Lesson: An introduction to Fork

Programming model
Hello World
Shared and private variables
Expression-level synchronous execution
Multiprefix operators
Synchronicity declaration
Synchronous regions: Group concept
Asynchronous regions: Critical sections and locks
Sequential vs. synchronous parallel critical sections
`join` statement
Software packages for Fork

The PRAM programming language Fork

language design: [Hagerup/Seidl/Schmitt'89] [K./Seidl'95,'97] [Keller,K.,Träff'

extension of C

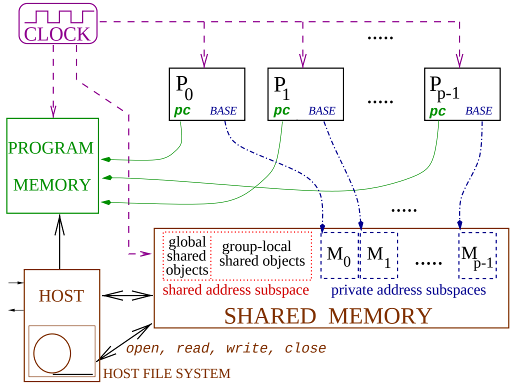
Arbitrary CRCW PRAM with atomic multiprefix operators

synchronicity of the PRAM transparent at expression level

variables to be declared either private or shared

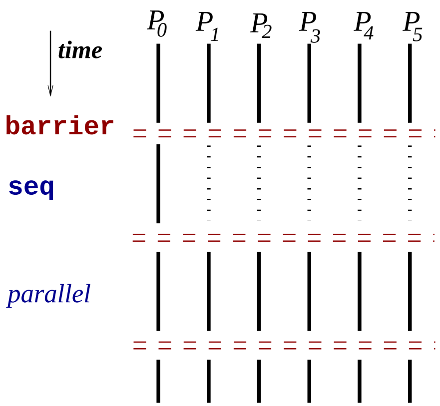
private address subspaces embedded in shared memory

implementation for SB-PRAM



SPMD style of parallel program execution

- fixed set of processors
- no spawn() command
- main() executed by all started processors as one group



Hello World

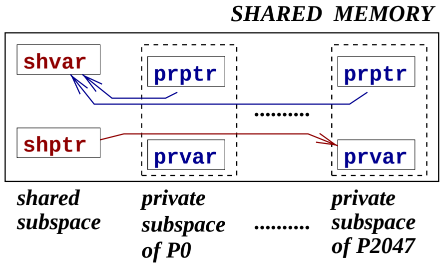
```
#include <fork.h>
#include <io.h>

void main( void )
{
 if (__PROC_NR__ == 0)
   printf("Program executed by\
          %d processors\n",
           __STARTED_PROCS__ );
 barrier;
 pprintf("Hello world from P%d\n",
           __PROC_NR__ );
}
```

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

#0000# Hello world from P0
#0001# Hello world from P1
#0002# Hello world from P2
#0003# Hello world from P3
EXIT: vp=#0, pc=$000001fc
EXIT: vp=#1, pc=$000001fc
EXIT: vp=#2, pc=$000001fc
EXIT: vp=#3, pc=$000001fc
Stop nach 11242 Runden, 642.400 kIps
01fc 18137FFF   POPNG   R6, ffffffff, R

PRAM P0 = (p0, v0)>
```

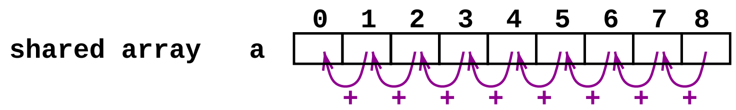
Shared and private variables

- each variable is classified as either shared or private "sharity"
- sh relates to defining group of processors
- pointers: no specification of pointee's sharity required

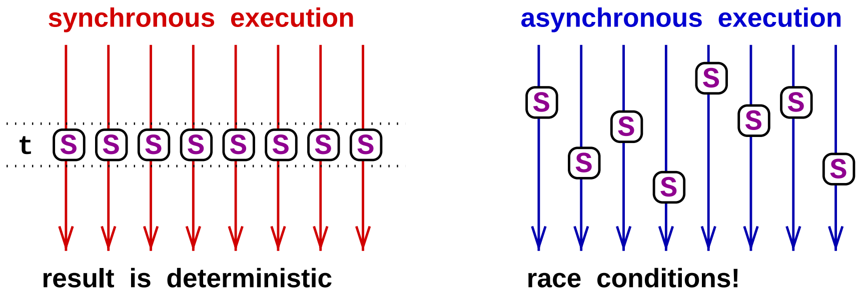
```
pr int prvar, *prptr;
sh int shvar, *shptr;

prptr = &shvar;

shptr = &prvar; // concurrent write!
```



Synchronous execution at the expression level




