Shared Memory Programming Models III

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

Shared Memory Programming Models III

Communication by shared memory

- Semaphore rep.
- Reader-Writer problem
- PThreads
- Active Objects

Semaphore

A semaphore is an abstraction of a synchronisation variable, that enables the elegant solution of multiple of synchronisation problems

Up-to-now all programs have used *active waiting*. This is very inefficient under quasi-parallel processing of multiple processes on one processor (multitasking). The semaphore enables to switch processes into an idle state.

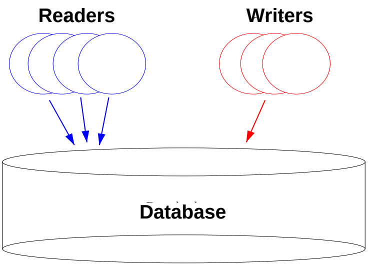
We understand a semaphore as abstract data type: Data structure with operations, that fulfill particular properties:

A semaphore S has a non-negative integer value *value*(S), that is assigned during creation of the Semaphore with the value *init*.

For a semaphore S two operations **P**(S) and **V**(S) are defined with:

- **P**(S) decrements the value of S by one if *value*(S) > 0, otherwise the process *blocks* as long as another process executes a **V**operation on S.
- **V**(S) frees another process from a **P**operation if one is waiting (are several waiting one is selected), otherwise the value of S is incremented by one. **V**operations never block!

Readers/Writers Problem



Two classes of processes, readers and writers, access a common database. Readers perform transactions, that are not modifying the database. Writers change the database and need to have exclusive access. If no writer has access an arbitrary number of readers can access simultaneously.

Problems:

- Deadlock-free coordination of processes
- Fairness: Final entry of writers

Naive Readers/Writers

Two Semaphores:

- *rw*: How has access to the database the readers/the writers
- *mutexR*: Protection of the writer counter *nr*

Program (Reader–Writer–Problem, first solution)

```
parallel readers–writers–1
{
    const int m = 8, n = 4;                  // number of readers and writers
    Semaphore rw=1;                          // Access onto database
    Semaphore mutexR=1;                      // Protect reader count
    int nr=0;                                // Count of accessing readers

    process Reader [int i ∈ {0, ..., m − 1}] {
        while (1) {
            P(mutexR);                       // Access reader counter
            nr = nr+1;                       // A further reader
            if (nr==1) P(rw);                // First is waiting for DB
            V(mutexR);                       // next reader can get in
            read database;
            P(mutexR);                       // Access reader counter
            nr = nr-1;                       // A reader fewer
            if (nr==0) V(rw);                // Last releases access to DB
            V(mutexR);                       // next reader can enter
        }
    }

}
```

Naive Readers/Writers

Program (Reeder–Writer–Problem, first solution cont.)

```
parallel process
{
    Writer [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(rw);                          // Access onto DB
            write database;
            V(rw);                          // Release DB
        }
    }
}
```

Solution is not fair: Writers can starve

Fair Readers/Writers

Schedule waiting processes according to FCFS in a waiting queue

Variable:

- *nr*, *nw*: Number of *active* readers/writers ($nw \leq 1$)
- *dr*, *dw*: Number of *waiting* readers/writers
- *buf*, *front*, *rear*: Waiting queue
- Semaphore *e*: Protection of waiting queue state
- Semaphore *r*, *w*: Waiting of readers/writers

Program (Reader–Writer–Problem, fair solution)

```
parallel readers–writers–2
{
    const int m = 8, n = 4;                    // Number of readers and writers
    int nr=0, nw=0, dr=0, dw=0;                // State
    Semaphore e=1;                             // Access onto waiting queue
    Semaphore r=0;                             // Delay of readers
    Semaphore w=0;                             // Delay of writers
    const int reader=1, writer=2;              // Marks
    int buf[n + m];                            // Who waits?
    int front=0, rear=0;                       // Pointer

}
```

Fair Readers/Writers

Program (Reader–Writer–Problem, fair Solution cont1.)

```
parallel readers–writers–2 cont1.
{
    int wake_up (void)                              // May be excuted by exactly one!
    {
        if (nw==0 ∧ dr>0 ∧ buf[rear]==reader)
        {
            dr = dr-1;
            rear = (rear+1) mod (n + m);
            V(r);
            return 1;                               // Have awaked a reader
        }
        if (nw==0 ∧ nr==0 ∧ dw>0 ∧ buf[rear]==writer)
        {
            dw = dw-1;
            rear = (rear+1) mod (n + m);
            V(w);
            return 1;                               // Have awaked a writer
        }
        return 0;                                   // Have awaked noone
    }

}
```

