }
//    2. Neu einrichten
  if ((shmid = shmget(1, 10000, 0600|IPC_CREAT))
      < 0) {
    cerr << "\n\nError in shmget for"
         << " shared memory: "
         << sys_errlist[errno]
         << "\n" << flush;
    exit(0);
  }
//    3. Zugaenglich machen
  if ((int)(local_shm
        = (char *)shmat(shmid, (void *)0, 0))
      == -1) {
    cerr << "\n\nError: " << sys_errlist[errno]
         << "\n" << flush;
    exit(0);
  }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
// Speicherverwaltung fuer
// gemeinsames Segment einrichten
  shm_free = (char **) local_shm;
  *shm_free = local_shm + 4;

// Allen Prozessen gemeinsame
// Datenstrukturen anlegen
  interruptible = (int *)(local_shm + 4);
  *shm_free += 4;
  clock_init(interruptible);
  mutex = new lock;
  qnode = new t_qnode[NUMBER_OF_PROCESSES];

  for (int lfd_nummer = 0;
       lfd_nummer < NUMBER_OF_PROCESSES;
       lfd_nummer++) {
    if (fork() == 0) {
      meine_prozessnummer = lfd_nummer;
      task(meine_prozessnummer);
      exit(0);
    }
  }
  for (i = 0; i < NUMBER_OF_PROCESSES; i++) {
    wait(&status);
  }
  cout << "\nAlle Prozesse sind beendet\n"
       << flush;
  }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
void task(int arg)
{ int i, my_place;
  for(i = 0; i < 10; i++) {
    cout << char(97+arg) << flush;
    mutex->acquire(&qnode[arg]);
    cout << "[" << arg << flush;
    spinning(100);
    cout << arg << flush;
    mutex->release(&qnode[arg]);
    cout << arg << "]" << flush;
    spinning(200);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
Beginn des Wurzelprogramms
a[000]a[0bc00][111]d[222]abe[333]dc[000][111]a[444][33
3][222]b[000]edc[111]b[444][333][222][111]a[000]e[444]
d[333]c[222]b[111]a[000]e[444]b[1dc11]ae[333][222]b[00
0][444]edc[111]a[444][333][222]b[000]edc[111]a[444][33
3]e[222]b[000]d[444]ec[111]a[333][444]d[222]b[000]e[33
3]dc[111][444][333][222]c[222]
Alle Prozesse sind beendet
```

An Ultra-Sparc mit 1 Prozessor

```
Beginn des Wurzelprogramms
a[000]a[000]b[111]a[000]a[000]b[111]b[111]c[222]c[222]
a[000]a[000]a[000]b[111]b[111]b[1dc11]bae[333][222][11
1][000][444]d[333]d[333]c[222]c[222]b[111]b[111]a[000]
a[000]e[444]e[444]d[333]d[333]d[333]d[333]c[222]e[444]
c[222]b[111]d[3ec33]d[444]e[222]c[333]d[444]e[222]c[33
3][444]e[222][444]e[444]e[444]
Alle Prozesse sind beendet
2.63u 0.01s 0:02.87 91.9%
```

An Ultra-Sparc mit 8 Prozessoren

```
Beginn des Wurzelprogramms
ab[0cde00][111][2a22][3b33][4c44][0d00][11e1][2a22][3b
33][4c44][000][1de11][2a22][3b33][4c44][000][1de11][2a
22][3b33][4c44][0d00][1e11][2a22][3b33][4c44][0d00][1e
11][2a22][3b33][4c44][000][1de11][2a22][3b33][4c44][0d
00][1e11][2a22][3b33][4c44][0d00][1e11][2a22][3b33][4c
44][000][1de11][222][333][444]
Alle Prozesse sind beendet
3.0u 0.0s 0:00 428% 0+0k 0+0io 0pf+0w
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

