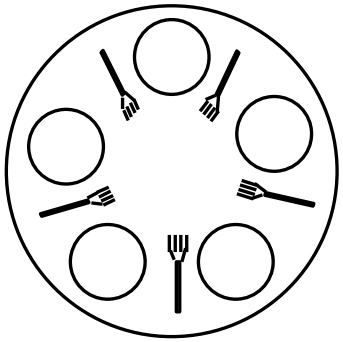
English Edition

5. Synchronization (3)

5. Synchronization
5.3 Synchronization methods

Dining Philosophers (1)

- Five philosophers at one table (Dijkstra 1965)
  - Each philosopher has a plate of spaghetti.
  - Between two plates there is one fork.
  - Each philospher changes state between thinking and eating
  - In order to eat, a philosopher requires both forks, left and right of his plate.



- Task: – don't let the philosophers starve
  – attain maximal level of parallelism

Dining Philosophers (2)

- first idea:

```
#define  N  5                    // number of philoshphers
philosopher (int i) {            // i: index of the philosopher (0...N-1)
  while (TRUE)  {
    think();                     // philosopher thinks
    take_fork(i);                // take left fork
    take_fork( (i+1)%N );        // take right fork (% = modulo)
    eat();                       // philosopher eats
    put_fork (i);                // return left fork
    put_fork ( (i+1)%N );        // return right fork
  }
}
```

  - take_fork(i) blocks the thread until the fork is available
  - solution is wrong: deadlock occurs when all philosophers simultaneously take the left fork

Dining Philosophers (3)

- 1st correction attempt:
  - each philosopher checks (after taking the left fork) whether the right fork is available
  - if it is not, he puts back the left fork, waits for a short time and takes the left fork again
- if waiting time is random, this could work „often“ (not good enough)
- in case of equal waiting times there could be an infinite loop (all philosophers take left fork, put it down, take it again etc.) ➡ „starvation“

Dining Philosophers (4)

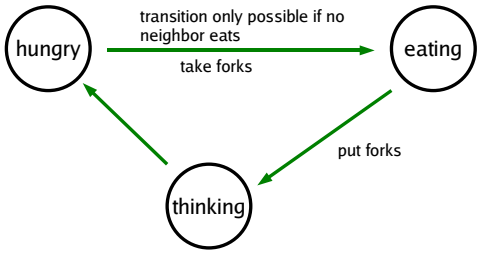
- 2nd correction attempt:
  - protect whole block from taking the first work to putting them both down with a mutex, i. e.

```
while (TRUE)  {
    think();
    wait (mutex);      // enter critical region
    take_fork(i);
    take_fork( (i+1)%N );
    eat();
    put_fork (i);
    put_fork ( (i+1)%N );
    signal (mutex);    // leave critical region
  }
```

  - OK, but not efficient: only one philosopher can eat at any given time – but five forks would allow two to eat

Dining Philosophers (5)

- correct solution:
  - save philosophers' states in array *state[ ]*



  - semaphore *sem[i]* for each philosopher: blocks if one fork (or both) is unavailable

Dining Philosophers (6)

```
#define N 5                 // number of philosophers
#define LEFT  (i-1)%N       // index of left neighbor
#define RIGHT (i+1)%N       // index of right neighbor
#define THINKING 0          // philosopher thinks
#define HUNGRY   1          // philosopher tries , Gabeln zu nehmen
#define EATING   2          // philosopher eats

typedef int semaphore;      // semaphores are special "int"s
int       state[N];         // vector for states
semaphore mutex=1;          // semaphore (mutex) for mutual exclusion
                            // of access to vector state
semaphore sem[N]={0};       // one semaphore per philosopher

philosopher (int i)         // i: which philosopher (0 -- N-1)
{
  while (TRUE)  {           // infinite loop
    think ();               // philosopher thinks
    take_forks (i);         // take both forks or block
    eat ();                 // eat
    put_forks (i);          // put both forks
  }
}
```


Dining Philosophers (7)

```
take_forks (int i) {     // i: which philosopher (0 to N-1)
  wait (mutex);          // enter critical region
  state[i] = HUNGRY;     // philosopher is hungry
  test (i);              // try to get both forks
  signal (mutex);        // leave critical region
  wait (sem[i]);         // block if cannot get both forks
}

put_forks (int i) {      // i: which philosopher (0 to N-1)
  wait (mutex);          // enter critical region
  state[i] = THINKING;   // philosopher finished with eating
  test (LEFT);           // test whether left phil. can and wants to eat
  test (RIGHT);          // test whether right phil. can and wants to eat
  signal (mutex);        // leave critical region
}

test (int i) {           // test whether phil. i can and wants to eat
  if ( state[i]     == HUNGRY &&
       state[LEFT]  != EATING &&
       state[RIGHT] != EATING ) {
    state[i] = EATING;
    signal (sem[i]);     // philosopher i can eat now, so wake him up
  }
}
```

Example: Phil. 2 and 3 want to eat

```
// i=2, sem[2]=0

think();
take_forks (2);
   wait (mutex);
   state[2] = HUNGRY;
   test (2);
      state[2] == HUNGRY?  yes
      state[1] != EATING?  yes
      state[3] != EATING?  yes
      ->
      state[2] = EATING;
      signal (sem[2]);   // sem[2]=1
   signal (mutex);
   wait (sem[2]);   // Sem. is 1, turns 0
eat ();
put_forks (2);
   wait (mutex);
   state[2] = THINKING;
   test (1);  // possibly wake up others
   test (3);
   signal (mutex);
loop...
```

```
// i=3, sem[3]=0

think();
take_forks (3);
   wait (mutex);
   state[3] = HUNGRY;
   test (3);
      state[3] == HUNGRY?  yes
      state[2] != EATING?  NO !!
      state[4] != EATING?  yes
      -> do nothing (no signal() call)
   signal (mutex);
   wait (sem[3]);   // is 0, block!

   [blocks while philosopher 2 eats]

eat ();
put_forks (3);
   wait (mutex);
   state[3] = THINKING;
   test (2);  // possibly wake up others
   test (4);
   signal (mutex);
loop...
```


Monitors (1)

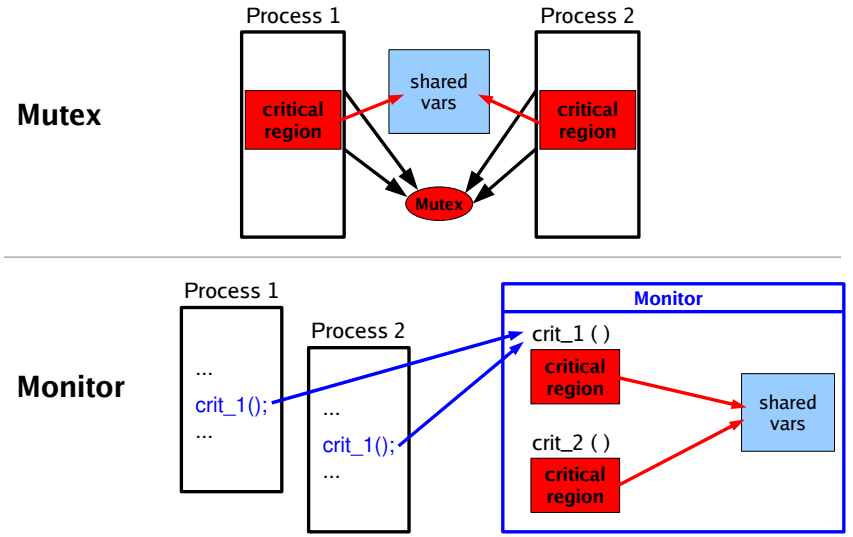
Motivation

- semaphores and mutexes force the programmer to call wait() and signal() before or after each critical region, respectively
- if this is forgotten just one time, synchronization will break
- **Monitor** encapsulates the critical regions

Monitors (2)

- **Monitor:** collection of procedures/functions (methods), variables, special condition variables and data structures:
  - processes can call methods of the monitor, but cannot otherwise access its internal data structures
  - at each point in time only one single process can be active in the monitor (i.e.: exceute a monitor method)
- monitor is released by exiting the monitor method

Monitors (3)



Monitors (4)

simple example: accessing a disk, using a mutex

```
mutex disk_access = 1;
```

```
wait (disk_access);
// read data from disk
signal (disk_access);

wait (disk_access);
// write data to disk
signal (disk_access);
```

same example, now with monitor

```
monitor disk {
  entry read (diskaddr, memaddr) {
    // read data from disk
  };
  entry write (diskaddr, memaddr) {
    // write data to disk
  };
  init () {
    // initialize device
  };
};
```

```
disk.read (da, ma);

disk.write (da, ma);
```


Monitors (5)

- monitor construct is part of a programming language
- compiler (and not the programmer) is responsable for guaranteeing mutual exclusion
- implementation (by the compiler) e.g. with semaphore/mutex:

```
- monitor disk          ➞  semaphore m_disk = 1;

- entry funktion () {  ➞  void funktion () {
    /* Code */              wait (m_disk);
  }                          /* Code */
                             signal (m_disk);
                           }

- disk.funktion();     ➞  funktion();
```


Monitors (6)

- Monitor concept reminds of
  - classes (object oriented programming)
  - modules (modular programming)
- encapsulation of procedures and variables (except through procedures explicitly defined *public*, there is no way to access the monitor)
- simple and concise method for protecting critical regions, but:
- **busy waiting** ➞ sleep/wakeup would be better

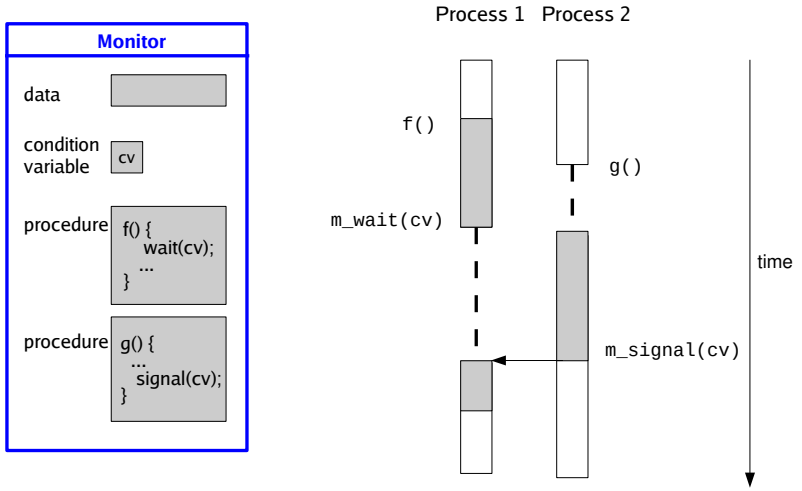
Monitors (7)

Condition Variables

for each condition variable there are wait and signal functions:

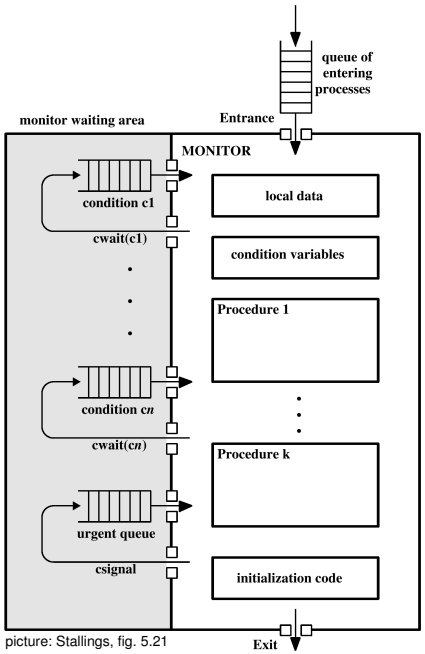
- *m_wait (var)*: block calling process (it releases the monitor)
- *m_signal (var)*: unblock blocked process (this will wake up a process which has left the monitor by calling *m_wait*); is called by a thread that is just about to leave the monitor

Monitors (8)



Monitors (9)

- blocked processes move to a queue belonging to the condition variable (on which the process blocked)
- status (cv) returns number of processes waiting on this cond. var.
- internal queues have precedence over processes trying to enter the monitor from outside



picture: Stallings, fig. 5.21

Monitors (10)

Producer Consumer Problem with monitor

```
monitor iostream {
  item buffer;
  int count;
  condition nonempty, nonfull;

  entry append(item x) {
    if (count == 1) m_wait(nonfull);
    put(buffer, x); // put is a local procedure
    count = 1;
    m_signal(nonempty);
  }

  entry remove(item x) {
    if (count == 0) m_wait(nonempty);
    get(buffer, x); // get is a local procedure
    count = 0;
    m_signal(nonfull);
  }

  init() {
    count = 0;  // initialization
  }
}
```

Source: Prof. Scheidig, Univ. Saarbrücken, http://hssun5.cs.uni-sb.de/lehrstuhl/WS0607/Vorlesung_Betriebssysteme/ - adapted to C syntax

Java and Monitors (1)

- Java uses monitors to synchronize threads
- key word **synchronized**
- a class that contains only synchronized methods is effectively a monitor
- no *named* condition variables
- queue:
  - m_wait: wait
  - m_signal: notify (wakes up a thread)
    notifyAll (wakes up all threads)

Java and Monitors (2)

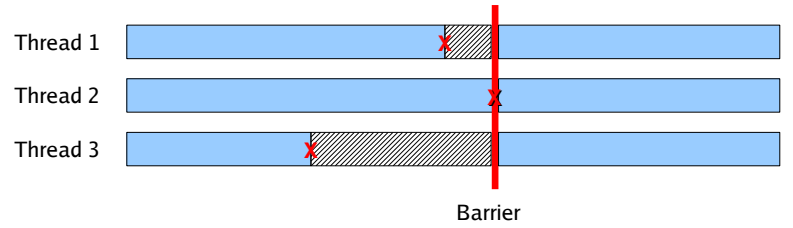
```
class BoundedBuffer extends MyObject {
   private int size = 0;
   private double[] buf = null;
   private int front = 0, rear = 0,
     count = 0;

   public BoundedBuffer(int size) {
      this.size = size;
      buf = new double[size];
   }

   public synchronized void
   deposit(double data) {
      while (count == size) wait();
      buf[rear] = data;
      rear = (rear+1) % size;
      count++;
      if (count == 1) notify();
   }

   public synchronized double fetch() {
      double result;
      while (count == 0) wait();
      result = buf[front];
      front = (front+1) % size;
      count--;
      if (count == size-1) notify();
      return result;
   }
}
```

source: http://www.mcs.drexel.edu/~shartley/ConcProgJava/Monitors/bbse.java

Barriers (1)

- Idea: break down complex computation into several phases
- before entering a new phase, all threads wait until they have all finished the old phase
- then e.g. distribution of intermediate results
- finally all threads continue their computations (independently) – until reaching the next barrier

Barriers (2)



- threads call *barrier()* and block
- only when all threads (all members of a group) have called *barrier()*, they can continue

Locking (1)

Locking extends the functionality of mutexes by offering miscellaneous lock modes and defining their compatibility:

- Concurrent Read: read access, other writers are allowed.
- Concurrent Write: write access, other writers are allowed.
- Protected Read: read access, other readers allowed, but no other writer (share lock)
- Protected Write: write access, other readers allowed, but no other writer (update lock)
- Exclusive: write access, all other accesses forbidden

Locking (2)

	concurrent read	concurrent write	protected read	protected write	exclusive
concurrent read	X	X	X	X	-
concurrent write	X	X	-	-	-
protected read	X	-	X	-	-
protected write	X	-	-	-	-
exclusive	-	-	-	-	-

Locking (3)

- thread requests lock with a specific mode.
  - if the lock mode agrees with already active locks of other threads, the lock will be granted.
  - if the lock is incompatible with some other process' lock already in place, the thread will block until the lock can be granted.
- locking mechanisms are implemented
  - by the operating systems
  - by user level applications (especially data bases)