Shared Memory Programming Models II

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

Parallel Programming Models II

Communication using shared memory

- Barrier – synchronization of all processes
- Semaphores
- Philosphers problem

Global Synchronization

- *Barrier*: All processors shall wait on each other until all have arrived
- Barriers are often repeatedly executed repeatedly:

```
while (1) {
    a calculation;
    Barrier;
}
```

- Since the calculation is load balanced, all arrive simultanously at the barrier
- First idea: Count all arriving processes

Global Synchronization

Program (First proposal of a barrier)

parallel *barrier-1*
{
const int *P=8;* **int** *count=0;* **int** *release=0;*

process Π [**int** $p \in \{0, ..., P-1\}$]
{
while *(1)*
{
calculation;
CSenter; *// entry*
if *(count==0) release=0;* *// reset*
count=count+1; *// increment counter*
CSexit; *// exit*
if *(count==P) {*
count=0; *// last resets counter*
release=1; *// and frees*
}
else **while** *(release==0) ;* *// waiting*
}
}

Barrier with Sense Reversal

Wait *reversible* for *release*==1 and *release*==0

Program (Barrier with direction reversal)

```
parallel sense-reversing-barrier
{
    const int P=8;      int count=0;      int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int local_sense = release;
        while (1)
        {
            calculation;
            local_sense = 1-local_sense;          // change direction
            CSenter;                              // entry
            count=count+1;                        // increment counter
            CSexit;                               // exit
            if (count==P) {
                count=0;                          // last resets
                release=local_sense;              // and frees
            } else
                while (release≠local_sense) ;
        }
    }
}
```

Complexity is $O(P^2)$ since all P processes have to pass through a critical section at a time. Is there a better approach?

Hierarchical Barrier: Variant 1

In the barrier with counter all P processes have to pass through a critical section. This necessitates $O(P^2)$ memory accesses. We now develop a solution with $O(P \log P)$ accesses.

We start with *two* processes and consider the following program segment:

int *arrived*=0, *continue*=0;

Π_0:	Π_1:
	arrived=1;
while ($\neg$*arrived*) ;	
arrived=0;	
continue=1;	
	while ($\neg$*continue*) ;
	continue=0;

We use two synchronization variables, so called *flags*

Hierarchical Barrier: Variant 1

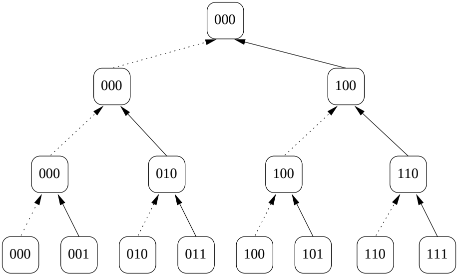
When using flags the following rules have to be met:

1. The process, that waits for a flag, also resets it.
2. A flag may first be newly set, if it has been savely reset.

Both rules are respected by our solution.

The solution assumes sequential consistency of the memory!

We now apply this idea in a hierarchical way:



Hierarchical Barrier: Variant 1

Program (Barrier with tree)

parallel *tree-barrier*
{
  const int $d = 4$, $P = 2^d$; **int** *arrived[P]=*$\{0[P]\}$, *continue[P]=*$\{0[P]\}$*;*

  process Π [**int** $p \in \{0, ..., P-1\}$]
  {
    int *i, r, m, k;*
    while *(1) {*
      calculation;
      for $(i = 0; i < d; i{+}{+})$ *{* *// upward*
        $r = p \;\&\; \left[\sim\left(\sum_{k=0}^{i} 2^k\right)\right];$ *// reset bits 0 to i*
        $m = r \mid 2^i;$ *// set bit i*
        if $(p == m)$ *arrived[m]=1;*
        if $(p == r)$ *{*
          while$(\neg$*arrived[m])* *;* *// wait*
          arrived[m]=0;
        }
      } *// process 0 knows that all are there*
      . . .

Hierarchical Barrier: Variant 1

Program (Barrier with tree cont.)

parallel *tree-barrier cont.*

{

. . .

for $(i = d - 1; i \geq 0; i - -)$ { // *downward*

$r = p \ \& \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right];$ // *reset bits 0 to i*

$m = r \mid 2^i;$

if $(p == m)$ {

while(¬*continue[m]*) ;

continue[m]=0;

}

if $(p == r)$ *continue[m]=1;*

}

}

}

}

Caution: Flag variables should be stored in different cache lines, that accesses do not hinder themselves!

Hierarchical Barrier: Variant 2

This variant presents a symmetric solution of the barrier with *recursive doubling*.

We consider at first again the barrier for two processes Π_i and Π_j:

Π_i:	Π_j:
while (*arrived*[*i*]) ;	**while** (*arrived*[*j*]) ;
arrived[*i*]=1;	*arrived*[*j*]=1;
while (¬*arrived*[*j*]) ;	**while** (¬*arrived*[*i*]) ;
arrived[*j*]=0;	*arrived*[*i*]=0;

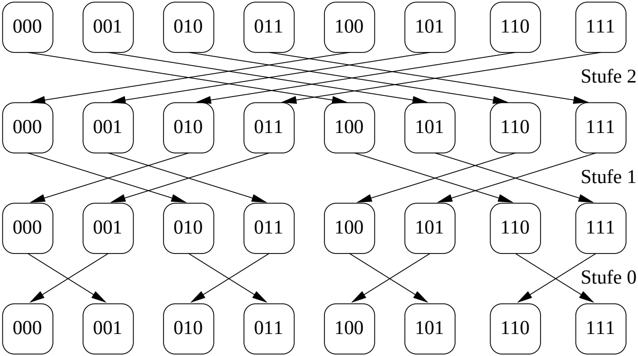
As prerequisite for the general solution the flags are organized as arrays, in the beginning all flags are 0.

Sequence in words:

- Line 2: Each sets *its* flag to 1
- Line 3: Each waits onto the flag of the other
- Line 4: Each resets the flag of the *other*
- Line 1: Because of rule 2 from above wait until the flag is reset
- Now we use this ideas in a recursive manner!

Hierarchical Barrier: Variant 2

Recursive doubling uses the following communication structure:



- No idle processors
- Each step is a two way communication

Hierarchical Barrier: Variant 2

Program (Barrier with recursive doubling)

```
parallel recursive-doubling-barrier
{
    const int d = 4,      P = 2^d; int arrived[d][P]={0[P · d]};

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, q;
        while (1) {
            calculation;
            for (i = 0; i < d; i++)                 // all steps
            {
                q = p ⊕ 2^i;                         // reverse bit i
                while (arrived[i][p]) ;
                arrived[i][p]=1;
                while (¬arrived[i][q]) ;
                arrived[i][q]=0;
            }
        }
    }
}
```

Semaphore

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

Semaphore

Is the number of *successfully finished* **P** operations n_P and the one of **V** operations n_V, then for the value of the semaphore applies always:

$$value(S) = n_V + init - n_P \geq 0$$

or equivalent $n_P \leq n_V + init$.

The value of a semaphore is *not* visible from the outside. It shows only by the executability of the **P** operation.

The increment resp. decrement of a semaphore is performed in an atomic way, multiple processes can also perform **P**/**V** operations concurrently.

Semaphores, that can take a value larger than one, are called *general semaphores*.

Semaphores, that only have values $\{0, 1\}$, are called *binary semaphores*.

Notation:

Semaphore *S*=1;
Semaphore *forks*[5] = {1 [5]};

Mutual Exclusion with Semaphore

We now present in which way all already treated synchronisation problems can be solved with semaphore variables. The first application is dedicated to mutual exclusion by usage of a single binary semaphore:

Program (Mutual exclusion with semaphore)

parallel *cs-semaphore*
{
    const int *P=8;*
    Semaphore *mutex=1;*
    process Π [**int** $i \in \{0, ..., P-1\}$]
    {
        while *(1)*
        {
            P*(mutex);*
            critical section;
            V*(mutex);*
            uncritical section;
        }
    }
}

Mutual Exclusion with Semaphore

By multitasking processes can be switched to the idle state (waiting).

Fairness is easy to integrate into the wake-up mechanism (FCFS).

Memory consistency model can be respected by the implementation, programs remains portable (e. g. Pthreads)

Barrier with Semaphore

- Each process has to be delayed until the other(s) arrive at the barrier.
- The barrier has to be reusable, since it is usually executed several times.

Program (Barrier with semaphore for two processes)

```
parallel barrier-2-semaphore
{
    Semaphore b1=0, b2=0;
    process Π1
    {
        while (1) {
            calculation;
            V(b1);
            P(b2);
        }
    }
    process Π2
    {
        while (1) {
            calculation;
            V(b2);
            P(b1);
        }
    }
}
```

Barrier with Semaphore

After unrolling of the loop, the code looks as follows:

Π_1:

```
    calculation 1;
    V(b1);
    P(b2);
    calculation 2;
    V(b1);
    P(b2);
    calculation 3;
    V(b1);
    P(b2);
    ...
```

Π_2:

```
    calculation 1;
    V(b2);
    P(b1);
    calculation 2;
    V(b2);
    P(b1);
    calculation 3;
    V(b2);
    P(b1);
    ...
```

Assume process Π_1 works in calculation phase i, thus it has executed **P**(*b2*) $i-1$-times. Assume further Π_2 *works in* calculation phase $j < i$, therefore it has executed **V**(*b2*) $j-1$. Then holds

$$n_P(b2) = i - 1 > j - 1 = n_V(b2).$$

On the other hand the semaphore rules assure, that

$$n_P(b2) \leq n_V(b2) + 0.$$

This is a contradiction and it can not apply $j < i$. The argument is symmetric and applies also when the processor numbers are exchanged.

Producer/Consumer $m/n/1$

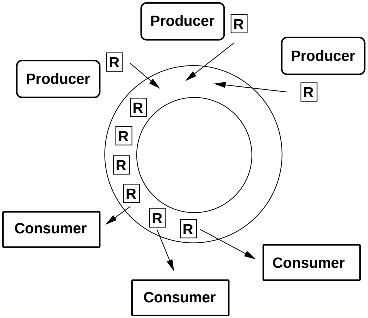
m producers, *n* consumers, 1 buffer location,

Producer has to block if the buffer location is occupied.

Consumer has to block if no request is stored.

We use two semaphores:

- *empty*: counts number of *free* buffer locations
- *full*: counts number of *occupied* locations (requests)



Produce/Consumer $m/n/1$

Program (m producer, n consumer, 1 buffer location)

```
parallel prod-con-nm1
{
    const int m = 3, n = 5;
    Semaphore empty=1;                          // free buffer location
    Semaphore full=0;                           // available request
    T buf;                                      // the buffer
    process P [int i ∈ {0, ..., m − 1}] {
        while (1) {
            Generate request t;
            P(empty);                           // Is buffer free?
            buf = t;                            // store request
            V(full);                            // request available
        }
    }
    process C [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(full);                            // Is request available?
            t = buf;                            // remove request
            V(empty);                           // buffer is empty
            Process request t;
        }
    }
}
```

Shared binary semaphore (*split binary semaphore*):

$$0 \leq empty + full \leq 1 \quad \text{(invariant)}$$

Producer/Consumer $1/1/k$

1 producer, 1 consumer, k buffer locations,

Buffer is array of length k of type T. Insertion and deletion works with

$$buf[front] = t; \quad front = (front + 1) \mod k;$$

$$t = buf[rear]; \quad rear = (rear + 1) \mod k;$$

Semaphore as above, only initialized with k!

Program (1 producer, 1 consumer, k buffer locations)

```
parallel prod-con-11k
{
    const int k = 20;
    Semaphore empty=k;         // counts free buffer locs
    Semaphore full=0;          // count avaiable requests
    T buf[k];                  // the buffer
    int front=0;               // newest request
    int rear=0;                // oldest request

}
```

Producer/Consumer $1/1/k$

Program (1 producer, 1 consumer, *k* buffer locations)

```
parallel prod-con-11k
{
    process P {
        while (1) {
            Generate request t;
            P(empty);                      // Is buffer free?
            buf[front] = t;                // store request
            front = (front+1) mod k;       // next free location
            V(full);                       // request available
        }
    }
    process C {
        while (1) {
            P(full);                       // Is request there?
            t = buf[rear];                 // remove request
            rear = (rear+1) mod k;         // next request
            V(empty);                      // buffer is free
            Process request t;
        }
    }
}
```

Producer/Consumer $m/n/k$

m producers, *n* consumers, *k* buffer locations,

We only have to ensure, that producers among each other and consumers cannot manipulate the buffer at the same time.

Use two additional binary semaphores *mutexP* und *mutexC*

Program (*m* producer, *n* consumer, *k* buffer locations)

```
parallel prod-con-mnk
{
    const int k = 20, m = 3, n = 6;
    Semaphore empty=k;              // count free buffer locations
    Semaphore full=0;               // count available requests
    T buf[k];                       // the buffer
    int front=0;                    // newest request
    int rear=0;                     // oldest request
    Semaphore mutexP=1;             // access of producers
    Semaphore mutexC=1;             // access of consumersr

}
```

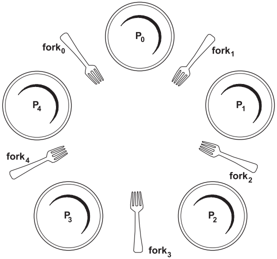
Producer/Consumer $m/n/k$

Program (m producer, n consumer, k buffer locations)

```
parallel process
{
    P [int i ∈ {0, ..., m − 1}] {
        while (1) {
            Generate request t;
            P(empty);                      // Is buffer free?
            P(mutexP);                     // manipulate buffer
            buf[front] = t;                // store request
            front = (front+1) mod k;       // next free position
            V(mutexP);                     // ready with buffer
            V(full);                       // request available
        }
    }
    process C [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(full);                       // Is request there?
            P(mutexC);                     // manipulate buffer
            t = buf[rear];                 // remove request
            rear = (rear+1) mod k;         // next request
            V(mutexC);                     // ready with buffer
            V(empty);                      // buffer is free
            Process request t;
        }
    }
}
```

Dining Philosophers

Complex synchronisation task: A process necessitates exclusive access onto several ressources to perform a specific task.
→ overlapping critical sections.



Five philosophers sit at a round table. The exercise of each philosopher consists out of interchanging phases of thinking and eating. In between two of the philosophers a fork is positioned and in the center of the table a mountain Spaghetti is located. To eat a philosopher needs two forks – the one laying left and *right next to him.*

Dining Philosophers

The problem:
Write a parallel program, with one process per philosopher, that

- enables a maximal count of philosophers to eat and
- that avoids a deadlock.

Skeletal structure of a philosopher:

```
while (1)
{
    think;
    take forks;
    eat;
    lay back forks;
}
```

Naive Philosophers

Program (Naive solution of the philosophers problem)

parallel *philosophers–1*
{
    const int $P = 5$*;* // *number of philosophers*
    Semaphore *forks[P] = { 1 [P] };* // *forks*

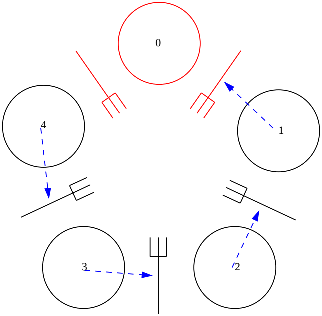
    process *Philosopher* [**int** $p \in \{0, ..., P-1\}$] *{*
        while *(1) {*
            Thinking;
            P*(fork[p]);* // *left fork*
            P*(fork[*$(p+1)$ *mod P]);* // *right fork*
            Eating;
            V*(fork[p]);* // *left fork*
            V*(fork[*$(p+1)$ *mod P]);* // *right fork*
        }
    }
}

Naive Philosophers

Philosophers are deadlocked, if all take at first the left fork!

Simple solution of the deadlock problem: Avoid cyclic dependencies,
e. g. philosopher 0 takes his forks in a different sequence right then left.

This solution allows eventually not maximal concurrency:



Clever Philosophers

Take forks only, when *both* are available

Critical section: only one can manipulate the forks

Three states of a philosopher: thinking, hungry, eating

Program (Solution of philosophers problem)

```
parallel philosophers–2
{
    const int P = 5;                              // count philosophers
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };                   // eating philosopher
    int state[P] = { think [P] };                 // state



}
```

Clever Philosophers

Program (Soluton of philosophers problem)

```
parallel process
{
    Philosopher [int p ∈ {0, ..., P − 1}] {
        void test (int i) {
            int l=(i + P − 1) mod P, r=(i + 1) mod P;
            if (state[i]==hungry ∧ state[l]≠eat ∧ state[r]≠eat)
            {
                state[i] = eat;
                V(s[i]);
            }
        }

        while (1) {
            Thinking;
            P(mutex);                          // take forks
            state[p] = hungry;
            test(p);
            V(mutex);
            P(s[p]);                           // wait, if neighbor eats
            Eating;
            P(mutex);                          // lay forks downs
            state[p] = think;
            test((p + P − 1) mod P);           // wake-up left neigbor
            test((p + 1) mod P);               // wake-up right neighbor
            V(mutex);
        }
    }
}
```