```
S:   a[$$] = a[$$] + a[$$+1];

             //  $$ in {0..p-1} is processor rank
```



Expressions: Atomic Multiprefix Operators (for integers only)

Set P of processors executes simultaneously

k = `mpadd(` ps, $expression$ `);`

Let ps_i be the location pointed to by the ps expression of processor $i \in P$.
Let s_i be the old contents of ps_i.
Let $Q_{ps} \subseteq P$ denote the set of processors i with $ps_i = ps$.
Each processor $i \in P$ evaluates $expression$ to a value e_i.

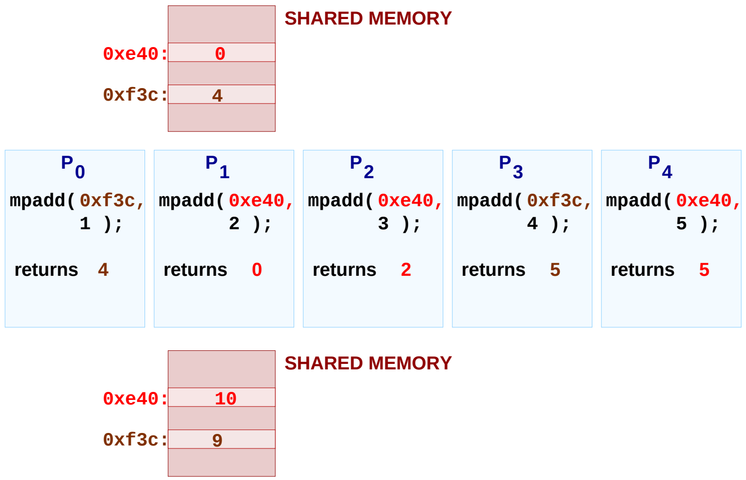
Then the result returned by `mpadd` to processor $i \in P$ is the prefix sum

$$k \leftarrow s_i + \sum_{j \in Q_{s_i},\ j<i} e_j$$

and memory location ps_i is assigned the sum

$$*ps_i \leftarrow s_i + \sum_{j \in Q_{s_i}} e_j$$

Example: Multiprefix addition



`mpadd` may be used as atomic *fetch&add* operator.

Expressions: Atomic Multiprefix Operators (cont.)

Example: User-defined consecutive numbering of processors

```
sh int counter = 0;
pr int me = mpadd( &counter, 1 );
```

Similarly:

`mpmax` (multiprefix maximum)

`mpand` (multiprefix bitwise and)

`mpand` (multiprefix bitwise or)

`mpmax` may be used as atomic *test&set* operator.

Example:

```
pr int oldval = mpmax( &shmloc, 1 );
```


Atomic Update Operators / ilog2

`syncadd(`ps`,` e`)` atomically add value e to contents of location ps

`syncmax` atomically update with maximum

`syncand` atomically update with bitwise and

`syncor` atomically update with bitwise or

`ilog2(`k`)` returns $\lfloor \log_2 k \rfloor$ for integer k

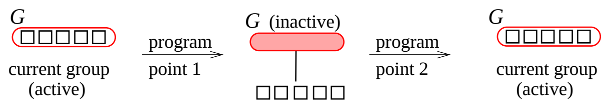
Synchronous and asynchronous program regions

