CONCURRENT AND PARALLEL PROGRAMMING

Wojciech Mikanik, PhD
wojciech.mikanik@polsl.pl
room 503

February 2009

Part I

Introduction

Outline

1. Outline of the course
2. Bibliography
3. Sequential and parallel processes
4. Paradigms of concurrent programming

Context

Prevalence of concurrency and parallelism in information technology

- Sending a text message from my mobile phone
- Surfing the web in multi tab browser
- Playing a computer game
  - GPU processing: very regular
  - Game objects: irregular
- A MMOG server
- Company/department level data mining of unstructured data
- Google data center
- Large scale scientific and engineering simulations
  - Airplane, car etc design (virtual crash test)
  - Drug design
  - Nuclear stockpile maintenance
  - Weather simulations
  - Prediction of the development of a fire in the (super)real-time

Context
Diversity of computing architectures

- Single core
- Multicores
  - "Fat" multicore (Intel, AMD, ...)
  - Cell Broadband Engine (Sony Playstation 3, IBM Cell Blade)
- Multiprocessors
  - SMPs (Sun Fire 15K, ...)
  - ccNUMAs (SGI Altix, ...)
- MPP (IBM BlueGene, ...)
- Less conventional hardware
  - GP GPU (Nvidia CUDA, ATI Brook)
  - FPGA
- Vector computers
- Mulithreaded architectures
- Clusters
  A cluster of SMPs with multicore CPUs and GP-GPU added

Topics

Lecture

- Fundamentals
- Expressing concurrency
- Concurrent programming with shared memory
- Concurrent programming with distributed memory
- Performance measures of parallel algorithms
- Parallel programming with shared memory
- Parallel programming with distributed memory
- Partitioned Global Address Space languages (option)

Laboratory

- Posix threads
- OpenMP
- MPI
- Chapel (option)

Grading

Listen carefully ...

Outline

1. Outline of the course
2. **Bibliography**
3. Sequential and parallel processes
4. Paradigms of concurrent programming

Bibliography

1. M. Ben-Ari: *Podstawy programowania współbieżnego i rozproszonego*, Warszawa, WNT 1996.
2. Z. Weiss, T. Gruźlewski: *Programowanie współbieżne i rozproszone*, Warszawa, WNT 1993.
3. B. Chapman, G. Jost, R. van der Pas, *Using OpenMP*, MIT Press, 2008.
4. P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufman Publishers, Inc., San Francisco, 1996.

Lecture notes:
`http://sun.aei.polsl.pl/pub/wmikanik/concprg/index.html`

Lecture notation

- You have seen it during the course on algorithms
- New elements introduced gradually during the course
  - declarations
  - data types
  - statements
- Notation, not a programming language
- A mean to an end

Outline

1. Outline of the course
2. Bibliography
3. **Sequential and parallel processes**
4. Paradigms of concurrent programming

Sequential process

P — process

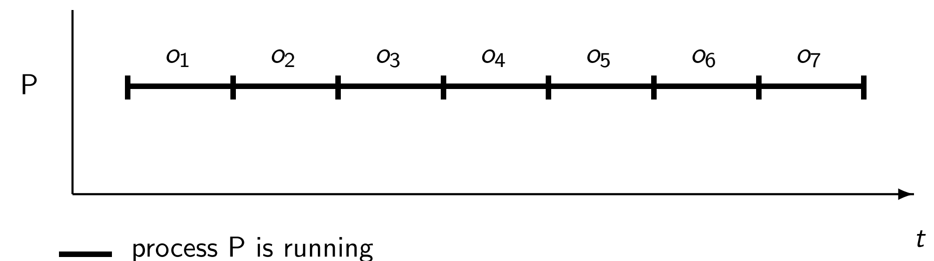
o_i — i-th operation

$t(\underline{o_i})$ — time when the operation o_i started

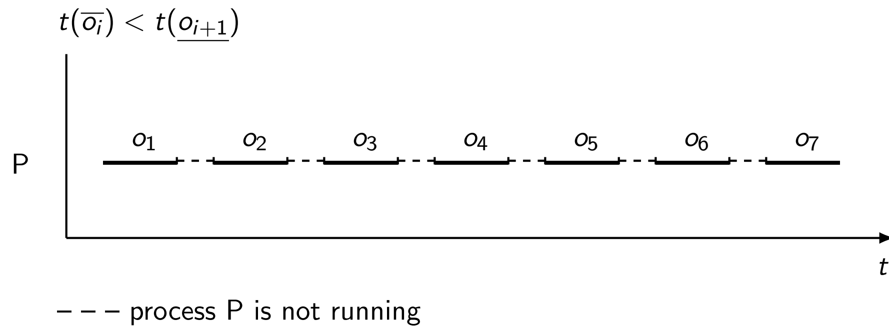
$t(\overline{o_i})$ — time when the operation o_i finished

In case of sequential process $t(\overline{o_i}) \leq t(\underline{o_{i+1}})$

$t(\overline{o_i}) = t(\underline{o_{i+1}})$



Sequential process



Concurrent processes

P_k — k-th process

o_i^k — i-th operation of k-th process

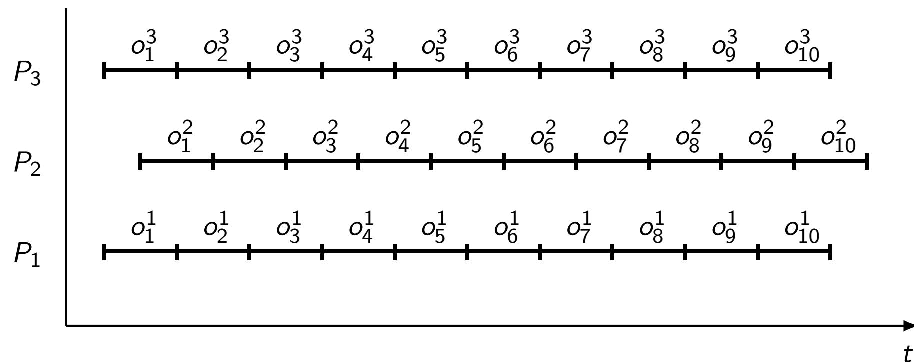
Sequence of operations of process P_1: $o_1^1, o_2^1, o_3^1, o_4^1, o_5^1, o_6^1$
Sequence of operations of process P_2: $o_1^2, o_2^2, o_3^2, o_4^2, o_5^2, o_6^2$
Possible sequences of operations in case of concurrent execution of P_1 and P_2:

- $o_1^1, o_1^2, o_2^1, o_2^2, o_3^1, o_3^2, o_4^1, o_4^2, o_5^1, o_5^2, o_6^1, o_6^2$
- $o_1^1, o_2^1, o_3^1, o_4^1, o_5^1, o_6^1, o_1^2, o_2^2, o_3^2, o_4^2, o_5^2, o_6^2$
- $o_1^1, o_2^1, o_3^1, o_4^1, o_1^2, o_2^2, o_5^1, o_6^1, o_3^2, o_4^2, o_5^2, o_6^2$
- ...

NOTHING IS KNOWN ABOUT THE SPEED OF EXECUTION OF CONCURRENT PROCESSES

Concurrent processes

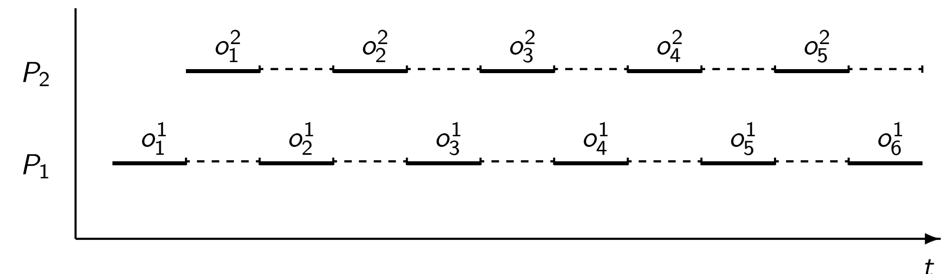
Parallel vs concurrent execution



$$t(\underline{o_1^1}) < t(\underline{o_1^2}) < t(\overline{o_1^1})$$

Concurrent processes

Time sharing



$$t(\overline{o_1^1}) = t(\underline{o_1^2})$$

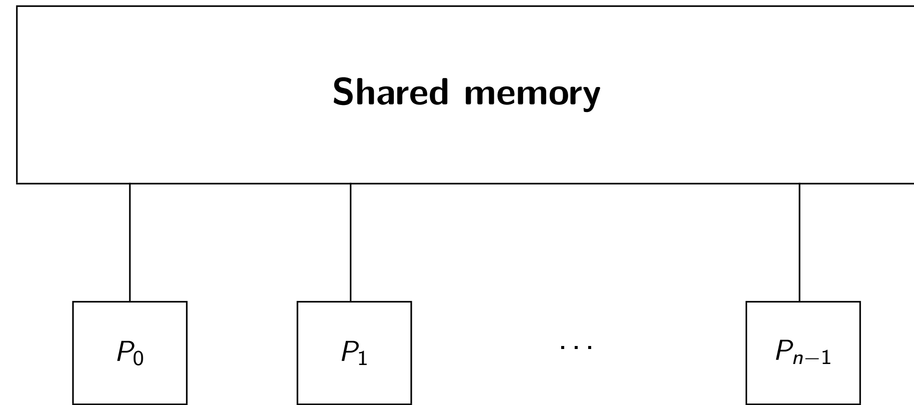
NOTHING IS KNOWN ABOUT THE SPEED OF EXECUTION OF CONCURRENT PROCESSES

Outline

1. Outline of the course
2. Bibliography
3. Sequential and parallel processes
4. **Paradigms of concurrent programming**

Parallel programming models

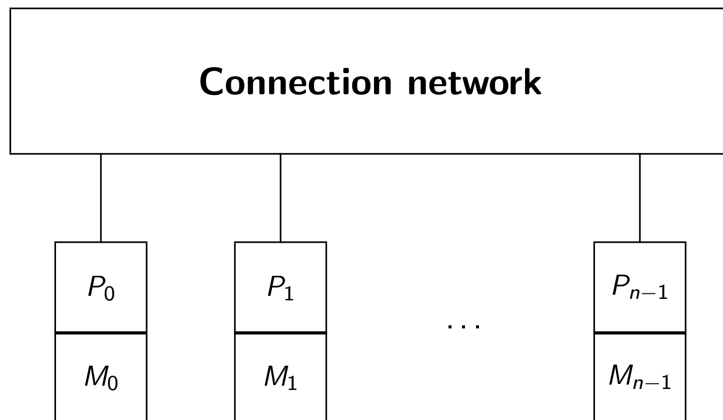
Shared memory (single address space)



Random access to any memory cell

Parallel programming models

Distributed memory (many address spaces)



Part II

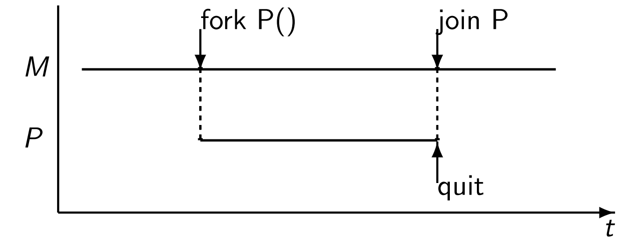
Expressing concurrency

Outline

1. Statements: `fork`, `join`, `quit`
2. `cobegin` ... `coend` block
3. `parfor` statement

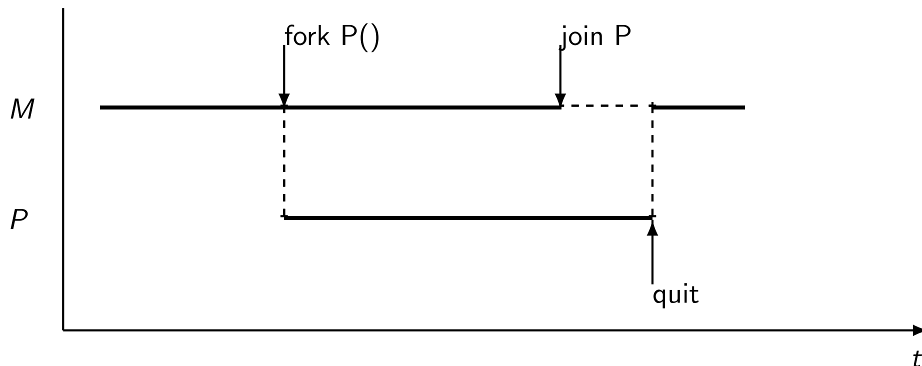
Statements: `fork`, `join`, `quit`

```
procedure P;
begin
  ... //statements
  ...
  quit;
end; //P
procedure M;
begin
  ...
  fork P();
  ...
  join P;
  ...
end; //M
begin //main block
  M();
end;
```



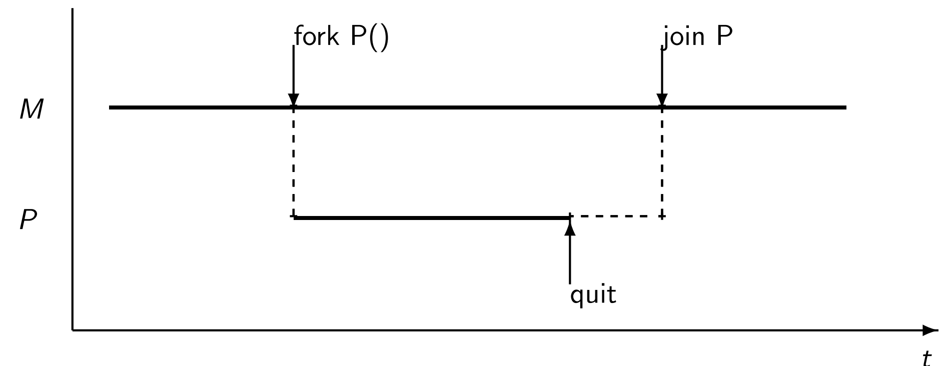
Statements: `fork`, `join`, `quit`

M waits for *P*



Statements: `fork`, `join`, `quit`

P waits for *M*



Names of processes

```
procedure P;
begin
  ... //statements
  quit;
end;//P
procedure M;
begin
  ...
  fork P();
  fork P();
  ...
  join P;  //which process P ?
  ...
end; //M
begin  // main block
  M();
end;
```

Names of processes

```
procedure P;
begin
  ... //statements
  quit;
end;//P
procedure M;
begin
  ...
  fork P() alias a;
  fork P() alias b;
  ...
  join a;  //awaiting termination of process a
  ...
  join b;  //awaiting termination of process b
  ...
end; //M
begin  // main block
  M();
end;
```

fork, join, quit: an example

```
var
  int a[1 .. 2 * N];
  int sP;

procedure sum(int f, int l);
  var
    int i;
    int s := 0;
begin
  for i := f to l do
    s += a[i];
  end for;
  sP := s;
  quit;
end; //sum
```

```
procedure M;
  var
    int i, s := 0;
begin
  read_data (a, 2 * N);
  fork sum (1, N) alias P;
  for i := N + 1 to 2 * N do
    s += a[i];
  end for;
  join P;
  printf ("Sum = %d\n", s + sP);
  return;
end; //M
begin  // main block
  M();
end;
```

fork, join, quit: an example

Modified procedure sum

```
procedure M;
  var
    int s;
begin
  read_data (a, 2 * N);
  fork sum (...) alias P;
  sum (...);
  join P;
  ...
  printf ("Sum = %d\n", s);
  return;
end; //M
```

fork, join, quit: an example

Modified procedure sum

```
int a[1 .. 2 * N];

procedure sum (int f,
               int l,
               int * s);
  var
    int i;
begin
  *s = 0;
  for i := f to l do
    *s += a[i];
  return;
end; //sum
```

```
procedure M;
  var
    int s, sP;
begin
  read_data (a, 2 * N);
  fork sum (1, N, &sP) alias P;
  sum (N + 1, 2 * N, &s);
  join P;
  s += sP;
  printf ("Sum = %d\n", s);
  return;
end; //M
begin //main block
  M();
end;
```


Block cobegin . . . coend

cobegin
  S_1;
  S_2;
  . . .
  S_n;
coend;

cobegin
  $a := 2$;
  $b := 4 * sin(12)$;
  . . .
  $z := F(102)$;
coend;

Block cobegin . . . coend

Computing sum of elements of an array

```
var
  int s1 := 0, s2 := 0, a[1 .. 2 * N];
begin
  read_data (a, 2 * N);
  cobegin

    var int i;
    begin
      for i := 1 to N do
        s1 += a[i];
      end for;
    end;

    var int i;
    begin
      for i := N + 1 to 2 * N do
        s2 += a[i];
      end for;
    end;

  coend;
  printf ("Sum = %d\n", s1 + s2);
end;
```


Block cobegin . . . coend

Names of processes

cobegin
  as *processName1 statement 1*;
  as *processName2 statement 2*;
  ⋮
coend

```
cobegin
  as printer print (a);
  as reader read(b);
  as computer
    begin
      compute(c);
      compute(d);
    end;
coend
```

cobegin coend versus fork, join, quit

```
cobegin
  S1;
  S2;
  ...
  Sn;
coend;
```

```
...
fork P2();
fork P3();
...
...
...
...
fork Pn();
S1;
join P2;
join P3;
...
...
join Pn;
...
```

```
procedure P2;
begin
  S2;
  quit;
end;
procedure P3;
begin
    S3;
    quit;
end;
...
procedure Pn;
begin
    Sn;
    quit;
end;
```


parfor statement

```
cobegin
  S_1;
  S_2;
  ...
  S_n;
coend
```

```
parfor i := 1 to n do
  S_i;
end parfor;
```

```
parfor i := low_value to high_value do
  S_i;
end parfor;
```

```
int a[1 .. n], i;
parfor i := 1 to n do
  a[i] := i;
end parfor;
```


Names of processes

```
parfor i := low_value to high_value as name<i> do
  S_i;
end parfor;
```

```
int a[1 .. n], i;
parfor i := 1 to n as proc<i> do
  a[i] := i;
end parfor;
```

```
proc<1>, proc<2>, proc<3>, ..., proc<n>
```


Part III

Shared memory

Outline

1. Mutual exclusion
2. Mutexes
3. Semaphores
4. Monitors
5. Condition variables versus semaphores

Mutual exclusion

```
begin
  int x;

  cobegin
    begin   //process P1
      ...   //statements
      x := x + 1;
      ...   //statements
    end;
    begin   //process P2
      ...   //statements
      x := x + 2;
      ...   //statements
    end;
  coend;
end.
```


Mutual exclusion

```
  int x;
  cobegin
    begin  //process P1
      ...  //statements
(1.1) reg1 := x;
(1.2) reg1 := reg1 + 1;
(1.3) x := reg1;
      ...  //statements
    end;
    begin  //process P2
      ...  //statements
(2.1) reg2 := x;
(2.2) reg2 := reg2 + 2;
(2.3) x := reg2;
      ...  //statements
    end;
  coend.
```


Mutual exclusion

- an increment of 3

(1.1) reg1 := x;
(1.2) reg1 := reg1 + 1;
(1.3) x := reg1;
(2.1) reg2 := x;
(2.2) reg2 := reg2 + 2;
(2.3) x := reg2;

- an increment of 2

(1.1) reg1 := x;
(1.2) reg1 := reg1 + 1;
(2.1) reg2 := x;
(1.3) x := reg1;
(2.2) reg2 := reg2 + 2;
(2.3) x := reg2;

- an increment of 1

(2.1) reg2 := x;
(1.1) reg1 := x;
(1.2) reg1 := reg1 + 1;
(2.2) reg2 := reg2 + 2;
(2.3) x := reg2;
(1.3) x := reg1;

Mutual exclusion

```
int x;   //shared data

cobegin
  begin  //process P1
    ...  //statements
    x := x + 1;      //critical section
    ...  //statements
  end;
  begin  //process P2
    ...  //statements
    x := x + 2;      //critical section
    ...  //statements
  end;
coend
```

Mutex

- MUTual EXclusion
- type mutex = record

```
type mutex = record
               enum {locked, unlocked} status;
               queue_t q;
             end;
```

- procedure init (mutex &m);

```
procedure init (mutex &m);
begin
  m.status := unlocked;
  m.q := NULL;
end;
```

Mutex

Procedures *lock* and *unlock*

- procedure lock (mutex &m);

```
procedure lock (mutex &m);
begin
  if m.status = locked then
    suspend the current process and put into m.q
  else
    m.status := locked;
  end if;
end;
```

- procedure unlock (mutex &m);

```
procedure unlock (mutex &m);
begin
  if !  empty (m.q) then
    get a process from m.q and resume its execution
  else
    m.status := unlocked;
  end if;
end;
```

Mutex

Mutual exclusion

```
int x;          //shared data
mutex m ;

cobegin
  begin         //process P1
    ...         //statements
    lock (m);
    x := x + 1;//critical section
    unlock (m);
    ...         //statements
  end;
  begin         //process P2
    ...         //statements
    lock (m);
    x := x + 2;//critical section
    unlock (m);
    ...         //statements
  end;
coend;
```

Semaphore

- ```
  type sem = record
               int val;
               queue_t q;
             end;
  ```
- ```
  procedure init (sem &s, int v);
  begin
    s.val := v;
    s.q := NULL;
  end;

  s := {v};
  ```

Semaphore
Procedures *P* and *V*

- ```
  procedure P (sem &s);
  begin
    if s.val <= 0 then
      suspend the current process and put into s.q
    else
      s.val --;
  end;
  ```
- ```
  procedure V (sem &s);
  begin
    if ! empty (s.q) then
      get a process from s.q and resume its execution
    else
      s.val ++;
  end;
  ```

Semaphore
Mutual exclusion

```
int x;          //shared data
sem s := {1};

cobegin
  begin         //process P1
    ...         //statements
    P (s);
    x := x + 1;//critical section
    V (s);
    ...         //statements
  end;
  begin         //process P2
    ...         //statements
    P (s);
    x := x + 2;//critical section
    V (s);
    ...         //statements
  end;
coend;
```

Producers and Consumers

- Problem description
  Two kinds of processes:
  - Producers, by executing local procedure *Create*, generate data items to be passed to Consumers,
  - Consumers having received a data item process it using local procedure *Use*.

  Design passing of the data from Producers to Consumers.
- Assumption: data items are integers.

Producers and Consumers (1 → 1)

```
var
  sem can_read := {0};
  sem can_write := {1};
  int x; //shared data

cobegin
  begin  //Producer
    var int lx;
    while true do
      lx := Create();
      P (can_write);
      x := lx;
      V (can_read);
    end while;
  end; //Producer
```

```
  begin //Consumer
    var int lx;
    while true do
      P (can_read);
      lx := x;
      V (can_write);
      Use (lx);
    end while;
  end; //Consumer
coend;
```


Producers and Consumers ($N \longrightarrow 1$)

```
var
  sem can_read := {0};
  sem can_write := {1};
  int x; //shared data

cobegin
  parfor i := 1 to N do   //Producers
    var int lx;
    while true do
      lx := Create();
      P (can_write);
      x := lx;
      V (can_read);
    end while;
  end parfor; //Producers
```

```
  begin //Consumer
    var int lx;
    while true do
      P (can_read);
      lx := x;
      V (can_write);
      Use (lx);
    end while;
  end; //Consumer
coend;
```


Producers and Consumers ($N \longrightarrow M$)

```
var
  sem can_read := {0};
  sem can_write := {1};
  int x; //shared data

cobegin
  parfor i := 1 to N do   //Producers
    var int lx;
    while true do
      lx := Create();
      P (can_write);
      x := lx;
      V (can_read);
    end while;
  end parfor; //Producers
```

```
  //Consumers
  parfor i := 1 to M do
    var int lx;
    while true do
      P (can_read);
      lx := x;
      V (can_write);
      Use (lx);
    end while;
  end parfor;

coend;
```


Once again: Producers and Consumers (1 → 1)

```
var
  sem can_read := {0};
  sem can_write := {1};
  int x; //shared data

cobegin
  begin  //Producer
    var int lx;
    while true do
      lx := Create();
      P (can_write);
      x := lx;
      V (can_read);
    end while;
  end; //Producer
```

```
  begin //Consumer
    var int lx;
    while true do
      P (can_read);
      lx := x;
      V (can_write);
      Use (lx);
    end while;
  end; //Consumer
coend;
```

Bounded buffer

```
  int b[size], head := 0, tail := 0;
  mutex me;
  sem free := {size};
  sem taken := {0};

  cobegin
      var int lx;
      while true do  //Producer
        lx := Generate();
(1)     P (free);
        lock (me);
        b[head] := lx;
        head := (head + 1) % size;
        unlock (me);
(2)     V (taken);
      end while;  //Producer

      var int lx;
      while true do //Consumer
(3)     P (taken);
        lock (me);
        lx := b[tail];
        tail := (tail + 1) % size;
        unlock (me);
(4)     V (free);
        Use (lx);
      end while; //Consumer
  coend;
```


Readers and Writers

```
var
  sem write := {1};
  mutex me;
  int rr := 0;

cobegin
  //Writers
  parfor i := 1 to nW do
    P (write);
    ...  //change data
    V (write);
  end parfor;

  //Readers
  parfor i := 1 to nR do
    lock (me);
    rr ++;
    if rr = 1 then
      P(write);
    end if;
    unlock (me);
    ... //read data
    lock(me);
    rr --;
    if rr = 0 then
      V (write);
    end if;
    unlock (me);
  end parfor;
coend;
```


Barrier synchronization

```
parfor i := 1 to n do  //Processes P1 ..Pn
  A();
  B();
end parfor;
```

None of the processes $P_1, \ldots, P_n$ is allowed to start executing procedure $B()$, unless all processes $P_1, \ldots, P_n$ have finished the execution of procedure $A()$.

```
parfor i := 1 to n do  //Processes P1 ..Pn
  A();
  barrier();
  B();
end parfor;
```


Barrier synchronization

```
var
  int nbr := 0;
  mutex m_nbr;
  sem s_waiting := {0};

procedure barrier;
begin
  lock(m_nbr);
  nbr ++;
  if nbr = n then
    for i := 1 to n do
      V(s_waiting);
    end for;
  end if;
  unlock(m_nbr);
  P(s_waiting);
end;

parfor i := 1 to n do
  A();
  barrier();
  B();
end parfor;
```

Barrier synchronization

```
var
  sem sb1 := {0};
  sem sb2 := {0};

procedure barrier;
begin
  V (sb1);
  P (sb2);
end;

cobegin
  parfor i := 1 to n do
    A();
    barrier();
    B();
  end parfor;
```

```
  //barrier manager
  var int i;
  while true do
    for i := 1 to n do
      P (sb1);
    end for;
    for i := 1 to n do
      V (sb2);
    end for;
  end while;
coend;
```

Weaknesses of the semaphores

- *P* and *V* mismatch:

  ```
  P (a_semaphore);
  ```
  critical section
  ```
  V (wrong_semaphore);
  ```

- A risk of not protecting a part of critical section
- No syntax bindings between a resource and a semaphore
- One way to solve two quite different problems
  - mutual exclusion
  - synchronization of conditions

Weaknesses of the semaphores

```
  int b[size], h := 0, t := 0;
  sem me := {1};
  sem free := {size};
  sem taken := {0};

  cobegin
      var int lx;
      while true do  //Producer
        lx := Generate();
(1)     P (free);
        P (me);
        b[h] := lx;
        h := (h + 1) % size;
        V (me);
(2)     V (taken);
      end while;  //Producer
```

```
      var int lx;
      while true do //Consumer
(3)     P (taken);
        P (me);
        lx := b[t];
        t := (t + 1) % size;
        V (me);
(4)     V (free);
        Use (lx);
      end while; //Consumer
  coend;
```

Monitors

Members of a monitor

- Private variables
  - shared variables
  - condition variables
- Procedures
- Setup block

Monitor type versus class

Monitor type	Class
shared variables	instance variables
procedures	methods
all procedures „public”	methods can be: public, protected, private
monitor.procedure(p)	object.method(p)
setup block	constructor
	destructor
condition variables	
	inheritance, polimorphism

Declaration of a monitor type

```
id = monitor
    private:
      declaration of private variables
    public:
      procedure  id (arguments); //setup block
      begin
        setting up the state of the monitor
      end;
      procedure  op1 (arguments); //procedure 1
        declaration of local variables of procedure op1;
      begin
        body of procedure op1
      end;
      ⋮
      procedure  opN (arguments); //procedure N
        declaration of local variables of procedure opN;
      begin
        body of procedure opN
      end;
  end;
```


Procedures of condition variables

```
procedure wait;
begin
  put the process into condition variable's queue;
end;


procedure signal;
begin
  if  condition variable's queue is not empty  then
    suspend the current process;
    resume one process from condition variable's queue;
  end if;
end;
```

Calling procedures: *condition_variable.procedure*()

`taken.signal()`

Bounded buffer

```
bbuffer = monitor
    private:
      const int n := ...
      T elem[n];
      int head := 0;
      int tail := 0;
      int taken := 0;
      condition notEmpty, notFull;
    public:
      procedure put (T p)
      begin
        if taken = n then
          notFull.wait();
        end if;
        elem[head] := p;
        taken += 1;
        head := (head + 1) % n;
        notEmpty.signal();
      end;
```

```
  procedure get (T &q)
  begin
    if taken = 0 then
      notEmpty.wait();
    end if;
    q := elem[tail];
    taken -= 1;
    tail := (tail + 1 ) % n;
    notFull.signal();
  end;
end;
```

Producers and Consumers

```
bbuffer b;

cobegin
  //Producers
  parfor i := 1 to N do
    var int lx;
    while true do
      lx := Create();
      b.put(lx);
    end whle;
  end parfor;
```

```
  //Consumers
  parfor i := 1 to M do
    var int lx;
    while true do
      b.get(lx);
      Use(lx);
    end while;
  end parfor;
coend;
```

Readers and Writers

```
const int nR = ...
const int nW = ...
  var
    monitor_cp arbiter;
    typ_data_base data_base;

  cobegin {
    parfor i = 1 to nR do
      ...  //statements
      arbiter.read_start();
      ... //read the data base
      arbiter.read_stop();
      ... //statements
    end parfor;
```

```
    parfor i := 1 to nW do
      ... //statements
      arbiter.write_start();
      ... //write the data base
      arbiter.write_stop();
      ... //statements
    end parfor;
  coend;
```

Readers and Writers

```
monitor_cp = monitor
  private:
    int cc := 0;
    bool write := false;
    condition can_read, can_write;
  public:
    procedure read_start
    begin
      if write || !  empty(can_write) then
        can_read.wait();
      end if;
      cc := cc + 1;
      can_read.signal();
    end;
    procedure read_stop;
    begin
      cc := cc - 1;
      if cc = 0 then
        can_write.signal();
      end if;
    end;
```

Readers and Writers

```
    procedure write_start;
    begin
      if cc > 0 || write then
        can_write.wait();
      end if;
      write := true;
    end;
    procedure write_stop;
    begin
      write := false;
      if !  empty(can_read) then
        can_read.signal();
      else
        can_write.signal();
      end if;
    end;
  end; //monitor_cp
```

Barrier synchronization

Monitor

```
const int n = ...;
monitor_b = monitor
  private:
    int cntWaiting := 0;
    condition barrier;
  public:
    procedure sync;
    begin
      if cntWaiting = n - 1 then
        for i := 1 to n - 1 do
          barrier.signal();
        end for;
        cntWaiting := 0;
      else
        cntWaiting ++;
        barrier.wait();
      end if;
    end;
  end; //monitor_b
```

```
var monitor_b m;
parfor i := 1 to n do
  A();
  m.sync();
  B();
end parfor;
```


Condition variables versus semaphores

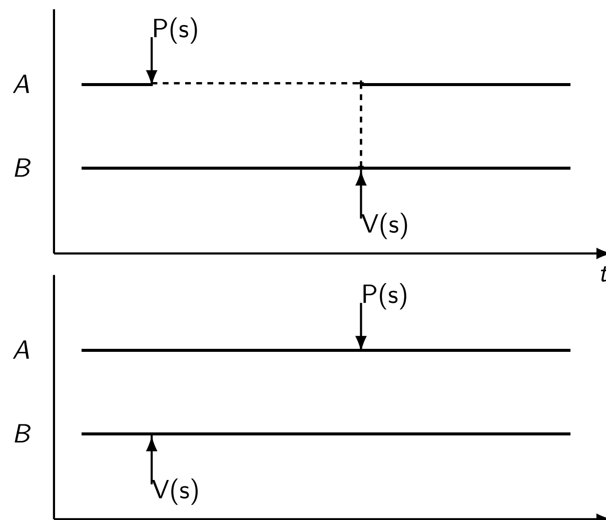
Comparison of procedures

Condition variable	**Semaphore**
procedure *wait()*	procedure *P()*
suspends a process (always)	▶ decrements the semaphore's counter ▶ suspends a process if counter negative
procedure *signal()*	procedure *V()*
no effect if the queue is empty	if the queue is empty the counter is incremented

Condition variables versus semaphores

Semaphore

```
sem s := {0};
cobegin
  begin  //process A
    ... //statements
    P(s)
    ... //statements
  end;
  begin  //process B
    ... //statements
    V(s)
    ... //statements
  end;
coend;
```



Condition variables versus semaphores

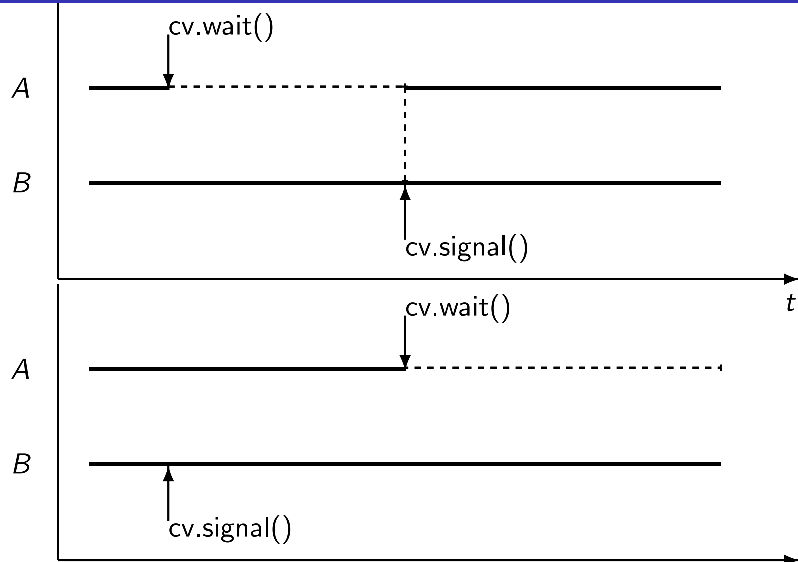
Condition variable

```
mn = monitor
  private:
    condition cv;
  public:
    procedure check;
    begin
      cv.wait();
    end;
    procedure set;
    begin
      cv.signal();
    end;
end;
```

```
mn m;   //monitor
cobegin
  begin  //process A
    ... //statements
    m.check(); // cv.wait()
    ... //statements
  end;
  begin  //process B
    ... //statements
    m.set(); // cv.signal()
    ... //statements
  end
coend;
```

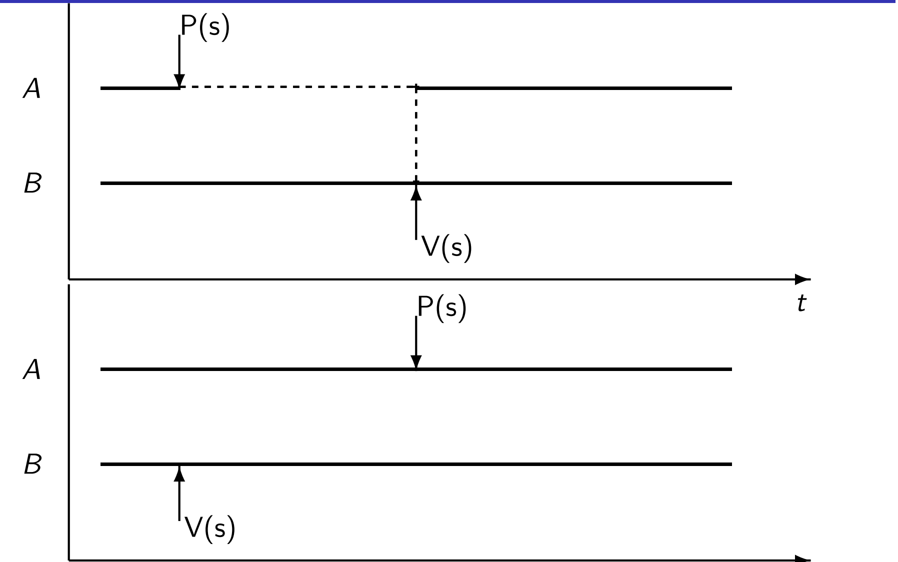
Condition variables versus semaphores

Condition variable cv



Condition variables versus semaphores

Semaphore s



Part IV

POSIX Threads

Outline

1. Operating system process vs. thread
2. Posix threads (pthreads)
3. Mutex
4. Condition variables
5. Semaphores

Operating System Process vs. Thread

- Operating system process: unit of resource management
  - virtual address space
  - access to resources:
    - computing core(s),
    - devices,
    - files
  - unit(s) of scheduling
- Thread: unit of scheduling
  - state (ready, idle, running, etc)
  - stack
  - access to resources of hosting process

Operating System Process vs. Thread

Performance advantages of threads

- Shorter time of creation
- Shorter time of destruction
- Shorter switching time
- Lower time overhead on communication (shared memory)
- Lower resource usage of the implementation

Implementation of Multithreading

- User-level Threads
  - Software library
  - Scheduling outside the operating system kernel
- Kernel-level Threads
  - System functions
  - Scheduling done by the operating system kernel

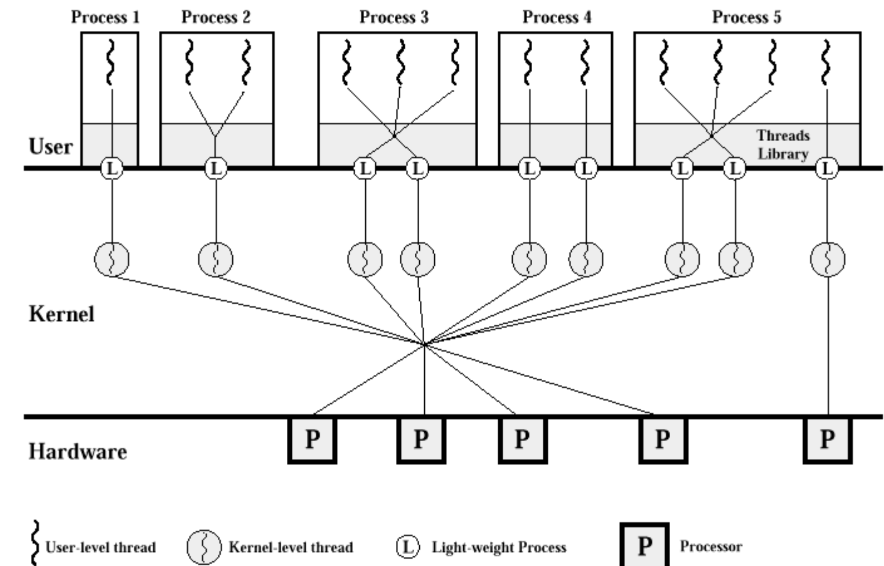
Implementation of Multithreading (cnt'd)

- User-level threads
  - Advantages
    - Very fast thread switching, creation and destruction
    - Customized scheduling
    - Easy porting of the application
  - Disadvantages
    - I/O operation blocks execution of all threads
    - Only one CPU used
- Kernel-level threads
  - Advantages
    - Possible use of many CPUs
    - Only one thread blocked on I/O
  - Disadvantages
    - Fixed scheduling algorithm
    - Application tied to OS
    - Time consuming thread switching, creating and destroying

Using User- and Kernel- Level Threads Together

- Benefits of user-level threads
  - Threads created in user mode
  - Majority of scheduling and synchronization done in user mode
  - API common for all supported OS
- Benefits of kernel-level threads
  - Kernel level concurrency
- Number of both kinds of threads set independently

Using User- and Kernel- Level Threads Together



Picture courtesy of Sun Microsystems. Used with permission.

POSIX Thread Library (libpthread)

- Thread creation and destruction
- Thread synchronization
- Scheduling
- Context management
- Operating system independence
- Thread attributes object
- Thread cancellation
- Implementation of fork – join – quit model

Thread attributes

- Scope
  - `PTHREAD_SCOPE_PROCESS` (default)
  - `PTHREAD_SCOPE_SYSTEM`
- Detachstate
  - `PTHREAD_CREATE_JOINABLE` (default)
  - `PTHREAD_CREATE_DETACHED`
- Stack address
  - default: NULL
- Stack size
  - default: 1MB
- Scheduling inheritance
  - `PTHREAD_INHERIT_SCHED` (default)
  - `PTHREAD_EXPLICIT_SCHED`
- Scheduling policy
  - `SCHED_OTHER` (default)
  - `SCHED_FIFO`
  - `SCHED_RR`

Creating a Thread

```
int pthread_create(pthread_t *tid, pthread_attr_t *tattr,
                   void*(*start_routine)(void *), void *arg);
```

```
#include <pthread.h>
pthread_t tid;
extern void *start_routine(void *arg);
void *arg = .....;
int ret;
ret = pthread_create(&tid, NULL, start_routine, arg);
```

Thread Termination

- Return from `start_routine`
- Termination of the **process**
- `void pthread_exit(void *status);`
- `int pthread_cancel(pthread_t thread);`

Joining a Thread

```
int pthread_join(pthread_t tid, void **status);
```

```
#include <pthread.h>
pthread_t tid;
int ret, status;
ret = pthread_join(tid, &status);
ret = pthread_join(tid, NULL);
```

```
int pthread_detach(pthread_t tid);
```

POSIX Thread Library vs. fork – join – quit model

fork
```
int pthread_create(pthread_t *tid, pthread_attr_t *tattr,
                   void*(*start_routine)(void *),
                   void *arg);
```

```
pthread_t tid;
extern void *start_routine(void *arg);
void *arg = .....;
int ret;
ret = pthread_create(&tid, NULL, start_routine, arg);
```

quit
```
void pthread_exit(void *status);
```

join
```
int pthread_join(pthread_t tid, void **status);
```

```
pthread_t tid = <id of a thread>;
int ret, status;
ret = pthread_join(tid, &status);
```

Remainig Functions

- Thread identifiers
  - `pthread_t pthread_self(void);`
  - `int pthread_equal(pthread_t tid1, pthread_t tid2);`
- `int sched_yield(void);`

Mutex

- MUtual Exclusion lock
- Using mutex
  - initialize
  - lock/unlock
  - destroy
- Attribute: scope
  - `PTHREAD_PROCESS_PRIVATE`
  - `PTHREAD_PROCESS_SHARED`

Setting up and Destroying a Mutex

- `pthread_mutex_t mp;`
- `pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;`
-
  ```
  pthread_mutex_t * mp;
  mp = calloc (1, sizeof (pthread_mutex_t));
  ```
- `int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);`
  ```
  pthread_mutex_t mp;
  pthread_mutex_init(&mp, NULL);

  pthread_mutex_t mp =  PTHREAD_MUTEX_INITIALIZER;
  ```
- `int pthread_mutex_destroy(pthread_mutex_t *mp);`
  ```
  pthread_mutex_t mp;
  int ret;
  ret = pthread_mutex_destroy(&mp);
  ```

Working with a Mutex

- `int pthread_mutex_lock(pthread_mutex_t *mp);`
  ```
  #include <pthread.h>
  pthread_mutex_t mp;
  int ret;
  ret = pthread_mutex_lock(&mp);
  ```
- `int pthread_mutex_unlock(pthread_mutex_t *mp);`
  ```
  #include <pthread.h>
  pthread_mutex_t mp;
  int ret;

  ...
  ret = pthread_mutex_unlock(&mp);
  ```
- `int pthread_mutex_trylock(pthread_mutex_t *mp);`

Mutex
An example

```
#include <stdio.h>
#include <thread.h>

void * change_g_data(void *null)   {
  static pthread_mutex_t g_mutex;
  static int             g_data = 0;
  pthread_mutex_lock(&g_mutex);
  g_data++;
  printf("%d is global data\n",g_data);
  pthread_mutex_unlock(&g_mutex);
  return NULL;
}
int i;
...
for ( i = 0; i < MAX; i ++)
  pthread_create(NULL, NULL, change_g_data, NULL);
```


Condition Variables

- Setting up
  - PTHREAD_PROCESS_PRIVATE
  - PTHREAD_PROCESS_SHARED
- pthread_cond_destroy
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_cond_signal
- pthread_cond_broadcast

Setting up and Destroying a Condition Variable

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Alternatively:

  ```
  pthread_cond_t cond;
  cond = calloc(1, sizeof (pthread_cond_t));
  ```
- `int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);`

  ```
  pthread_cond_t cv;
  ret = pthread_cond_init(&cv, NULL);

  cv  = PTHREAD_COND_INITIALIZER;
  ```
- `int pthread_cond_destroy(pthread_cond_t *cv);`

  ```
  pthread_cond_t cv;
  int ret;

  ret = pthread_cond_destroy(&cv);
  ```


pthread_cond_wait(pthread_cond_t *cv);

- `int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);`

  ```
  pthread_cond_t cv;
  pthread_mutex_t mutex;
  int ret;

  ...
  ret = pthread_cond_wait(&cv, &mutex);
  ```
- Usage scenario
  1. pthread_mutex_lock(m);
  2. while(condition_is_false)
     pthread_cond_wait(cv, m);
  3. pthread_mutex_unlock(m);

pthread_cond_signal(pthread_cond_t *cv);

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count() {
   pthread_mutex_lock(&count_lock);

   while (count == 0)
      pthread_cond_wait(&count_nonzero, &count_lock);
   count = count - 1;
   pthread_mutex_unlock(&count_lock);
}

increment_count() {
   pthread_mutex_lock(&count_lock);
   if (count == 0)
      pthread_cond_signal(&count_nonzero);
   count = count + 1;
   pthread_mutex_unlock(&count_lock);
}
```


pthread_cond_broadcast(pthread_cond_t *cv)

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount){
   pthread_mutex_lock(&rsrc_lock);
   while (resources < amount)
      pthread_cond_wait(&rsrc_add,&rsrc_lock);
   resources -= amount;
   pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount) {
   pthread_mutex_lock(&rsrc_lock);
   resources += amount;
   pthread_cond_broadcast(&rsrc_add);
   pthread_mutex_unlock(&rsrc_lock);
}
```


pthread_cond_timedwait

```
int pthread_cond_timedwait( pthread_cond_t *cv,
                            pthread_mutex_t *mp,
                            const struct timespec *abstime);
```

```
pthread_timestruc_t to;
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;
ret = pthread_cond_timedwait(&cv, &mp, &abstime);

//------------------------------------------------------------
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
  err = pthread_cond_timedwait(&c, &m, &to);
  if (err == ETIMEDOUT)
  break; //timeout
}
pthread_mutex_unlock(&m);
```


Semaphores

- Two kinds:
  - Named
  - Anonymous
- Common functions
  - `int sem_post(sem_t *sem);`
  - `int sem_wait(sem_t *sem);`
  - `int sem_trywait(sem_t *sem);`
  - `int sem_getvalue(sem_t *sem, int * sval);`

Named semaphores

- Name
  - begins with /
  - only one / allowed
  - recommended length: max 14 characters (PATH_MAX, NAME_MAX)
- `sem_t *sem_open( const char *name, int oflag, /*unsigned long mode, unsigned int value */ ...);`
  - oflag
    - O_CREAT
      required: mode, value
    - O_EXCL
  - SEM_FAILED returned in case of an error
- `int sem_close(sem_t *sem);`
- `int sem_unlink(const char *name);`

Anonymous semaphores

- `int sem_init( sem_t *sem, int pshared, unsigned int value);`
  - pshared == 0 => process private
  - pshared != 0 => process shared
- `int sem_destroy(sem_t *sem);`

Part V

Distributed memory

Outline

1. Message passing
   - Basic statements: send, receive
   - Names of processes
   - Synchronization of sender and receiver
   - Selective communication (guarded statements)
   - Channels

Message passing

- Basic statements
  - `send` *list of expressions* `to` *receiver(s)*
  - `receive` *list of variables* `from` *sender(s)*
- Assumption: communication is reliable
- Issues
  - How to specify sender(s) and receiver(s)
    - Names of (group of) processes
    - Communication „objects”: channels
  - Synchronization of sender and receiver

Names of processes

- `send`
  - $* \to 1$ $(1 \to 1,\ N \to 1)$:
    `send` *list of expressions* `to` *name of a receiver*;
  - $* \to M$ (broadcasting: $1 \to M$, $N \to M$):
    `send` *list of expressions*
    `to` *name of a set of receivers*;
- `receive`
  - $1 \to *$ $(1 \to 1,\ 1 \to M)$:
    `receive` *list of variables* `from` *name of a sender*;
  - $N \to *$ $(N \to 1,\ N \to M)$:
    `receive` *list of variables*
    `from` *name of a set of senders*;
    `receive` *list of variables*;

Names of processes

- `fork`, `join`, `quit`

  `fork` $P()$ `alias a;`
- `cobegin`

  `cobegin`
  `as` *processName1 statement 1*;
  `as` *processName2 statement 2*;
  ⋮
  `coend`
- `parfor`

  `parfor` $i :=$ *low_value* `to` *high_value* `as` *name*`<i>` `do`
  S_i;
  `end parfor;`

Producers and Consumers

```
cobegin //1->1
  as Producer
    var int lx;
    loop
      lx := Create();
      send lx to Consumer;
    end loop;

  as Consumer
    var int lx;
    loop
      receive lx from Producer;
      Use (lx);
    end loop;
coend;
```

```
cobegin //N->1
  parfor i := 1 to N do
    var int lx;
    loop
      lx := Create();
      send lx to Consumer;
    end loop;
  end parfor;

  as Consumer
    var int lx;
    loop
      receive lx;
      Use (lx);
    end loop;
coend;
```

Computation of a minimum

Problem Distributed memory parallel computer runs $N+1$ concurrent processes $P_0, P_1, \ldots, P_N$. Each of them executes the following code:

```
val int x;
begin
  x := Create();
  x := findMin (x, i);
  Use (x);
end;
```

The procedure *findMin* (int x, int i):

- Input arguments:
  - x a value created in current process,
  - i id of the current process.
- Result: the lowest value (a minimum) of all the values x created in the processes P_0, P_1, ..., P_N.

Develop procedure *findMin*.

Computation of a minimum

```
procedure findMin (int x, int i);
begin
  if i = 0 then
    val int j, xTmp;
    for j := 1 to N do
      receive xTmp;
      if xTmp < x then
        x := xTmp;
      end if;
    end for;
    for j := 1 to N do
      send x to P<j>;
    end for;
  else
    send x to P<0>;
    receive x from P<0>;
  end if;
  return x;
end;
```

```
parfor i := 0 to N as P<i> do
  val int x;
  begin
    x := Create ();
    x := findMin (x, i);
    Use (x);
  end; //P<i>
end parfor;
```

Computation of a minimum

Using an additional named process

```
procedure findMin (int x, int i);
//i unused
begin
  send x to New;
  receive x from New;
  return x;
end;

cobegin
  parfor i := 0 to N as P<i> do
    val int x;
    begin
      x := Create ();
      x := findMin (x, i);
      Use (x);
    end; //P<i>
  end parfor;
```

```
  var int j, x := MaxInt, xTmp;
  as New begin
    for j := 0 to N do
      receive xTmp;
      if xTmp < x then
        x := xTmp;
      end if;
    end for;
    for j := 0 to N do
      send x to P<j>;
    end for;
  end; //New
coend;
```

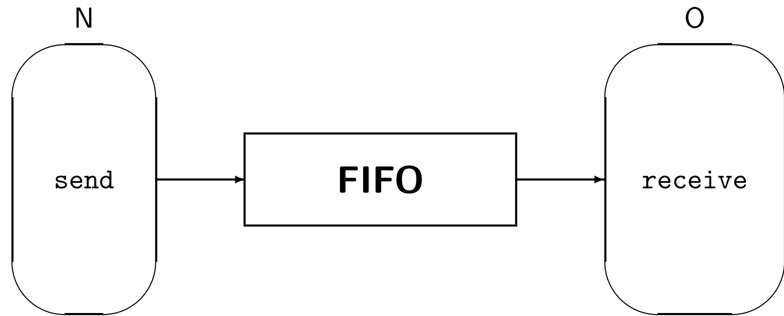
Computation of a minimum

Without an additional named process

```
procedure findMin (int x, int i);
var int j, k, xp[N+1];
begin
  cobegin //shared memory
    parfor k := 0 to N do
      send x to P<k>;
    end parfor;
    parfor j := 0 to N do
      receive xp[j] from P<j>;
    end parfor;
  coend;
  x := xp[0];
  for  j:= 1 to N do
    if xp[j] < x then
      x := xp[j];
    end if;
  end for;
  return x;  end;
```

```
parfor i := 0 to N as P<i> do
  val int x;
  begin
    x := Create ();
    x := findMin (x, i);
    Use (x);
  end; //P<i>
end parfor;
```

Synchronization of sender and receiver



n	Type	send	receive
∞	Asynchronous	Not blocking	Blocking
0	Synchronous	Blocking	Blocking
$0 < n < \infty$	Buffered	Blocking	Blocking

Selective communication

```
while true do
  receive list1 from sender1;
  receive list2 from sender2;
    ⋮
  receive listN from senderN;
end while;
```

```
while true do
  if receive from sender1 does not block then
    receive list1 from sender1;
  if receive from sender2 does not block then
    receive list2 from sender2;
      ⋮
  if receive from senderN does not block then
    receive listN from senderN;
end while;
```


Guards

guard $\longrightarrow$ *a statement*

Guard =

- boolean expression
- boolean expression and message passing statement.

States of a guard:

- false,
- true,
- third state (neither true nor false).

Guards

Examples

- `x < 0 -> x := -x;`
- ```
  neg := 0;
  for i := 0 to N - 1 do
    a[i] < 0 ->
      neg ++;
  end for;
  ```
- ```
  free, receive x from producer ->
    free := false;
  ```
- ```
  true, send 1 to consumer ->
    ;
  ```
  ```

guarded if statement

guarded if
 $guard_1 \longrightarrow statement_1$
 $guard_2 \longrightarrow statement_2$
 $\vdots$
 $guard_N \longrightarrow statement_N$
end guarded if

```
var
  int x;
  bool free := true;
guarded if
  free, receive x from P ->
    free := false;
  !free, send x to C ->
    free := true;
end guarded if;
```

Replicated guarded if statement

guarded if *idx* := *low_value* to *high_value*
 guard $\longrightarrow$ *statement*
end guarded if

```
var int x;
guarded if i := 1 to 10
  receive x from P<i> ->
    ;
end guarded if;
```

guarded do statement

guarded do
 $guard_1 \longrightarrow statement_1$
 $guard_2 \longrightarrow statement_2$
 $\vdots$
 $guard_N \longrightarrow statement_N$
end guarded do

```
var
  int x;
  bool free := true;
guarded do
  free, receive x from P ->
    free := false;
  !free, send x to C ->
    free := true;
end guarded do;
```

Replicated guarded do statement

guarded do *idx* := *low_value* to *high_value*
 guard $\longrightarrow$ *statement*
end guarded do

```
var   bool rcv[1 .. N];
      int x, sum := 0;
begin
  for i := 1 to N do
    rcv[i] := true;
  end for;
  guarded do i := 1 to N
    rcv[i], receive x from P<i> ->
    begin
      rcv[i] := false;
      sum += x;
    end;
  end guarded do;
end
```

Buffer

```
procedure buffer;
var
  T elem[0 .. N - 1];
  int head := 0, tail := 0, taken := 0;
begin
  guarded do
    taken < N, receive elem[head] from P →
    begin
      taken ++;
      head := (head + 1) % N;
    end;

    taken > 0, send elem[tail] to C →
    begin
      taken −−;
      tail := (tail + 1) % N;
    end;
  end guarded do;
end;
```


Producers and Consumers

```
procedure producer;
var
  T message;
begin
  while true do
    message := Create();
    send message to B;
  end while;
end;
```

```
procedure consumer;
var
  T message;
begin
  while true do
    receive message from B;
    Use (message);
  end while;
end;

cobegin
  as B buffer();
  as P producer();
  as C consumer();
coend;
```


Producers and Consumers

Without send in a guard

```
procedure buffer;
var
  T elem[0 .. N - 1];
  int head := 0, tail := 0, taken := 0;
begin
  guarded do
    taken < N, receive elem[head] from P →
    begin
      taken ++;
      head := (head + 1) % N;
    end;
    taken > 0, receive from C →
    begin
      send elem[tail] to C;
      taken −−;
      tail := (tail + 1) % N;
    end;
  end guarded do;
end;
```

```
procedure consumer;
var
  T msg;
begin
  while true do
    send to B;
    receive msg from B;
    Use (msg);
  end while;
end;
```


Producers and Consumers ($N \rightarrow 1$)

```
cobegin
  parfor i := 1 to N as P<i> do
    var int lx;
    loop
      lx := Create();
      send lx to Consumer;
    end loop;
  end parfor;
  as Consumer  var int lx;
               loop
                 guarded if i := 1 to N
                   true, receive lx from P<i> ->
                     ; //NOP
                 end guarded if ;
                 Use (lx);
               end loop;
coend
```

Channels

- 1 — 1
- uni- or bidirectional
- `channel` *message structure* `end;`
- `send` *list of expressions* `to` *identifier*;
  `receive` *list of variables* `from` *identifier*;

Data mismatch problem

```
cobegin
  as sender
    send 7UL to recipient;
  as recipient
    var float y;
    receive y from sender;
coend
```

Channel solution

```
var
  channel
    unsigned long;
  end c;
cobegin
  send 7UL to c;

  var float y;
  receive y from c;
coend;
```


Producers and Consumers (1 $\longrightarrow$ 1)

```
var
  channel
    int;
  end cpm;
cobegin

    var int lx;
    loop
      lx := Create();
      send lx to cpm;
    end loop;


    var int lx;
    loop
      receive lx from cpm;
      Use (lx);
    end loop;

coend
```


Producers and Consumers (N $\longrightarrow$ 1)

```
var
  channel int; end cpm[1 .. N];
cobegin

   parfor i := 1 to N do
     loop
       send Create() to cpm[i];
     end loop;
   end parfor;

   var int lx;
   loop
     guarded if i:= 1 to N do
       true, receive lx from cpm[i]->
         Use (lx);
     end guarded if;
   end loop;

coend
```


Part VI

OpenMP

Outline

1. Matrix multiplication example
   - sequential solution
   - pthreads solution
   - OpenMP solution
2. Introduction to OpenMP
3. Expressing parallel execution
4. Work sharing
5. Data storage attributes
6. Synchronization
7. Nested parallelism
8. Using third party libraries

Matrix multiplication

Sequential version: pseudocode

```
for i := 1 to n do
  for j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end for;
end for;
```


Matrix multiplication

Sequential version: C implementation

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;


  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

File MATRIX-SEQ.C.

Matrix multiplication

PRAM CREW

```
parfor i := 1 to n do
  parfor j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end parfor;
end parfor;
```

Matrix multiplication

(AGAIN) Sequential version: pseudocode

```
for i := 1 to n do
  for j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end for;
end for;
```

Matrix multiplication

Parallel version: C + pthread library

Facts

- Reasonable size of matrices ($n \geq 64$)
- Thread creation is not free
- Computing power comes from a CPU not from a thread
- Computing system with small to medium number of CPUs

Decision

Thread i computes rows $i, i + NT, i + 2NT, i + 3NT, \ldots$ of the result matrix C
(NT — Number of Threads)

Matrix multiplication

Parallel version: C + pthread library (part I). File MATRIX-PTHREADS.C.

```
typedef struct {
  double (*a) [N];
  double (*b) [N];
  double (*c) [N];
  sem_t * s;
  int id;
} thread_args;

void * thread_proc (void * a){
  thread_args * arg = a;
  int i, j, s;
  for (i = arg->id; i < N; i += NT)
    for (j = 0; j < N; j ++){
      arg->c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        arg->c[i][j] += arg->a[i][s] * arg->b[s][j];
    }
  assert (sem_post (arg->s) == 0);
  return NULL;
}
```

Matrix multiplication

Parallel version: C + pthread library (part II)

```
void matrix_mult (double a [N][N], double b[N][N], double c[N][N]){
  int i;
  thread_args ta[NT];
  sem_t s;
  pthread_t p;

  assert (sem_init (&s, 0, 0) == 0);
  for (i = 0; i < NT; i ++){
    thread_args x = { a, b, c, &s, i};
    ta[i] = x;
    assert (0 == pthread_create (&p, NULL, thread_proc, ta + i));
  }
  for (i = 0; i < NT; i ++)
    assert (sem_wait (&s) == 0);
}
```

.

Matrix multiplication
Parallel version: C + OpenMP

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;

  #pragma omp parallel for private(j, s)
  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

File: MATRIX-OMP.C

Matrix multiplication
Sequential version: C implementation

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;


  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

File: MATRIX-SEQ.C

OpenMP

- A way to do multithreaded programming
- Focus: parallel execution of loops
- OpenMP defines
  - set of compiler directives
  - (small) set of subroutines
  - bindings for programming languages (Fortran, C, C++)
- OpenMP extends, but does not modify, a base programming language

OpenMP

- Goals
  - Scalable parallel applications for shared memory computers
  - Portability of performance
  - Fast and gradual development of parallel versions of existing sequential code
  - Easy simultaneous maintenance of parallel and sequential source code
  - Development of new parallel applications from scratch
- History
  - Nov 1996 — spec. ANSI X3H5
  - Oct 1997 — spec. 1.0 OpenMP (FORTRAN)
  - 1998 — spec. 1.0 OpenMP (C/C++)
  - 1999 — spec. 1.1 OpenMP (FORTRAN)
  - 2006 — spec. 2.5 compilers available (FORTRAN, C, C++)
  - Feb 2008 — spec. 3.0 defined
- OpenMP homepage: `www.openmp.org`
- OpenMP user group: `www.cOMPunity.org`

OpenMP enabled compilers

- Intel (Linux, MSWindows)
  - C++ compiler ver 9.x: spec. 2.5 OpenMP
  - FORTRAN compiler
- Microsoft Visual Studio 2005, Visual .NET: spec. 2.0 OpenMP (MSWindows)
- GNU C++ ver. 4.2.0 (TBA)
- Sun Studio ver 11 (Solaris)
- PGF77 and PGF90 Compilers from The Portland Group, Inc. (PGI) (Intel Linux, Intel Solaris, Intel Windows/NT)
- IBM XL Fortran and C (IBM AIX)
- HP, SGI, Fujitsu

Execution of OpenMP programs

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;

  #pragma omp parallel for private(j, s)
  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

File: MATRIX-OMP.C

Coding OpenMP programs

- Complier directives
  - Format
    `#pragma omp` *directive* `[`*clause*`]`
  - Parallel region
    `#pragma omp parallel [`*clause*`]`
  - Work sharing
  - Synchronization
- Runtime library
  - `#include <omp.h>`
  - Query functions
  - Lock functions
  - Dynamic adjustment of number of threads
  - Nested parallelism
- Environment variables
  - OMP_DYNAMIC
  - OMP_NUM_THREADS
  - OMP_NESTED
  - OMP_SCHEDULE

Expressing parallelism

- parallel directive
```
#pragma omp parallel
{
  printf("hello world\n");
}
```
- Useful query functions
  - Getting current thread number: omp_get_thread_num()
  - Getting number of threads: omp_get_num_threads()
  - Detecting parallel region: omp_in_parallel ()
- Setting number of threads
  - application wide
    - function omp_set_num_threads()
    - OMP_NUM_THREADS environment variable
  - region wide
    num_threads clause
- Dynamic adjustment of the number of threads
  - function omp_set_dynamic ()
  - function omp_get_dynamic ()
  - OMP_DYNAMIC environment variable

Work sharing directives

- sections directive
- combined parallel sections directive
- for directive
- combined parallel for directive
- single directive

```
#pragma omp single
{
  doSth (defaultDelay);
  single2 = omp_get_thread_num();
}
```

- master directive

```
#pragma omp master
master = omp_get_thread_num();
```


sections directive
Syntax

```
#pragma omp sections [clause]
 {
   [ #pragma omp section ]
     block1
   [ #pragma omp section
     block2 ]
   [ #pragma omp section
     block3 ]
   ...
 }
```

```
cobegin
    block1
    block2
    block3
    ...
coend;
```


sections directive
An example

```
#pragma omp parallel
{
  #pragma omp sections
  {
    printf ("This code has been executed by thread %d of %d\n",
             omp_get_thread_num (), omp_get_num_threads());

    #pragma omp section
    printf ("This code has been executed by thread %d of %d\n",
             omp_get_thread_num (), omp_get_num_threads());

    #pragma omp section
    {
      printf ("This code has been executed by thread %d of %d\n",
               omp_get_thread_num (), omp_get_num_threads());
    }
    #pragma omp section
    {
      printf ("This code has been executed by thread %d of %d\n",
                omp_get_thread_num (), omp_get_num_threads());
    }
  }
}
```


sections directive
sections vs pure parallel

```
#pragma omp parallel
{
    printf ("This code has been executed by thread %d of %d\n",
              omp_get_thread_num (), omp_get_num_threads());
}
```

sections
- Exactly 4 messages
- Thread numbers printed is any 4 element permutation with repetitions of integers $[0 \ldots teamSize - 1]$

parallel
- Exactly *teamSize* messages
- Thread numbers printed is a permutation without repetitions of integers $[0 \ldots teamSize - 1]$

parallel sections directive

```
#pragma omp parallel sections
{
  printf ("This code has been executed by thread %d of %d\n",
          omp_get_thread_num (), omp_get_num_threads());

  #pragma omp section
  printf ("This code has been executed by thread %d of %d\n",
          omp_get_thread_num (), omp_get_num_threads());

  #pragma omp section
  {
    printf ("This code has been executed by thread %d of %d\n",
            omp_get_thread_num (), omp_get_num_threads());
  }
  #pragma omp section
  {
    printf ("This code has been executed by thread %d of %d\n",
             omp_get_thread_num (), omp_get_num_threads());
  }
}
```

for directive

- Syntax

  `#pragma omp for [`*clause*`]`
  for-loop
- Example

```
int a[ARR_SIZE];
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i < ARR_SIZE; i ++)
      a[i] = 1;
}
```

parallel for directive

schedule clause

- Default scheduling: implementation dependent
- `schedule(static, chunk_size)`
  - chunks of a `chunk_size` size or equal size (if `chunk_size` not provided)
  - round-robin in the order of the thread number,
- `schedule(dynamic, chunk_size)`
  - chunks of a `chunk_size` size or 1 (if `chunk_size` not provided)
  - a queue of chunks
- `schedule(guided, chunk_size)`
  - chunks of a decreasing size (down to `chunk_size`) or 1 (if `chunk_size` not provided)
  - a queue of chunks
- `schedule(runtime)`
  scheduling according to the value of `OMP_SCHEDULE` environment variable

parallel for directive

- Syntax
  `#pragma omp parallel for [`*clause*`]`
  for-loop
- Example

```
double a [N][N], b [N][N], c [N][N];
int i;
...
#pragma omp parallel for
for (i = 0; i < N; i ++){
  int j, s;
  for (j = 0; j < N; j ++){
    c[i][j] = 0.0;
    for (s = 0; s < N; s ++)
      c[i][j] += a[i][s] * b[s][j];
  }
}
```

Data model

- Variables can be
  - Shared
  - Private
  - Subject to `reduction`
- Defaults
  - Usually: shared
  - Private
    - Local (**not** `static`) variables of subroutines called by a thread
    - Automatic variables within a block of thread's code
    - Control variable of parallel `for` loop

Data model
Declaration

- Clauses
  - `shared` (*list*)
  - `private` (*list*)
  - `default` (*list*)
  - `firstprivate`
  - `lastprivate`
  - `reduction` (*operator* : *list*)
- Thread private variables
  - `threadprivate` directive
  - `copyin` clause

Data model
Example (from MATRIX-OMP.C)

```
void matrix_mult2(){
  double a [N][N], b [N][N], c [N][N];
  int i;
  ...
  #pragma omp parallel
  {
    #pragma omp for
    for (i = 0; i < N; i ++){
      int j, s;
      for (j = 0; j < N; j ++){
        c[i][j] = 0.0;
        for (s = 0; s < N; s ++)
          c[i][j] += a[i][s] * b[s][j];
      }
    }
  }
}
```

Data model
Example (from MATRIX-OMP.C)

```
void matrix_mult3(){
  double a [N][N], b [N][N], c [N][N];
  int i, j, s;
  ...
  #pragma omp parallel
  {
    #pragma omp for private(j, s)
    for (i = 0; i < N; i ++){
      for (j = 0; j < N; j ++){
        c[i][j] = 0.0;
        for (s = 0; s < N; s ++)
          c[i][j] += a[i][s] * b[s][j];
      }
    }
  }
}
```

Data model
Consistency of shared data

- Sequential consistency
  - benefits
  - drawbacks
- Relaxed consistency
  - temporary view
  - synchronization points
    - barriers (both explicit and implicit)
    - entry to and exit from `critical` regions
    - entry to and exit from lock routines
  - directive `flush ( list )`

reduction clause

- Syntax: `reduction (`*operator* `:` *list* `)`
- List of operators: `+, *, -, &, ^, |, &&, ||`
- Example:

```
#define ARR_SIZE 100000
int a [ARR_SIZE], i, sum;
...
sum = 0;
#pragma omp parallel
{
  #pragma omp for reduction (+: sum)
  for (i = 0; i < ARR_SIZE; i ++)
    sum += a[i];
}
```

Synchronization

- Implicit synchronization
  - `for` directive
  - `sections` directive
  - `single` directive

  `nowait` clause
- Explicit synchronization (directives)
  - `master`
  - `critical`
  - `atomic`
  - `barrier`
  - `ordered` directive and `ordered` clause
- Explicit synchronization (locks)

nowait example

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < numThreads; i++)
    doSth(i * delay);

  #pragma omp sections
  {
    printf("Thread #%d in sections\n", omp_get_thread_num());

    #pragma omp section
    printf("Thread #%d in sections\n", omp_get_thread_num());

    #pragma omp section
    printf("Thread #%d in sections\n", omp_get_thread_num());
  }
}
```

critical directive

- Syntax

```
#pragma omp critical [ (name)]
    block
```

- Example

```
#define ARR_SIZE 100000
int a [ARR_SIZE], i, sum;
...
sum = 0;
#pragma omp parallel for
for (i = 0; i < ARR_SIZE; i ++)
  #pragma omp critical
  sum += a[i];
printf ("Sum == %d\n", sum);
```

Named critical directive

```
int cs_even = 0, cs_odd = 0;
...
#pragma omp parallel for num_threads (numThreads)
for (i = 0; i < numThreads; i ++)
  if (i % 2)
    #pragma omp critical (odd)
    cs_odd ++;
  else
    #pragma omp critical (even)
    cs_even ++;
```

ordered example

```
#pragma omp parallel for ordered
for (i = 0; i < numThreads; i ++)
{
  do_sth_in_any_order();
  #pragma omp ordered
  doSth(); //it will be done in ordered manner
}
```

Remaining issues

- Nested parallelism
  - function omp_set_nested ()
  - function omp_get_nested ()
  - OMP_NESTED environment variable
- Using third part libraries

Part VII

Message Passing Interface

Outline

1. Introduction
   - goals
   - history
   - supporters
   - features
   - implementations
2. Basics of MPI programming
3. Point-to-point communication
   - fundamentals
   - blocking communicaton
   - non-blocking communicaton

MPI Goals and History

1. Goals
   - standard message-passing interface
   - source code portability
   - efficient implementations
2. History
   2.1 MPI-1 Forum
      - MPI 1.0 — June 1994
      - MPI 1.1 — June 1995

   2.2 MPI-2 Forum
      - MPI 1.2 — mainly clarifications to MPI 1.2
      - MPI 2.0 — extensions to MPI 1.2 (1997)

MPI Supporters

1. Universities (US, Europe)
2. US National Laboratories (Sandia, Argonne, Lawrence Livermore, Los Alamos, Oak Ridge)
3. IT Industry
   - Cray Research
   - HP
   - Hitachi
   - Intel
   - IBM
   - NEC
   - SGI
   - SUN, . . .
4. non-IT
   - General Electric
   - Pratt & Whitney
   - Lockheed-Martin
   - NASA, . . .

MPI Feature List

- Naming
- Point-to-point communication
  - send and receive model
  - synchronous and asynchronous communication
  - blocking and non-blocking communication
  - sendrecv*
- Groups of processes
- Collective communication
  - barrier synchronization
  - broadcast
  - scatter/gather
  - global reduction
- Virtual topologies
- Data representation
- ...

MPI Implementations

- Computer vendors
  - Cray Research
  - HP
  - Hitachi
  - Intel
  - IBM
  - NEC
  - SGI
  - SUN
  - ...
- Universities
- Open source
  - MPICH
  - LAM-MPI
  - OpenMPI
  - ...
- Comprehensive list:
  `http://www.lam-mpi.org/mpi/implementations`

MPI Programming

The paradigm

- Hardware: distributed memory
- Software
  - program written in conventional sequential language
  - usually the same program on every node (SPMD)
    What to do if you need MPMD?
  - communication coded explicitly by means of special routines (e.g. MPI)

Practice

- MPI deamon on every node
- Link your code with an MPI library
- Start the program using MPI startup tool
  - `mpiexec -n` *number_of_processes executable.file*
  - `mpirun`

What to do if you need MPMD?

- Use MPI implementation that supports MPMD
- DYI

```
int main (int argc, char **argv){

  switch(what_to_do){
    case DRINK: drink();
                break;
    case SING:  sing();
                break;
    case SLEEP: sleep();
                break;
    default:    assert (0);
  }
}
```

MPI First Program

```
#include <mpi.h>
#include <stdio.h>
main(int argc, char ** argv){

  MPI_Init (&argc, &argv);
  printf ("Hello world!\n");
  MPI_Finalize ();
}
```

- Consistent structure of MPI identifiers: MPI_xxxxxxxx
- Functions return execution status: `MPI_SUCCESS`, error codes
- `int MPI_Init (int *argc, char ***argv);`
  The first MPI routine to be called in every process
- `int MPI_Finalize (void);`
  To be called last in every process

MPI First Program (II)

File: HELLO_WORLD.C

```
#include <assert.h>
#include <mpi.h>
#include <stdio.h>
main(int argc, char ** argv){

  assert (MPI_Init (&argc, &argv) == MPI_SUCCESS);
  printf ("Hello world!\n");
  assert (MPI_Finalize () == MPI_SUCCESS);
}
```


Reading MPI Documentation

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
[ IN buf] initial address of send buffer (choice)
[ IN count] number of elements in send buffer
            (nonnegative integer)
[ IN datatype] datatype of each send buffer element
               (handle)
[ IN dest] rank of destination (integer)
[ IN tag] message tag (integer)
[ IN comm] communicator (handle)

int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```


Reading MPI Documentation (II)

MPI Basic Data Types

MPI Constant	C type
MPI_CHAR	char
MPI_BYTE	like unsigned char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	long double
MPI_LONG_LONG_INT	long long

Communicator

- Communicator
- MPI_COMM_WORLD
  - predefined in MPI.H
  - contains all processes of a single execution of a MPI program
- How many processes within a communicator?
  `int MPI_Comm_size (MPI_Comm comm, int * size);`
- Rank of a process
  - ID of a process in the context of a communicator
  - Range [0, .., *communicator_size* - 1]
  - Application: work and data distribution, MPMD emulation, etc.
  - `int MPI_Comm_rank (MPI_Comm comm, int * rank);`
- Handles
  - Identify MPI objects
  - Constants defined in MPI.H; example: MPI_COMM_WORLD
  - Special MPI typedefs (in C/C++); example: MPI_Comm

MPI Second Program

File: HELLO_WORLD2.C

```
#include <assert.h>
#include <mpi.h>
#include <stdio.h>
main(int argc, char ** argv){

  int s, r;
  assert (MPI_Init (&argc, &argv) == MPI_SUCCESS);
  assert (MPI_Comm_size(MPI_COMM_WORLD, &s) == MPI_SUCCESS);
  assert (MPI_Comm_rank(MPI_COMM_WORLD, &r) == MPI_SUCCESS);
  printf ("Hello world from process %d out of %d\n", r, s);
  assert (MPI_Finalize () == MPI_SUCCESS);
}
```


Point-to-Point Communication

1. Two processes
   - source process sends
   - destination process receives
2. Always within a single communicator
3. How processes are identified?
4. Blocking vs non-blocking communication
5. Communication modes: synchronous, buffered, standard and ready
6. How many functions to learn?
   - blocking and non-blocking: 2
   - communication modes: 4
   - send and receive: 2
   - Total: 2*4*2 = 16
   - Good news: one receive for all communication modes! (But you have to learn additional stuff anyway).

Basics of Point-to-Point Communication

```
int MPI_Send (void *buf, int count, MPI_Datatype datatype,
              int rank, int tag, MPI_Comm comm);
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
              int rank, int tag, MPI_Comm comm,
              MPI_Status * status);
```

- buf, count, datatype — description of the data buffer
- comm — communicator
- rank — rank of corresponding process or MPI_ANY_SOURCE
- tag — tag of a message or MPI_ANY_TAG
- status — information about received message
  - rank of the source: status.MPI_SOURCE
  - tag of the message: status.MPI_TAG
  - count of received elements: via MPI_Get_count
    `int MPI_Get_count ( MPI_Status * status, MPI_Datatype datatype, int *cnt)`

Basics of Point-to-Point Communication

- **Requirements**
  - Valid ranks
  - The same communicator
  - Matching tags
  - Matching data types
  - Large enough destination buffer
- **Message order preservation**
  - Guaranteed within a single connection
  - Connection = communicator + source rank + destination rank
- **Example**
  Function `de1()` in file DATA_EXCHANGE.C

Communication Modes

- Sends

Synchronous	completes when the receive has started
Buffered	▸ always completes ▸ needes a buffer provided by the application (`MPI_Buffer_attach`)
Standard	▸ either Synchronous or Buffered (implementation decides) ▸ uses internal buffer
Ready	requires the matching receive to have already been posted

- Receive: one for all sends

Blocking Point-to-Point Communication Routines

Sends

```
int MPI_□end (void *buf, int count, MPI_Datatype datatype,
              int rank, int tag, MPI_Comm comm);
```

Synchronous	`MPI_Ssend`
Buffered	`MPI_Bsend`
Standard	`MPI_Send`
Ready	`MPI_Rsend`

Receive

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int rank, int tag, MPI_Comm comm,
             MPI_Status * status);
```


Buffered blocking communication

- `int MPI_Buffer_attach (void *buffer, int size)`
- `int MPI_Buffer_detach (void *bufferptr, int *size)`
- **Example**
  Function `de2()` in file DATA_EXCHANGE.C

Non-Blocking Communication

- Blocking communication
  - One step: just `MPI_*send` or `MPI_Recv`
  - Data buffers ready for reuse
- Two steps
  1. Initiate communication `MPI_I*`
  2. Wait for the communication to complete `MPI_Wait*`, `MPI_Test*`
- Data buffers must not be reused between step 1 and 2
- Any processing may be performed between step 1 and 2
- All communication modes apply
- Mixing blocking and non-blocking allowed: any send works with any receive

Non-blocking Point-to-Point Communication Routines

Sends

```
int MPI_I□end (void *buf, int count, MPI_Datatype datatype,
               int rank, int tag, MPI_Comm comm,
               MPI_Request * reqHandle);
```

Synchronous `MPI_Issend`
Standard `MPI_Isend`
Ready `MPI_Irsend`

Receive

```
int MPI_Irecv (void *buf, int count, MPI_Datatype datatype,
               int rank, int tag, MPI_Comm comm,
               MPI_Request * reqHandle);
```


Completion

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request,            MPI_Status *status);

int MPI_Testany(int count, MPI_Request array_of_requests[],
                int *index, int *flag, MPI_Status *status);
int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int *index,            MPI_Status *status);

int MPI_Testall(int count, MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[]);
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                           MPI_Status array_of_statuses[]);

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
```

Example: Function de3() and de4() in file DATA_EXCHANGE.C

Built-in Data Exchange

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,

                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,

                 MPI_Comm comm, MPI_Status *status);

int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,

                         int dest, int sendtag,

                         int source, int recvtag,

                         MPI_Comm comm, MPI_Status *status)
```

Example
Function de5() in file DATA_EXCHANGE.C