4.4.2 Barrieren

- ❑ Zentralisierte Barrieren

```
#define NUMBER_OF_PROCESSES 5

#include <iostream.h>
#include <sys/types.h>
union semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
};
#include <sys/ipc.h>

#include <sysent.h>
extern "C" void exit(int);
#include <sys/wait.h>

#include <errno.h>
extern char* sys_errlist[];

// Definitionen fuer gemeinsame Segmente
#include <sys/shm.h>
extern "C" int shmctl(int, int, struct shmid_ds *);
extern "C" int shmget(key_t, unsigned int, int);
extern "C" void *shmat(int, const void *, int);
char **shm_free;

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);

int *interruptible;

void begin_of_atomic()
{ clock_lock(interruptible);
}

void end_of_atomic()
{ clock_unlock(interruptible);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
class shared_int {
  public:
    int value;
    void *operator new(size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
};

typedef int boolean;
typedef shared_int shared_boolean;

int fetch_and_decrement(shared_int *v)
{ int old;
  begin_of_atomic();
  old = v->value;
  (v->value)--;
  end_of_atomic();
  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;
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
shared_int *count;
shared_boolean *sense;

void central_barrier(boolean *local_sense)
{ *local_sense = !(*local_sense);
  if (fetch_and_decrement(count) == 1) {
    count->value = NUMBER_OF_PROCESSES;
    sense->value = *local_sense;
  } else {
    while (sense->value != *local_sense) ;
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ int i, meine_prozessnummer, semid, shmid, status;
  char *local_shm;
  semun seminit;
  void task(int);

  cout << "Beginn des Wurzelprogramms\n" << flush;

// Einrichten eines `shared memory'-Segments
//    1. Tilgen, falls schon vorhanden
  shmid = shmget(1, 1, 0600 | IPC_CREAT);
  if ((status = shmctl(shmid, IPC_RMID, 0)) == -1) {
    cout << "***** Fehler beim Tilgen des gmeinsamen Sgementes: "
         << sys_errlist[errno] << "\n";
  }
//    2. Neu einrichten
  if ((shmid = shmget(1, 10000, 0600 | IPC_CREAT)) < 0) {
    cerr << "\n\nError in shmget for shared memory: "
         << sys_errlist[errno]
         << "\n" << flush;
    exit(0);
  }
//    3. Zugaenglich machen
  if ((int)(local_shm = (char *)shmat(shmid, (void *)0, 0))
      == -1) {
    cerr << "\n\nError: " << sys_errlist[errno] << "\n" << flush;
    exit(0);
  }
//    4. Speicherverwaltung fuer gemeinsames Segment einrichten
  shm_free = (char **) local_shm;
  *shm_free = local_shm + 4;

// Allen Prozessen gemeinsame Datenstrukturen anlegen
  interruptible = (int *)(local_shm + 4);
  *shm_free += 4;
  clock_init(interruptible);
  count = new shared_int();
  count->value = NUMBER_OF_PROCESSES;
  sense = new shared_boolean();
  sense->value = true;

  for (int lfd_nummer = 0;
       lfd_nummer < NUMBER_OF_PROCESSES; lfd_nummer++) {
    if (fork() == 0) {
      meine_prozessnummer = lfd_nummer;
      task(meine_prozessnummer);
      exit(0);
    }
  }
  for (i = 0; i < NUMBER_OF_PROCESSES; i++) {
    wait(&status);
  }
  cout << "\nAlle Prozesse sind beendet\n" << flush;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
void task(int arg)
{ boolean local_sense = true;
  int i, round = 0;
  for (i = 0; i < 10; i++) {
    round++;
    cout << "\nthread " << arg << " begins round
" << round << flush;
    spinning(1);
    central_barrier(&local_sense);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

- Dissemination Barrier

```
#define NUMBER_OF_PROCESSES 5
int LogP;  // Has to be initialized with
           // ceiling(ld(NUMBER_OF_PROCESSES))
#include <iostream.h>
#include <sys/types.h>
union semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
};
#include <sys/ipc.h>

#include <sysent.h>
extern "C" void exit(int);
#include <sys/wait.h>

#include <errno.h>
extern char* sys_errlist[];

// Definitionen fuer gemeinsame Segmente
#include <sys/shm.h>
extern "C" int shmctl(int, int, struct shmid_ds *);
extern "C" int shmget(key_t, unsigned int, int);
extern "C" void *shmat(int, const void *, int);
char **shm_free;

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);

int *interruptible;
void begin_of_atomic()
{ clock_lock(interruptible);
}
void end_of_atomic()
{ clock_unlock(interruptible);
}

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


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
class shared_int {
  public:
    int value;
    void *operator new(size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
    void *operator new[](size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
};

class shared_int_ptr {
  public:
    shared_int *ptr;
    void *operator new(size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
    void *operator new[](size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
class private_int {
  public:
    int value;
};

class private_int_ptr {
  public:
    private_int *ptr;
};

typedef shared_int shared_boolean;
typedef shared_int_ptr shared_boolean_ptr;
typedef private_int private_boolean;
typedef private_int_ptr private_boolean_ptr;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
class shared_flags {
  public:
    shared_boolean *myflags[2];
    shared_boolean_ptr *partnerflags[2];
    void *operator new[](size_t s)
    { char *tmp = *shm_free;
      *shm_free += s;
      return tmp;
    };
    void init(int LogP)
      { int i, r;
        myflags[0] = new shared_boolean[LogP];
        myflags[1] = new shared_boolean[LogP];
        partnerflags[0]
          = new shared_boolean_ptr[LogP];
        partnerflags[1]
          = new shared_boolean_ptr[LogP];
        for (r = 0; r < 2; r++)
          for (i = 0; i < LogP; i++)
            myflags[r][i].value = false;
    };
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
shared_flags *allnodes;