```
sync int *sort( sh int *a, sh int n )
{ extern straight int compute_rank( int *, int);
  if ( n>0 ) {
     pr int myrank = compute_rank( a, n );
     a[myrank] = a[__PROC_NR__];
     return a;
  }
  else
     farm  {
        printf("Error: n=%d\n", n);
        return NULL;
     }
}
extern async int *read_array( int * );
extern async int *print_array( int *, int );
sh int *A, n;
async void main( void )
{
  A = read_array( &n );
  start  {
     A = sort( A, n );
     seq  if (n<100) print_array( A, n );
  }
}
```

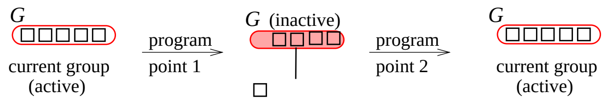
Fork program code regions statically classified as either synchronous, straight, or asynchronous.

Switching from synchronous to asynchronous mode and vice versa

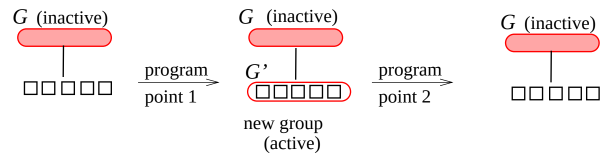
farm
program point 1
statement;
program point 2



seq
program point 1
statement;
program point 2



start
program point 1
statement;
program point 2

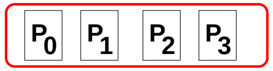


join (...)
statement;

(see later)

Group concept

Groups of processors are explicit:



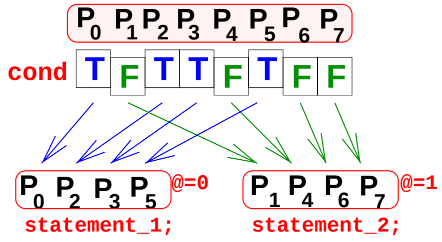
Group ID: `@`
Group size: `#` or `groupsize()`
Group rank: `$$` (automatically ranked from 0 to #-1)

+ Scope of sharing for function-local variables and formal parameters
+ Scope of barrier-synchronization
+ Scope of synchronous execution

Synchronicity invariant: (in synchronous regions):
All processors in the same active group operate synchronously.

Implicit group splitting: IF statement with private condition

```
if (cond)
        statement_1;
else
        statement_2;
```

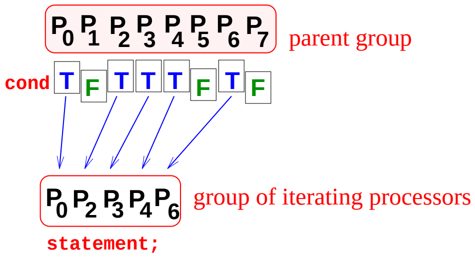


private condition expression
→ current group G of processors must be split into 2 subgroups
to maintain synchronicity invariant.

(parent) group G is reactivated after subgroups have terminated
→ G-wide barrier synchronization

Implicit subgroup creation: Loop with private condition

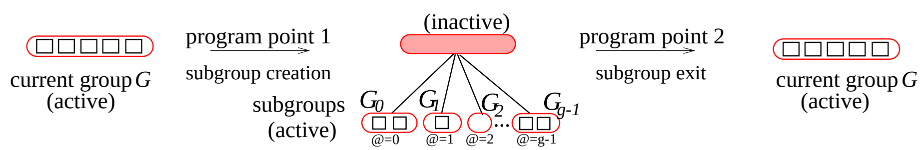
```
while ( cond ) do


        statement;
```



Explicit group splitting: The `fork` statement

```
fork ( g; @ = fn($$); $=$$)
      statement;
```

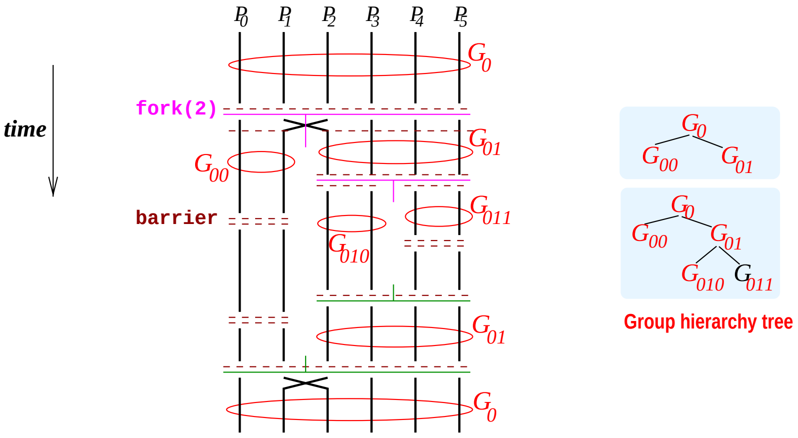
← program point 1
← program point 2



body statement is executed in parallel by all subgroups in parallel

(parent) group G is reactivated when all subgroups have terminated
and resumes after G-wide barrier synchronization at program point 2

The group hierarchy tree



Dynamic / recursive splitting of groups into disjoint subgroups

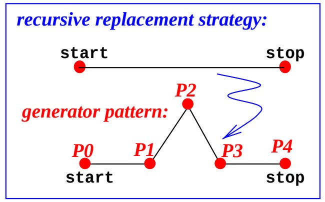
→ at any time the group hierarchy is a logical tree.

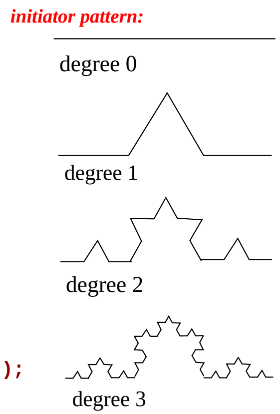
supports nested (multi-level) parallelism

Example: Drawing Koch curves in parallel (1)

Sequential algorithm:

```
void seq_Koch ( int startx, int starty,
                int stopx, int stopy, int level )
{
 int x[5], y[5], dx, dy;
 int i;
 if (level >= DEGREE) {   // reach limit of recursion:
    seq_line( startx, starty,
              stopx, stopy, color, width );
    return;
 }
 // compute x and y coordinates of interpolation points  P0, P1, P2, P3, P4:
 dx = stopx - startx;        dy = stopy - starty;
 x[0] = startx;              y[0] = starty;
 x[1] = startx + (dx/3);     y[1] = starty + (dy/3);
 x[2] = startx + dx/2 - (int)(factor * (float)dy);
 y[2] = starty + dy/2 + (int)(factor * (float)dx);
 x[3] = startx + (2*dx/3);   y[3] = starty + (2*dy/3);
 x[4] = stopx;               y[4] = stopy;

 for ( i=0; i<4; i++ )   //  4  recursive calls
    seq_Koch( x[i], y[i], x[i+1], y[i+1], level + 1 );
}
```



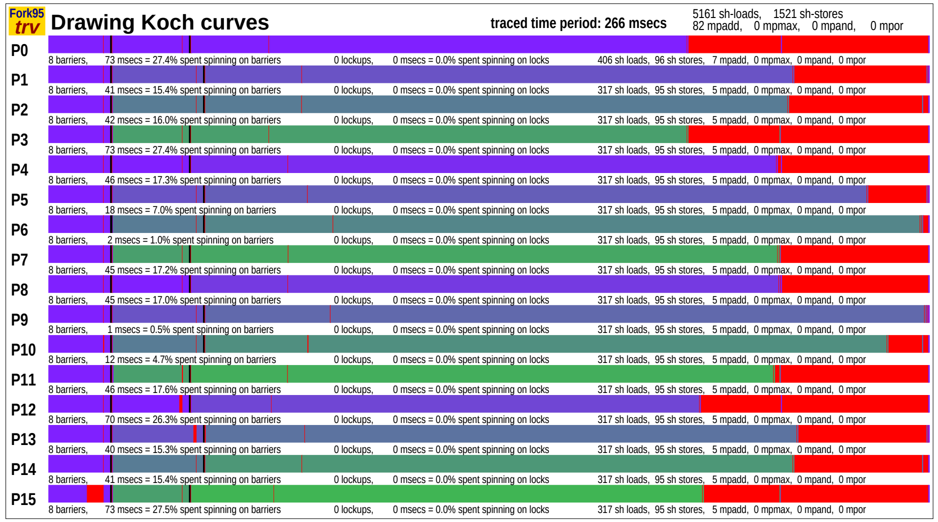


Example: Drawing Koch curves in parallel (2)

```
sync void Koch ( sh int startx, sh int starty,
                 sh int stopx,  sh int stopy,  sh int level )
{
 sh int x[5], y[5], dx, dy;
 pr int i;
 if (level >= DEGREE) {   // terminate recursion:
    line( startx, starty, stopx, stopy, color, width );
    return;
 }
 seq {    // linear interpolation:
    dx = stopx - startx;       dy = stopy - starty;
    x[0] = startx;             y[0] = starty;
    x[1] = startx + (dx/3);    y[1] = starty + (dy/3);
    x[2] = startx + dx/2 - (int)(factor * (float)dy);
    y[2] = starty + dy/2 + (int)(factor * (float)dx);
    x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);
    x[4] = stopx;              y[4] = stopy;
 }
 if (# < 4)   // not enough processors in the group?
    for ( i=$$; i<4; i+=# )    // partially parallel divide-and-conquer step
       farm seq_Koch( x[i], y[i], x[i+1], y[i+1], level + 1 );
 else
    fork ( 4; @ = $$ % 4; )    // parallel divide-and-conquer step
       Koch( x[@], y[@], x[@+1], y[@+1], level + 1 );
}
```


Program trace visualization with the trv tool

-T: instrument the target code to write events to a trace file. Can be processed with trv to FIG image



Asynchronous regions: Critical sections and locks (1)

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;

....
farm {
   ....

   var = var + 1.0;

   ....
}
```

time ↓	P0	P1
	....	....
	....	....
		read 0.0 add 1.0 store 1.0
	read 0.0 add 1.0 store 1.0	....
	....	

