3F6 - Software Engineering and Design

Handout 12

Concurrent Systems I

With Markup

Edward Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching.html`.

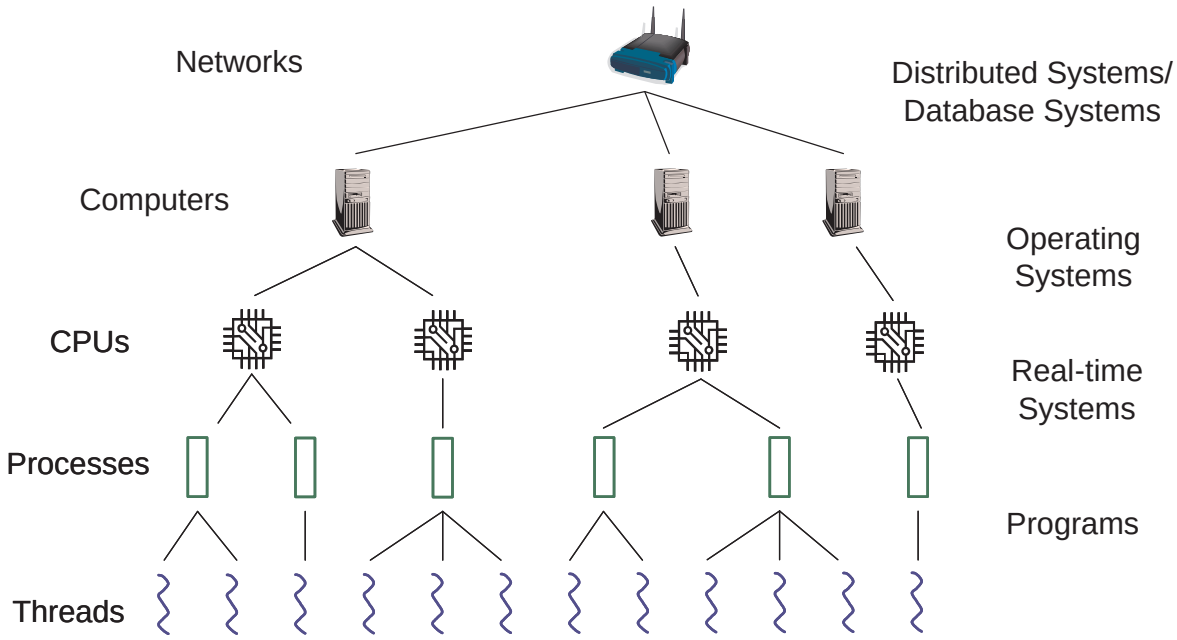
Concurrent Systems

A concurrent system consists of several elements which operate at the same time and communicate with each other.

Class of system	Examples
Web-based system	• Google ... • Department webserver
Transaction system	• HSBC internet banking system • BA flight reservation system • EBay
Distributed system	• Folding@Home (5×10^{15} FLOPS) • 'Jaguar' supercomputer (1.6×10^{15} FLOPS)
Operating System	• Linux or Windows • Mobile phone
Real time system	• Process control system • Engine management system
Embedded system	• Dishwasher • DVD Player
Applications	• Web browser • Microsoft Office

Levels of Concurrency

Concurrency may be viewed at the logical level and the physical level. There is a rough correspondence between the two but it is not exact.



Communication is via

- shared memory
- local databusses
- external network connections

Why is Concurrency an issue?

Concurrency is essential for:

- high performance computing by using more than one CPU to process data. The current fastest supercomputer has 224,256 cores.
- interaction with the real world since important events happen concurrently.

Concurrent systems must communicate with each other, and share resources. However, uncontrolled concurrent access to writable resources leads to errors:

- simultaneous access to the same memory location leads to errors (race conditions):

```
int i;
// thread 1          // thread 2;
++i                  ++i


//In assembler

LDAA i               LDAA i
INCA                 INCA
STAA i               STAA i
```

- Spirit rover suffered from a race condition

  `http://www.aiaa.org/spaceops2006/presentations/56207.ppt`

- simultaneous access to the same resources leads to unpredictable results. These concurrency problems were also highlighted in the lecture on Transaction Processing.

```
#include <iostream>
int main()
using namespace std;
{
      #pragma omp parallel for
      for(int i=0; i < 10; i++)
            cout << "Hello " << i << endl;

}
```

compile with: `g++ -o prog prog.cc -fopenmp`

- Resources can be locked:

// thread 1	// thread 2;
lock("printer");	lock("scanner");
lock("scanner");	lock("printer");

- uncontrolled locking can deadlock

- uncontrolled locking can cause much more subtle problems such as priority inversion.

  Shared resource: R
  Task 1:(Low priority)
  Lock R
  Do processing
  Release R

  Task 2: (High priority)
  Lock R (wait for 1 to finish)
  Do processing
  Release R

  Task 3: (Medium priority)
  Executes in preference to low priority tasks (1).
  Task 2 has to wait for task 3

- This happened on the Mars Pathfinder mission. The meteorological data gathering task locked a shared memory area used for all communication. This was suspended by a long running medium priority communications task. This prevented much higher priority tasks from running because they needed access to the memory area.

  `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html`

  `http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html`

- Using locks covered in lecture 13 and 14.

Implementation of Concurrency

1. At the cpu level and above, concurrency arises naturally since there are multiple physical processors. This is *real* concurrency.
2. Below the cpu level, a single cpu is shared amongst many software processes. This is *simulated* concurrency.

Why do we simulate concurrency if it creates problems?

Because overall it greatly simplifies many real world applications

- real time systems which have to respond to external asynchronous events
- desktop window-based applications which have to do several tasks at once
- server applications have to respond to several simultaneous requests

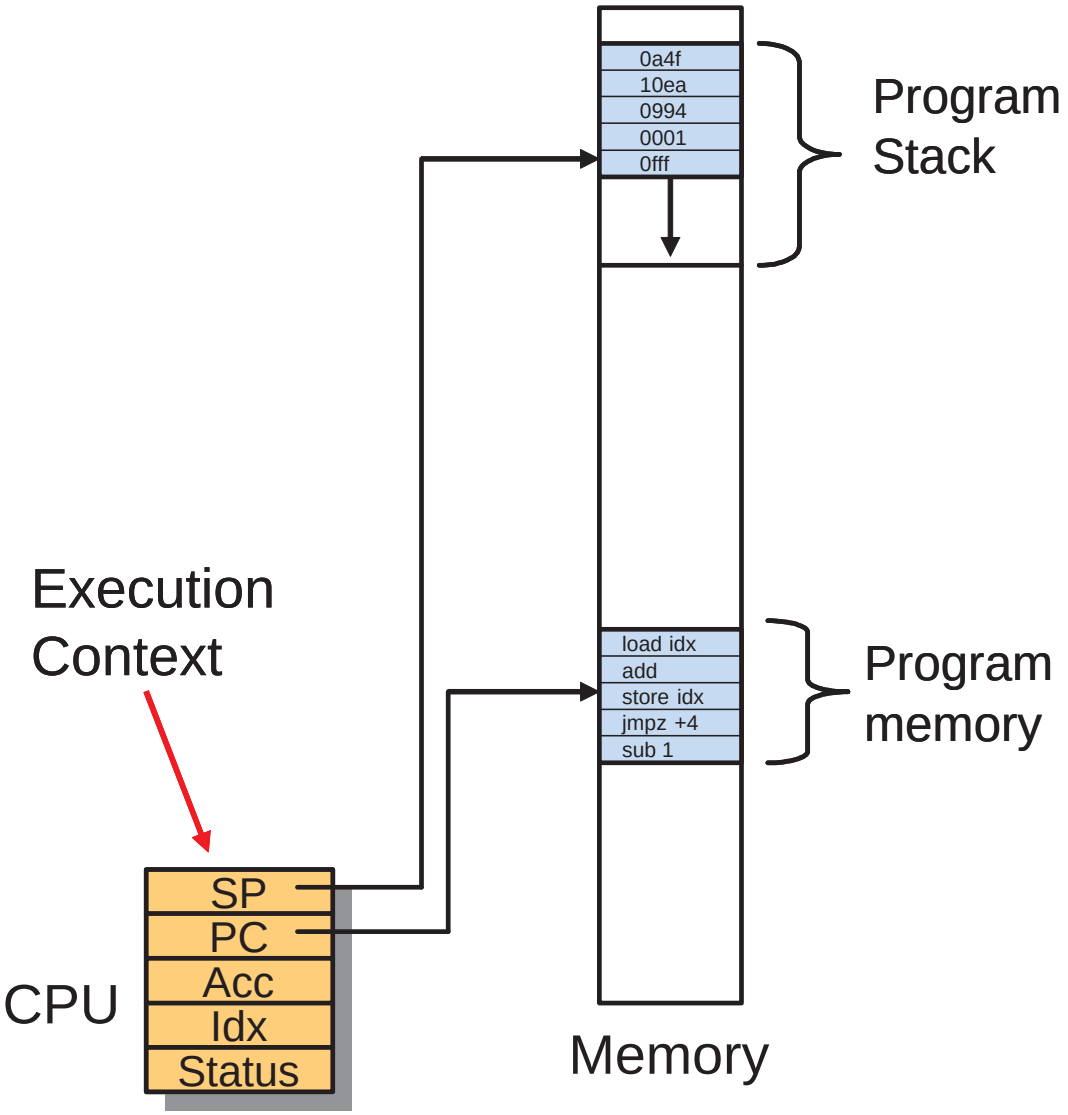
This lectures will focus on mainly on simulated concurrency and how it is implemented on a single processor. The next lecture will focus on the programming constructs and methods that allow concurrency to be used safely and effectively.

Note that the programmer should always assume that the concurrency is real! eg Core Duo

Processes and Execution Context

When a program is executing, the program state is determined by memory, the cpu state and the contents of its registers.

- **Program Stack** Part of the memory is allocated for procedure variables, parameters and return values. This memory is called the *Program Stack* since it grows and shrinks as procedures call and return.
- **Stack Pointer (SP)**. The SP is stored in the CPU and it points to the current top of the program stack. When a procedure is called, the SP is decremented by an amount equal to the total local memory requirement of that procedure. When the procedure exits, the SP is incremented by the same amount.
- **Program Memory** The program instructions are stored in the *Program Memory* which is often Read-only memory (ROM).
- **Program Counter (PC)** The PC points to the next instruction to execute in program memory. When each instruction completes, the PC is normally incremented automatically by the size of that instruction. Exceptions are jumps and procedure calls, in which case the PC is loaded with the address of the instruction to jump to or call.


0a4f
10ea
0994
0001
0fff
Program
Stack
Execution
Context
load idx
add
store idx
jmpz +4
sub 1
Program
memory
SP
PC
Acc
Idx
Status
CPU
Memory

Interrupts

When the voltage on a particular pin changes, the CPU stops whatever it is doing and jumps to a fixed location in memory.
Alter the PC

The memory location jumped to contains the *interrupt service routine*. This code will inevitably alter the state of the CPU. When the interrupt has been serviced the program should continue whatever it was doing when the interrupt happened.

First save the old PC, Acc, Idx, Status, etc ...

The interrupt then needs to be serviced. Interrupts are generated by external hardware such as:

- Keyboard, mouse, network card, timer, data capture

so the CPU needs to respond to the particular device causing an interrupt.

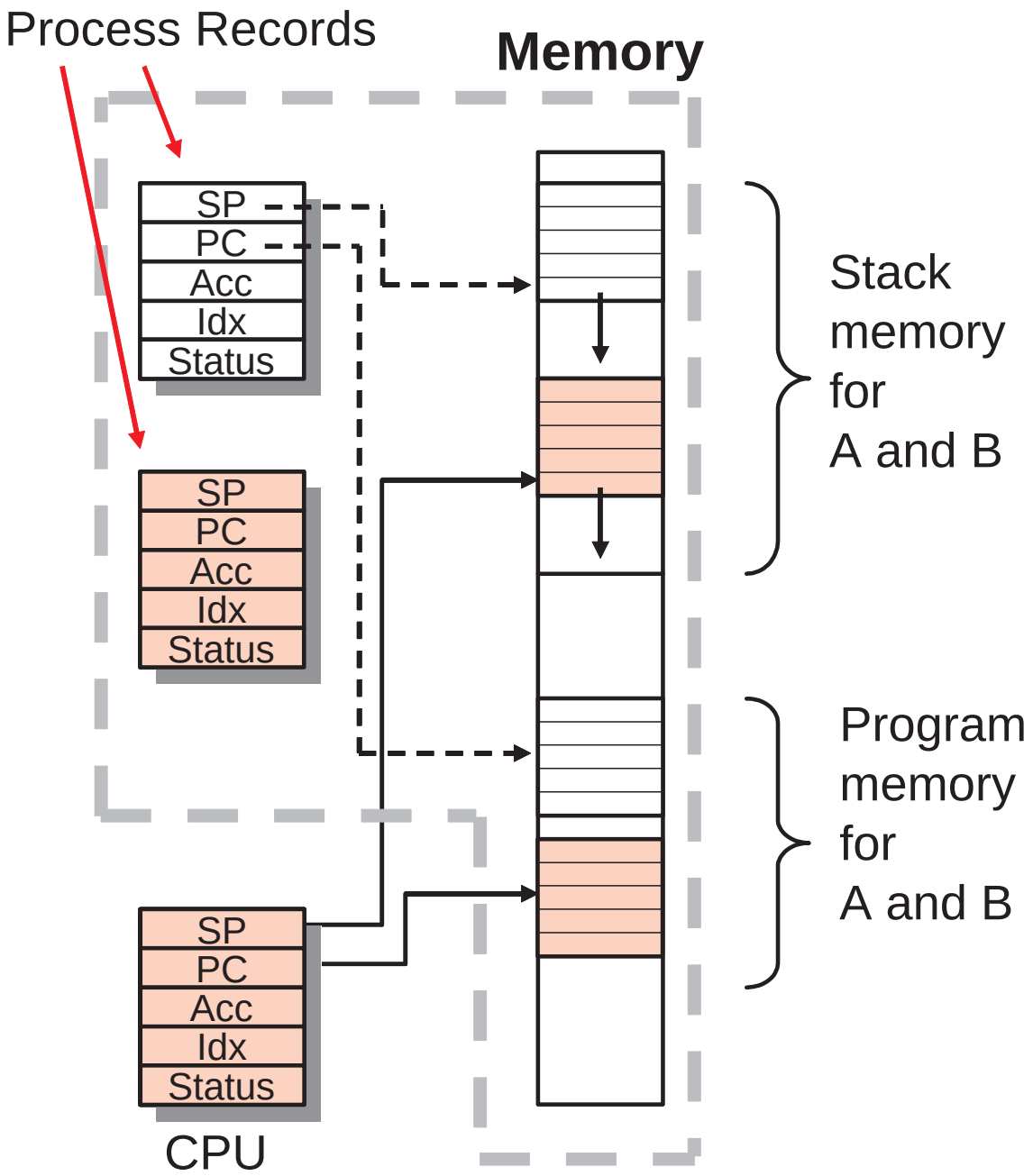
When the interrupt has been serviced, the processor state then has to be restored.

Restore Acc, Idx, Status, ...

Then last act of the interrupt service routine is to jump back in to the middle of whatever was happening before.

Restore the PC

Context Switching

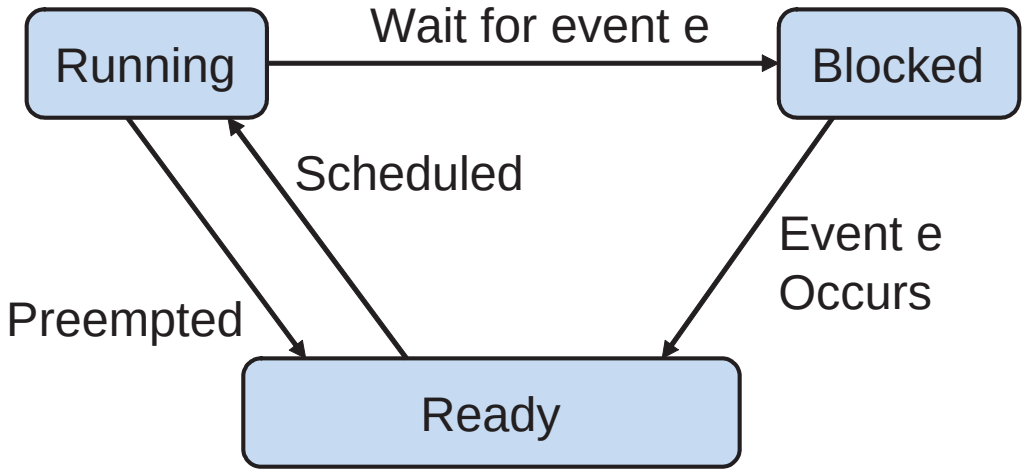


Context Switch: (a) save current cpu reg; (b) load new cpu reg

Process States

Each process is represented by a Process Record.

A process can be in one of three possible states:



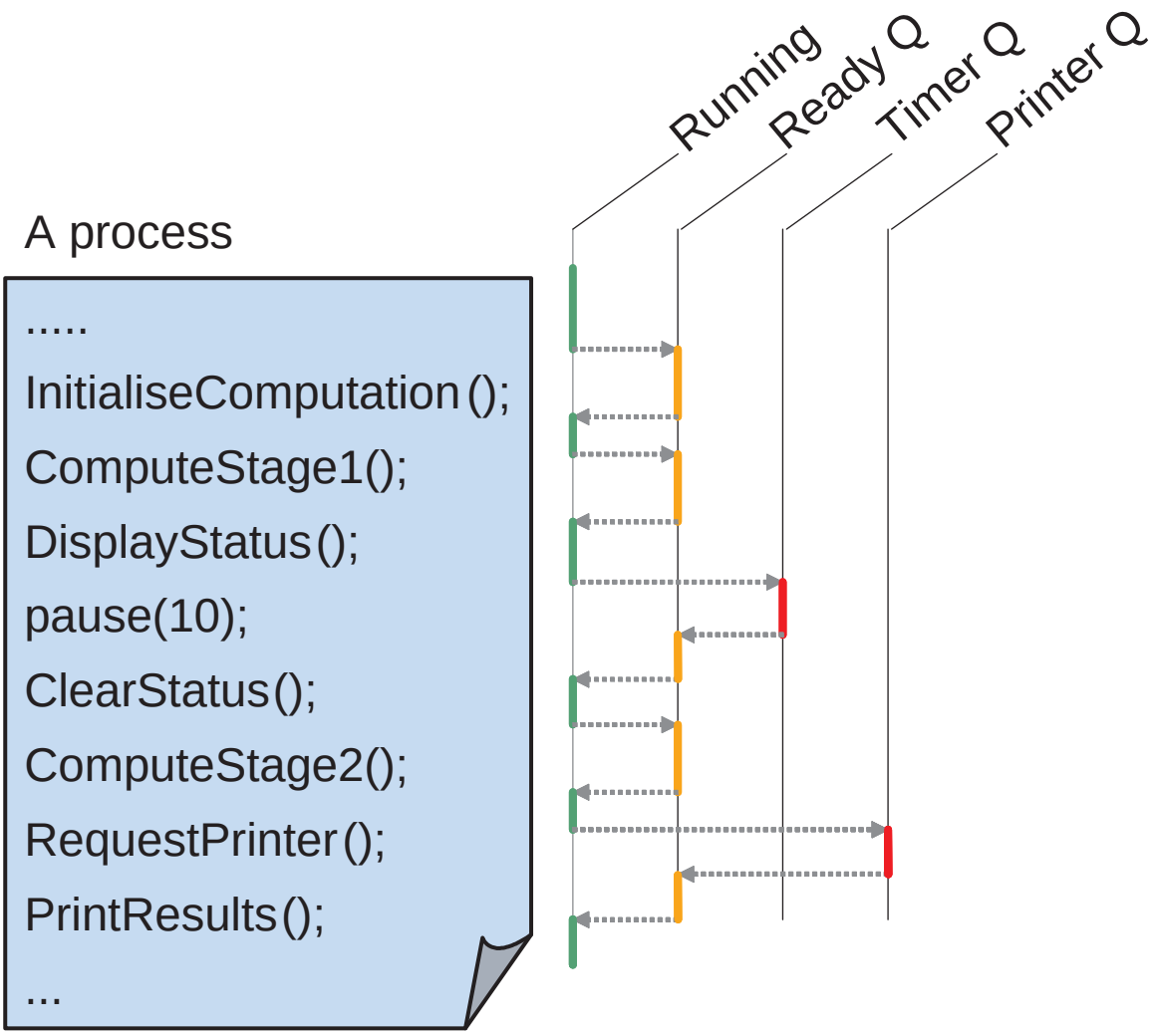
The Operating System (OS) maintains a number of process record queues

- ready queue (every process except running process is
- one queue for each possible event in one of these queues)

The OS schedules processes by moving them

1. from the ready queue to the physical cpu;
2. from the physical cpu to the ready queue;
3. from the physical cpu to an event queue.

Process Scheduling Example



This example illustrates pre-emptive scheduling. Embedded real time systems often use cooperative non-preemptive scheduling

Yield used instead of preemption

Process Scheduling Policy

Process scheduling policy is determined by the Operating System.

The scheduling strategy tries to fulfil the following criteria:

- fairness: every process gets a reasonable slice of cpu time
- efficiency: keep processor busy as much as possible
- response time: avoid holding a process in ready queue for too long
- control: allow programmer reasonable control

Programmer control is usually provided by allowing processes to be assigned priorities.

Note that priorities should never be used to solve "race conditions".

Processes and Threads

Process may be computationally expensive since

- Every process has its own private memory space and when a process is switched there is a time cost.
- Communication between processes must use the OS and/or Middleware (E.g. shared files, sockets, remote procedure calls). This can be cpu intensive if there is a lot of data to share.

Multiprogramming at the process-level is usually done using OS primitives such as sockets, messages or shared memory (and locks) or high level middle-ware systems systems such as CORBA, or Microsoft .Net.

A process can be divided into threads:

- Threads share the same memory space (each one has its own stack)
- Context switching is cheap since only the PC and registers are switched
- Communication is efficient via shared memory

Threads can be thought of as *lightweight processes* and often multi-programming on modern OS's involves using threads.

Multi-threaded Programming

Every operating system has its own set of primitives for multi-threaded programming. The names and details will vary but all provide similar core primitives.

Threads are defined like functions, but instead of calling them a thread is *spawned* using `create` and then later merged using `join`.

```
// define a thread as a function
void ChildThread(int i) {
  // define thread operations
}
```

Stores execution context →

```
// -------  main thread --------
Thread t = create(ChildThread,0,normal);
// ChildThread is now executing in parallel
// with main thread at normal priority.
...
...
```

Deletes execution context →

```
// wait for ChildThread t to terminate
join(t);
// now there is just one thread again
....
```

Other commonly provided thread management functions include:

```
kill(t);        // kill thread t
pause(n);       // pause calling thread for n msecs
exit();         // calling thread terminates
self();         // a reference to the caller
```

Example - Watchdog timer

```
const int N = 10;
Thread wd, t[N];
bool ok[N];

// Watchdog thread - checks workers are alive
void Watchdog(int i) {
    while(1){
       pause(1000);
       for (int i=0; i<N; i++) {
         if (!ok[i])
           kill(t[i]), RaiseAlarm(i);
         ok[i]=false;
       }
    }
}

// Worker threads - do the hard work
void Worker(int i) {
  DoCompute1();  ok[i] = true;
  DoCompute2();  ok[i] = true;
  ...
}

// main program
void main() {
   for (int i=0; i<N; i++) {
      t[i] = create(Worker,i,normal);
      ok[i] = false;
   }
   wd = create(Watchdog,0,high);
   // wait here for all workers to stop
   for (int i=0; i<N; i++) join(t[i]);
   kill(wd);
}
```