void dissemination_barrier
      (shared_flags *localflags,
       private_boolean *sense,
       private_int *parity,
       int pid,       // needed only for trace
       int round)     // needed only for trace
{ for (int instance = 0;
       instance < LogP;
       instance++) {
    cout << "process " << pid
         << ": round = " << round
         <<  ", dissemination round = "
         << instance << "\n" << flush;
    (localflags
       ->partnerflags[parity->value][instance])
         .ptr->value
      = sense->value;
    while ((localflags
              ->myflags[parity->value][instance])
                .value
            != sense->value) ;
  }
  if (parity->value == 1) {
    sense->value = !(sense->value);
  }
  parity->value = 1 - parity->value;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

```
int main(int argc, char **argv)
{ cout << "Beginn des Wurzelprogramms\n" << flush;

  { float tmp = NUMBER_OF_PROCESSES;
    LogP = 0;
    do {
      tmp /= 2;
      LogP++;
    } while (tmp > 1);
  }

  int i, meine_prozessnummer, shmid, status;
  char *local_shm;

// Einrichten eines `shared memory'-Segments
//    1. Tilgen, falls schon vorhanden
  shmid = shmget(1, 1, 0600 | IPC_CREAT);
  if ((status = shmctl(shmid, IPC_RMID, 0)) == -1) {
    cout << "***** Fehler beim Tilgen des gmeinsamen Sgementes: "
         << sys_errlist[errno] << "\n";
  }
//    2. Neu einrichten
  if ((shmid = shmget(1, 10000, 0600 | IPC_CREAT)) < 0) {
    cerr << "\n\nError in shmget for shared memory: "
         << sys_errlist[errno]
         << "\n" << flush;
    exit(0);
  }
//    3. Zugaenglich machen
  if ((int)(local_shm = (char *)shmat(shmid, (void *)0, 0))
     == -1) {
    cerr << "\n\nError: " << sys_errlist[errno] << "\n" << flush;
    exit(0);
  }

// Speicherverwaltung fuer gemeinsames Segment einrichten
  shm_free = (char **) local_shm;
  *shm_free = local_shm + 4;

  int j, r, k;
  void task(int);

// Allen Prozessen gemeinsame Datenstrukturen anlegen und vorbesetzen
  interruptible = (int *)(local_shm + 4);
  *shm_free += 4;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
  clock_init(interruptible);

  allnodes
    = new shared_flags[NUMBER_OF_PROCESSES];
  for (i = 0; i < NUMBER_OF_PROCESSES; i++)
    allnodes[i].init(LogP);
  for (i = 0; i < NUMBER_OF_PROCESSES; i++) {
    for (j = 0; j < NUMBER_OF_PROCESSES; j++) {
      for (k = 0; k < LogP; k++) {
        if (j == (i + (1 << k))
                  % NUMBER_OF_PROCESSES) {
          for (r = 0; r < 2; r++) {
            (allnodes[i].partnerflags[r][k]).ptr
              = &(allnodes[j].myflags[r][k]);
          }
        }
      }
    }
  }
  for (int lfd_nummer = 0;
       lfd_nummer < NUMBER_OF_PROCESSES; lfd_nummer++) {
    if (fork() == 0) {
      meine_prozessnummer = lfd_nummer;
      task(meine_prozessnummer);
      exit(0);
    }
  }
  for (i = 0; i < NUMBER_OF_PROCESSES; i++) {
    wait(&status);
  }
  cout << "\nAlle Prozesse sind beendet\n" << flush;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
void task(int arg)
{ int i;
  private_int parity;
  private_boolean sense;
  shared_flags *localflags = &(allnodes[arg]);
  int round = 0;
  parity.value = 0;
  sense.value = true;
  for(i = 0; i < 10; i++) {
    round++;
    cout << "process " << arg
         << " begins round " << round
         << "\n" << flush;
    spinning(1);
    dissemination_barrier(localflags,
                          &sense,
                          &parity,
                          arg,
                          round);
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
Beginn des Wurzelprogramms
process 0 begins round 1
process 0: round = 1, dissemination round = 0
process 1 begins round 1
process 1: round = 1, dissemination round = 0
process 1: round = 1, dissemination round = 1
process 2 begins round 1
process 2: round = 1, dissemination round = 0
process 2: round = 1, dissemination round = 1
process 3 begins round 1
process 3: round = 1, dissemination round = 0
process 3: round = 1, dissemination round = 1
process 3: round = 1, dissemination round = 2
process 4 begins round 1
process 4: round = 1, dissemination round = 0
process 4: round = 1, dissemination round = 1
process 4: round = 1, dissemination round = 2
process 3 begins round 2
process 3: round = 2, dissemination round = 0
process 1: round = 1, dissemination round = 2
process 0: round = 1, dissemination round = 1
process 0: round = 1, dissemination round = 2
process 0 begins round 2
process 0: round = 2, dissemination round = 0
process 4 begins round 2
process 4: round = 2, dissemination round = 0
process 4: round = 2, dissemination round = 1
process 2: round = 1, dissemination round = 2
process 2 begins round 2
process 2: round = 2, dissemination round = 0
process 1 begins round 2
process 1: round = 2, dissemination round = 0
process 1: round = 2, dissemination round = 1
process 1: round = 2, dissemination round = 2
process 0: round = 2, dissemination round = 1
process 3: round = 2, dissemination round = 1
process 3: round = 2, dissemination round = 2
process 2: round = 2, dissemination round = 1
process 2: round = 2, dissemination round = 2
process 2 begins round 3
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

❑ Tree-Based Barrier