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Asynchronous regions: Critical sections and locks (2)

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh int lock = 0; /* mutex var. */
....
farm {
   ....
   while (lock > 0) ; /* wait */
   lock = 1;
   var = var + 1.0;
   lock = 0;
   ....
}
```

!!NOT ATOMIC!!

time ↓	P0	P1
	var = 0.0 lock = 0	var = 0.0 lock = 0
	....	....
	lock == 0 lock = 1	lock == 0 lock = 1
	read 0.0 add 1.0 store 1.0	read 0.0 add 1.0 store 1.0
	....	....

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

Asynchronous regions: Critical sections and locks (3)

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh int lock = 0;
....
farm {
   ....
   while (mpmax(&lock, 1)) ;
                       /* wait */
   var = var + 1.0;
   lock = 0;
   ....
}
```

time ↓	P0	P1
	var = 0.0 lock = 0	var = 0.0 lock = 0 ....
	mpmax: 1	mpmax: 0
	/* wait */	read 0.0 add 1.0 store 1.0 lock = 0
	mpmax: 0	....
	read 1.0 add 1.0 store 2.0	
	....	

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

in Fork: use the mpadd / mpmax / mpand / mpor operators

Asynchronous regions: Critical sections and locks (4)

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh SimpleLock sl;
seq sl = new_SimpleLock();
....
farm {
   ....
   simple_lockup( sl );
                       /* wait */
   var = var + 1.0;
   simple_unlock( sl );
   ....
}
```

time ↓	P0	P1
	var = 0.0 lock = 0	var = 0.0 lock = 0 ....
	mpmax: 1	mpmax: 0
	/* wait */	read 0.0 add 1.0 store 1.0 lock = 0
	mpmax: 0	....
	read 1.0 add 1.0 store 2.0	
	....	

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

in Fork: alternatively: use predefined lock data types and routines

Asynchronous regions: Predefined lock data types and routines

(a) Simple lock

```
SimpleLock new_SimpleLock ( void );
void simple_lock_init ( SimpleLock s );
void simple_lockup ( SimpleLock s );
void simple_unlock ( SimpleLock s );
```

(b) Fair lock (FIFO order of access guaranteed)

```
FairLock new_FairLock ( void );
void fair_lock_init ( FairLock f );
void fair_lockup ( FairLock f );
void fair_unlock ( FairLock f );
```

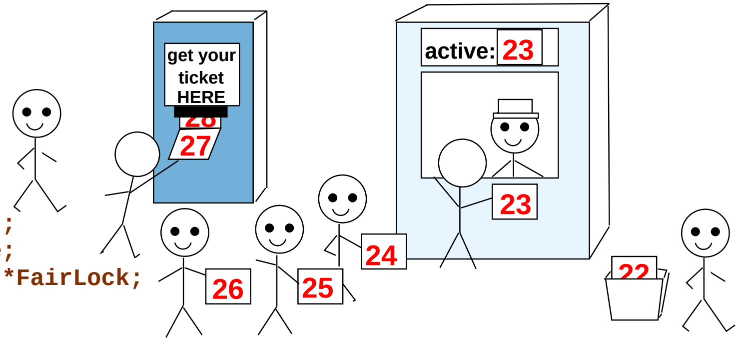
(c) Readers/Writers lock (multiple readers OR single writer)

```
RWLock new_RWLock ( void );
void rw_lock_init ( RWLock r );
void rw_lockup ( RWLock r, int mode );
void rw_unlock ( RWLock r, int mode, int wait );
                        mode in { RW_READ, RW_WRITE }
```

(d) Readers/Writers/Deletors lock (lockup fails if lock is being deleted)

```
RWDLock new_RWDLock ( void );
void rwd_lock_init ( RWDLock d );
int  rwd_lockup ( RWDLock d, int mode );
void rwd_unlock ( RWDLock d, int mode, int wait );
                        mode in { RW_READ, RW_WRITE, RW_DELETE }
```


Asynchronous regions: Implementation of the fair lock



2 counters:

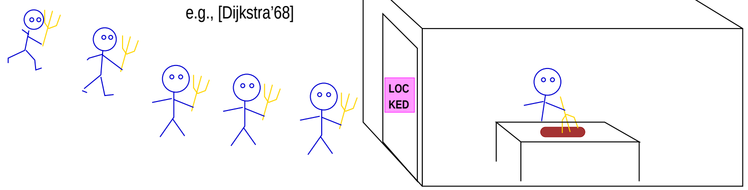
```
struct {
   int ticket;
   int active;
} fair_lock, *FairLock;
```

```
void fair_lockup ( FairLock fl )
{
  int myticket = mpadd( &(fl->ticket), 1);/*atomic fetch&add*/
  while (myticket > fl->active) ;    /*wait*/
}

void fair_unlock ( FairLock fl )
{
  syncadd( &(fl->active), 1 ); /*atomic increment*/
}
```


Sequential vs. synchronous parallel critical sections (1)

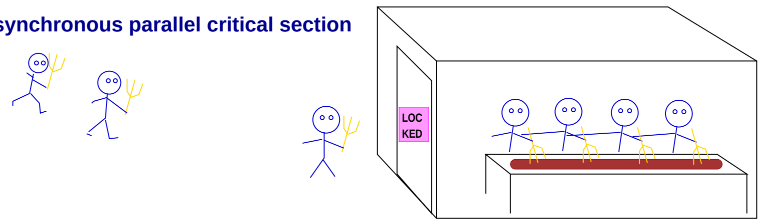
sequential critical section

e.g., [Dijkstra'68]



-> sequentialization of concurrent accesses to a shared object / resource

synchronous parallel critical section



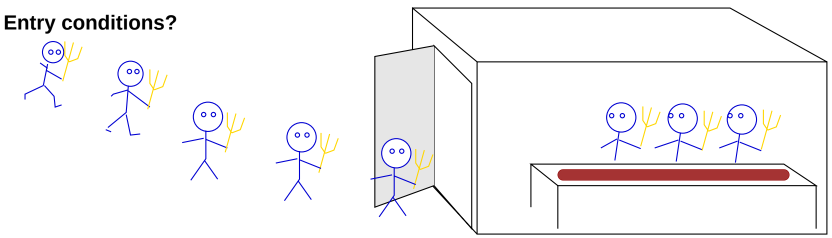
-> allow simultaneous entry of more than one processor

-> deterministic parallel access by executing a synchronous parallel algorithm

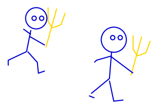
-> at most one group of processors inside at any point of time

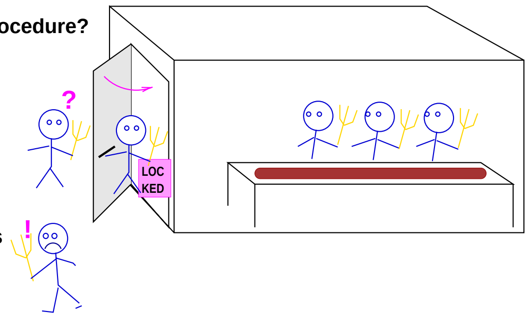
Sequential vs. synchronous parallel critical sections (2)

Entry conditions?



When to terminate the entry procedure?

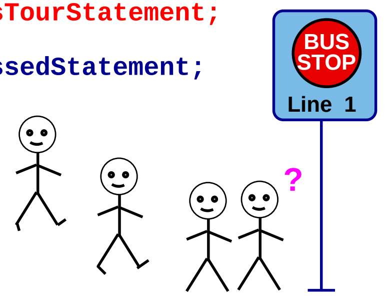




What happens with processors not allowed to enter?

The join statement: excursion bus analogy (1)

```
join ( SMsize; delayCond; stayInsideCond )
     busTourStatement;
else
     missedStatement;
```

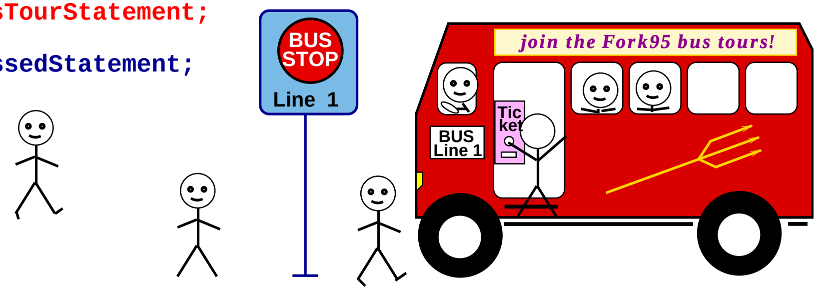


Bus gone? - execute else part: `missedStatement;`

- continue in else part: jump back to bus stop (join entry point)
- break in else part: continue with next activity (join exit point)

The join statement: excursion bus analogy (2)

```
join ( SMsize; delayCond; stayInsideCond )
     busTourStatement;
else
     missedStatement;
```

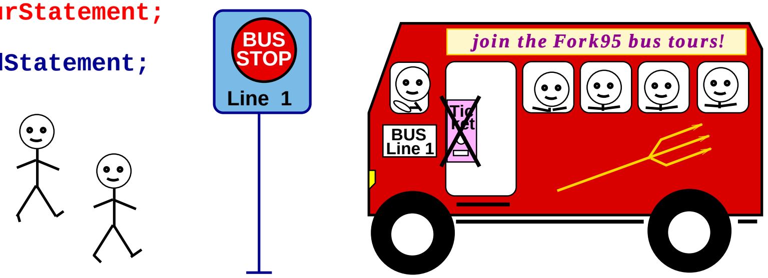


Bus waiting: - get a ticket and enter

- ticket number is 0? -> driver!
  driver initializes shared memory (SMsize) for the bus group
  driver then waits for some event: `delayCond`
  driver then switches off the ticket automaton

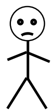
The join statement: excursion bus analogy (3)

```
join ( SMsize; delayCond; stayInsideCond )
     busTourStatement;
else
     missedStatement;
```



Bus waiting: - get a ticket and enter

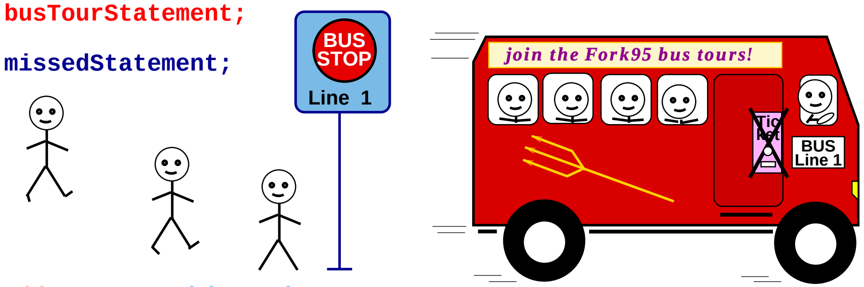
- ticket number is 0? -> driver!



- if not `stayInsideCond` spring off and continue with else part

The join statement: excursion bus analogy (4)

```
join ( SMsize; delayCond; stayInsideCond )
     busTourStatement;
else
     missedStatement;
```

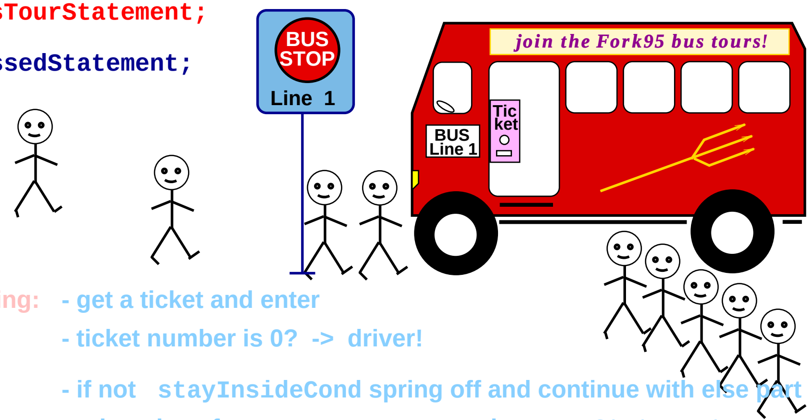


Bus waiting: - get a ticket and enter

- ticket number is 0? -> driver!
- if not `stayInsideCond` spring off and continue with else part
- otherwise: form a group, execute `busTourStatement` synchronously

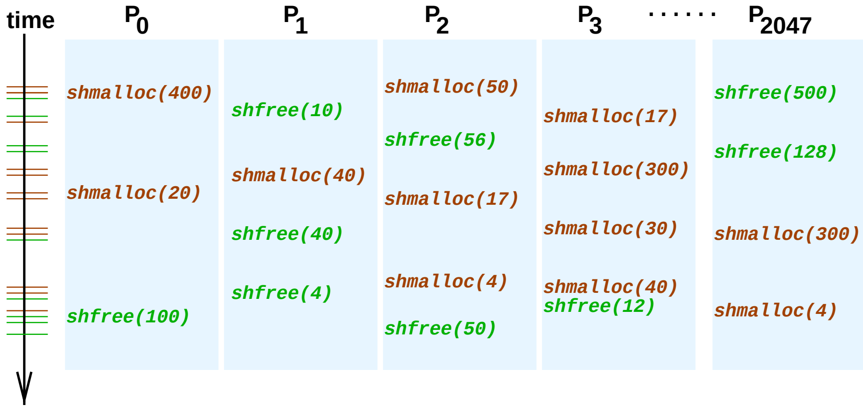
The join statement: excursion bus analogy (5)

```
join ( SMsize; delayCond; stayInsideCond )
     busTourStatement;
else
     missedStatement;
```



Bus waiting:
- get a ticket and enter
- ticket number is 0? -> driver!
- if not `stayInsideCond` spring off and continue with else part
- otherwise: form a group, execute `busTourStatement`
- at return: leave the bus, re-open ticket automaton and continue with next activity

The join statement, example (1): parallel shared heap memory allocation



Idea:
- use a synchronous parallel algorithm for shared heap administration
- collect multiple queries to shmalloc() / shfree() with join() and process them as a whole in parallel!

Question: Does this really pay off in practice?

The join statement, example (2)

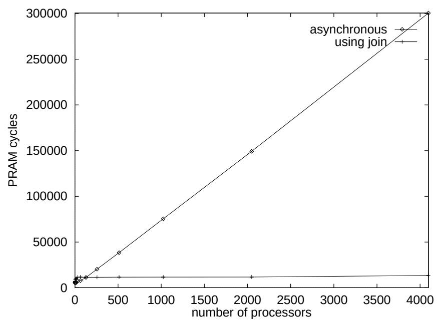
Experiment:

Simple block–oriented parallel shared heap memory allocator

First variant: sequential critical section, using a simple lock

Second variant: parallel critical section, using `join`

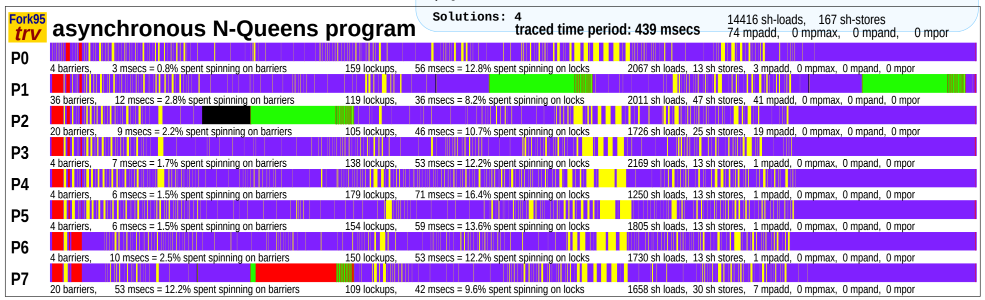
p	asynchronous		using `join`	
1	5390 cc	(21 ms)	6608 cc	(25 ms)
2	5390 cc	(21 ms)	7076 cc	(27 ms)
4	5420 cc	(21 ms)	8764 cc	(34 ms)
8	5666 cc	(22 ms)	9522 cc	(37 ms)
16	5698 cc	(22 ms)	10034 cc	(39 ms)
32	7368 cc	(28 ms)	11538 cc	(45 ms)
64	7712 cc	(30 ms)	11678 cc	(45 ms)
128	11216 cc	(43 ms)	11462 cc	(44 ms)
256	20332 cc	(79 ms)	11432 cc	(44 ms)
512	38406 cc	(150 ms)	11556 cc	(45 ms)
1024	75410 cc	(294 ms)	11636 cc	(45 ms)
2048	149300 cc	(583 ms)	11736 cc	(45 ms)
4096	300500 cc	(1173 ms)	13380 cc	(52 ms)



The join statement, example (3)

asynchronous parallel N-queens program uses join for parallel output of solutions

```
PRAM P0 = (p0, v0)> g
Enter N = 6
Computing solutions to the 6-Queens problem...
------------------- Next 2 solutions (1..2): -------------------
|..Q...|...Q..
|.....Q|Q.....
|.Q....|....Q.
|....Q.|.Q....
|Q.....|.....Q
|...Q..|..Q...
------------------- Next 1 solutions (3..3): -------------------
|.Q....
|...Q..
|.....Q
|Q.....
|..Q...
|....Q.
------------------- Next 1 solutions (4..4): -------------------
|....Q.
|..Q...
|Q.....
|.....Q
|...Q..
|.Q....
Solutions: 4
```



Available software packages

PAD library [Träff'95–98], [PPP 8]

PRAM algorithms and data structures

APPEND library [PPP 7.4]

asynchronous parallel data structures

MPI core implementation in Fork [PPP 7.6]

Skeleton functions [PPP 7]

generic map, reduce, prefix, divide-and-conquer, pipe, ...

FView fish-eye viewer for layouted graphs [PPP 9]

N-body simulation [PPP 7.8]