Fair Readers/Writers

Program (Reader–Writer–Problem, fair Solution cont2.)

```
parallel readers–writers–2 cont2.
{

    process Reader [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            P(e);                                   // want to change state
            if(nw>0 ∨ dw>0)
            {
                buf[front] = reader;                // in waiting queue
                front = (front+1) mod (n + m);
                dr = dr+1;
                V(e);                               // free state
                P(r);                               // wait until readers can continue
                                                    // here is e = 0 !
            }
            nr = nr+1;                              // here is only one
            if (wake_up()==0)                       // can one be awaked?
                V(e);                               // no, set e = 1

            read database;

            P(e);                                   // want to change state
            nr = nr-1;
            if (wake_up()==0)                       // can one be awaked?
                V(e);                               // no, set e = 1
        }
```

Fair Readers/Writers

Program (Reader–Writer–Problem, fair solution cont3.)

```
parallel readers–writers–2 cont3.
{

    process Writer [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(e);                               // want to change state
            if(nr>0 ∨ nw>0)
            {
                buf[front] = writer;            // in waiting queue
                front = (front+1) mod (n + m);
                dw = dw+1;
                V(e);                           // free state
                P(w);                           // wait until it is its turn
                                                // here is e = 0 !
            }
            nw = nw+1;                          // here is only one
            V(e);                               // here needs noone to be waked

            write database;                     // exclusive access

            P(e);                               // want to change state
            nw = nw-1;
            if (wake_up()==0)                   // can one be awaked?
                V(e);                           // no, set e = 1
        }
    }
}
```

Processes and Threads

A Unix process has

- IDs (process, user, group)
- Environment variables
- Directory
- Program code
- Register, stack, heap
- File descriptors, signals
- Message queues, pipes, shared memory segments
- Shared libraries

Each process owns its individual address space

Threads exist within a single process

Threads share an address space

A thread consists of

- ID
- Stack pointer
- Registers
- Scheduling properties
- Signals

Creation and switching times are shorter

„Parallel function“

PThreads

- Each manufacturer had an own implementation of threads or „light weight processes“
- 1995: IEEE POSIX 1003.1c Standard (there are several „drafts“)
- Standard document is liable to pay costs
- Defines threads in a portable way
- Consists of C data types and functions
- Header file `pthread.h`
- Library name is not normed. In Linux `-lpthread`
- Compilation in Linux: gcc <file> -lpthread

PThreads Overview

There are 3 functional groups
All names start with `pthread_`

- `pthread_`
  Thread management and other routines
- `pthread_attr_`
  Thread attribute objects
- `pthread_mutex_`
  All that has to do with mutex variables
- `pthread_mutex_attr_`
  Attributes for mutex variables
- `pthread_cond_`
  Condition variables
- `pthread_cond_attr_`
  Attributes for condition variables

Creation of Threads

- `pthread_t` : Data type for a thread.
- Opaque type: Data type is defined in the library and is processed by its functions. Contents is implementation dependent.
- `int pthread_create(thread,attr,start_routine,arg)` : Starts the function `start_routine` as thread.
  - `thread` : Pointer onto a `pthread_t` structure. Serves for identification of a thread.
  - `attr` : Thread attributes are explained below. Default is `NULL`.
  - `start_routine`: Pointer onto a function of type `void* func (void*);`
  - `arg` : `void*` pointer that is passed as function argument.
  - Return value that is larger than zero indicates an error.
- Threads can start further threads, maximal count of threads is implementation dependent

Termination of Threads

- There are the following possiblities to terminate a thread:
  - The thread finishes its `start_routine()`
  - The thread calls `pthread_exit()`
  - The thread is terminated by another thread via `pthread_cancel()`
  - The process is terminated by `exit()` or the end of the `main()` function
- `pthread_exit(void* status)`
  - Finishes the calling thread. Pointer is stored and can be queried with `pthread_join` (see below) (Return of results).
  - If `main()` calls this routine existing threads continue and the process is not terminated.
  - Existing files, that are opened, are not closed!

Waiting for Threads

- Peer model: Several equal threads perform a collective task. Program is terminated if all threads are finished
- Requires waiting of a thread until all others are finished
- This is a kind of synchronisation
- `int pthread_join(pthread_t thread, void **status);`
  - Waits until the specified thread terminates itself
  - The thread can return via `pthread_exit()` a `void*` pointer
  - Is the status parameter choosen as `NULL`, the return value is obsolete

Thread Management Example

```
#include <pthread.h>     /* for threads     */

void* prod (int *i) { /* Producer thread */
  int count=0;
  while (count<100000) count++;
}

void* con (int *j) { /* Consumer thread */
  int count=0;
  while (count<1000000) count++;
}

int main (int argc, char *argv[]) { /* main program */
  pthread_t thread_p, thread_c; int i,j;

  i = 1; pthread_create(&thread_p,NULL,(void*(*)(void*)) prod,(void *) &i);
  j = 1; pthread_create(&thread_c, NULL,(void*(*)(void*)) con, (void *) &j);

  pthread_join(thread_p, NULL); pthread_join(thread_c, NULL);
  return(0);
}
```

Passing of Arguments

- Passing of multiple arguments requires the definition of an individual data type:

```
struct argtype {int rank; int a,b; double c;};
struct argtype args[P];
pthread_t threads[P];

for (i=0; i<P; i++) {
  args[i].rank=i; args[i].a=...
    pthread_create(threads+i,NULL,(void*(*)(void*)) prod,(void *)args+i);
}
```

- The following example contains two errors:

```
pthread_t threads[P];
for (i=0; i<P; i++) {
  pthread_create(threads+i,NULL,(void*(*)(void*)) prod,&i);
}
```

  - Contents of `i` is eventually changed before the thread reads it
  - If `i` is a stack variable it exists eventually no more

Thread Identifiers

- `pthread_t pthread_self(void);`
  Returns the own thread-ID
- `int pthread_equal(pthread_t t1, pthread_t t2);`
  Returns true (value>0) if the two IDs are identical
- Concept of an „opaque data type“

Join/Detach

- A thread within state `PTHREAD_CREATE_JOINABLE` releases its resources only, if `pthread_join` has been executed.
- A thread in state `PTHREAD_CREATE_DETACHED` releases its resources as soon a it is terminated. In this case `pthread_join` is not allowed.
- Default is `PTHREAD_CREATE_JOINABLE`, but that is not implemented in all libraries.
- Therefore better:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
int rc = pthread_create(&t,&attr,(void*(*)(void*))func,NULL);
....
pthread_join(&t,NULL);
pthread_attr_destroy(&attr);
```

- Provides example for application of attributes

Mutex Variables

- Mutex variables realize mutual exclusion within PThreads
- Creation and initialisation of a mutex variable

  ```
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex,NULL);
  ```

  Mutex variable is in state free
- Try to enter the critical section (blocking):

  ```
  pthread_mutex_lock(&mutex);
  ```
- Leave critical section

  ```
  pthread_mutex_unlock(&mutex);
  ```
- Release resource of the mutex variable

  ```
  pthread_mutex_destroy(&mutex);
  ```

Condition Variables

- Condition variables enable *inactive* waiting of a thread until a certain condition has arrived.
- Simplest example: Flag variables (see example below)
- To a condition synchronisation belong *three* things:
  - A variable of type `pthread_cond_t`, that realizes inactive waiting.
  - A variable of type `pthread_mutex_t`, that realizes mutual exclusion during condition change.
  - A global variable, which value enables the calculation of the condition

Condition Variables: Creation/Deletion

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);`
  initializes a condition variable
  In the simplest case: `pthread_cond_init(&cond,NULL)`
- `int pthread_cond_destroy(pthread_cond_t *cond);`
  the resources of a condition variable is released

Condition Variables: Wait

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  blocks the calling thread until for the condition variable the function `pthread_signal()` is called
- When calling the `pthread_wait()` the thread has to be the owner of the lock
- `pthread_wait()` leaves the lock and waits for the signal in an atomic way
- After returning from `pthread_wait()` the thread is again the owner of the lock
- After return the condition has not to be true in any case
- With a single condition variable one should only use exactly one lock

Condition Variables: Signal

- `int pthread_cond_signal(pthread_cond_t *cond);`
  Awakes a thread that has executed a `pthread_wait()` onto a condition variable. If noone waits the function has no effect.
- When calling the thread should be owner of the associated lock.
- After the call the lock should be releases. First the release of the lock allows the waiting thread to return from `pthread_wait()` function.
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  awakes *all* threads that have executed a `pthread_wait()` on the condition variable. These then apply for the lock.

Condition Variables: Ping-Pong Example

```
#include<stdio.h>
#include<pthread.h>     /* for threads     */

int arrived_flag=0,continue_flag=0;
pthread_mutex_t arrived_mutex, continue_mutex;
pthread_cond_t arrived_cond, continue_cond;

pthread_attr_t attr;

int main (int argc, char *argv[])
{
  pthread_t thread_p, thread_c;

  pthread_mutex_init(&arrived_mutex,NULL);
  pthread_cond_init(&arrived_cond,NULL);
  pthread_mutex_init(&continue_mutex,NULL);
  pthread_cond_init(&continue_cond,NULL);
```

Example cont. I

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
                            PTHREAD_CREATE_JOINABLE);

pthread_create(&thread_p,&attr,
               (void*(*)(void*)) prod,NULL);
pthread_create(&thread_c,&attr,
               (void*(*)(void*)) con ,NULL);

pthread_join(thread_p, NULL);
pthread_join(thread_c, NULL);

pthread_attr_destroy(&attr);

pthread_cond_destroy(&arrived_cond);
pthread_mutex_destroy(&arrived_mutex);
pthread_cond_destroy(&continue_cond);
pthread_mutex_destroy(&continue_mutex);

return(0);
```

Example cont. II

```
void prod (void* p) /* Producer thread */
{
  int i;
  for (i=0; i<100; i++) {
    printf("ping\n");

    pthread_mutex_lock(&arrived_mutex);
    arrived_flag = 1;
    pthread_cond_signal(&arrived_cond);
    pthread_mutex_unlock(&arrived_mutex);

    pthread_mutex_lock(&continue_mutex);
    while (continue_flag==0)
      pthread_cond_wait(&continue_cond,&continue_mutex);
    continue_flag = 0;
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

Example cont. III

```
void con (void* p) /* Consumer thread */
{
  int i;
  for (i=0; i<100; i++) {
    pthread_mutex_lock(&arrived_mutex);
    while (arrived_flag==0)
      pthread_cond_wait(&arrived_cond,&arrived_mutex);
    arrived_flag = 0;
    pthread_mutex_unlock(&arrived_mutex);

    printf("pong\n");

    pthread_mutex_lock(&continue_mutex);
    continue_flag = 1;
    pthread_cond_signal(&continue_cond);
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

Thread Safety

- Hereby is understood whether a function/library can be used by multiple threads at the same time.
- A function is *reentrant* if it may be called by several threads synchronously.
- A function, that does not use a global variable, is reentrant
- The runtime system has to use shared resources (e.g. the stack) under mutual exclusion
- The GNU C compiler has to be configured for compilation with an appropriate thread model. With `gcc -v` you can see the type of thread model.
- STL: Allocation is thread save, access of multiple threads onto a single container has to be protected by the user.

Threads and OO

- Obviously are PThreads relatively impractical to code.
- Mutexes, conditional variables, flags and semaphores should be realized in an object-oriented way. Complicated `init/destroy` calls can be hidden in constructors/destructors.
- Threads are transformed into Active Objects.
- An active object „is executed“ independent of other objects.

Active Objects

```
class ActiveObject
{
public:
  //! constructor
  ActiveObject ();

  //! destructor waits for thread to complete
  ~ActiveObject ();

  //! action to be defined by derived class
  virtual void action () = 0;

protected:
  //! use this method as last call in constructor of derived class
  void start ();

  //! use this method as first call in destructor of derived class
  void stop ();

private:
    ...
};
```

Active Objects cont. I

```
#include<iostream>
#include"threadtools.hh"

Flag arrived_flag,continue_flag;

int main (int argc, char *argv[])
{
  Producer prod; // start prod as active object
  Consumer con;  // start con as active object

  return(0);
} // wait until prod and con are finished
```

Active Objects cont. II

```
class Producer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Producer () {
    this->start();
  }

  // execute action
  virtual void action () {
    for (int i=0; i<100; i++) {
      std::cout « "ping" « std::endl;
      arrived_flag.signal();
      continue_flag.wait();
    }
  }

  // destructor waits for end of action
  ~Producer () {
    this->stop();
  }
};
```

Active Objects cont. III

```
class Consumer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Consumer () {
    this->start();
  }

  // execute action
  virtual void action () {
    for (int i=0; i<100; i++) {
      arrived_flag.wait();
      std::cout « "pong" « std::endl;
      continue_flag.signal();
    }
  }

  // destructor waits for end of action
  ~Consumer () {
    this->stop();
  }
};
```

Links

1. PThreads tutorial from LLNL
   http://www.llnl.gov/computing/tutorials/pthreads/
2. Linux Threads Library
   http://pauillac.inria.fr/∼xleroy/linuxthreads/
3. Thread safety of GNU standard library
   http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/howto.html#3
4. Resources for PThreads Functions
   http://as400bks.rochester.ibm.com/iseries/v5r1/ic2924/index.htm?info/apis/rzah4mst.htm