```
#define NUMBER_OF_PROCESSES 5

#include <iostream.h>
#include <sys/types.h>
union semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
};
#include <sys/ipc.h>

#include <sysent.h>
extern "C" void exit(int);
#include <sys/wait.h>

#include <errno.h>
extern char* sys_errlist[];

// Definitionen fuer gemeinsame Segmente
#include <sys/shm.h>
extern "C" int shmctl(int, int, struct shmid_ds *);
extern "C" int shmget(key_t, unsigned int, int);
extern "C" void *shmat(int, const void *, int);
char **shm_free;

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);

int *interruptible;

void begin_of_atomic()
{ clock_lock(interruptible);
}

void end_of_atomic()
{ clock_unlock(interruptible);
}

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


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
typedef int boolean;

class shared_treenode {
  public:
    boolean parentsense;
    boolean *parentpointer;
    boolean *childpointers[2];
    boolean havechild[4];
    boolean childnotready[4];
    void *operator new(size_t s)
      { char *tmp = *shm_free;
        *shm_free += s;
        return tmp;
      };
    void *operator new[](size_t s)
      { char *tmp = *shm_free;
        *shm_free += s;
        return tmp;
      };
};

shared_treenode *nodes;
boolean dummy;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
void tree_barrier(int vpid, boolean *sense)
{ int i;
  shared_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);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
int main(int argc, char **argv)
{ cout << "Beginn des Wurzelprogramms\n" << flush;

  int i, meine_prozessnummer, shmid, status;
  char *local_shm;

// Einrichten eines `shared memory'-Segments
//    1. Tilgen, falls schon vorhanden
  shmid = shmget(1, 1, 0600 | IPC_CREAT);
  if ((status = shmctl(shmid, IPC_RMID, 0)) == -1) {
    cout << "***** Fehler beim Tilgen des gmeinsamen Sgementes: "
         << sys_errlist[errno] << "\n";
  }
//    2. Neu einrichten
  if ((shmid = shmget(1, 10000, 0600 | IPC_CREAT)) < 0) {
    cerr << "\n\nError in shmget for shared memory: "
         << sys_errlist[errno]
         << "\n" << flush;
    exit(0);
  }
//    3. Zugaenglich machen
  if ((int)(local_shm = (char *)shmat(shmid, (void *)0, 0))
      == -1) {
    cerr << "\n\nError: " << sys_errlist[errno] << "\n" << flush;
    exit(0);
  }

// Speicherverwaltung fuer gemeinsames Segment einrichten
  shm_free = (char **) local_shm;
  *shm_free = local_shm + 4;

  int j, r, k;
  void task(int);

// Allen Prozessen gemeinsame Datenstrukturen anlegen und vorbesetzen
  interruptible = (int *)(local_shm + 4);
  *shm_free += 4;
  clock_init(interruptible);
   nodes
     = new shared_treenode[NUMBER_OF_PROCESSES];

  void task(int);
  int pid;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

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

```
for (int lfd_nummer = 0;
     lfd_nummer < NUMBER_OF_PROCESSES; lfd_nummer++) {
  if (fork() == 0) {
    meine_prozessnummer = lfd_nummer;
    task(meine_prozessnummer);
    exit(0);
  }
}
for (i = 0; i < NUMBER_OF_PROCESSES; i++) {
  wait(&status);
```


© 1996, Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig


```
  }
  cout << "\nAlle Prozesse sind beendet\n" << flush;
}

void task(int arg)
{ boolean sense = true;
  int i, round = 0;
  for(i = 0; i < 10; i++) {
    round++;
    cout << "process " << arg << " begins round " << round
         << "\n" << flush;
    spinning(1);
    tree_barrier(arg, &sense);
  }
}
```


© 1996, Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig