# Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

# Topic

Karoly.Bosa@jku.at

- Multicore Software Development: Fact and Fiction
- An Overview The Challenges of Concurrency
  - Software Decomposition
  - Non-determinism
  - Task to Task Communication
  - Concurrent Acces to Data or Resources
    - Race Condition
    - Deadlock
  - Synchronization Relationship
  - How Many Concurrent Activities Are Enough?
  - Debugging and Testing
  - Processor Architecture Challenges
- Multicore Programming: Easy or Difficult?

# Sequential vs. Concurrent Programming

Karoly.Bosa@jku.at

- Sequential programming techniques are important and will always have their place.
- However, multicore computer configurations are now widely available (with relatively low cost).
- The trend is that multicore computers will in most cases replace single processor configurations in business, academia, and government.
- Software architectures that include **a mix(!) of sequential programming, multiprocessing, and multithreading** will become common place.
- This opens up some quite different approaches to program decomposition and software organization.

# Multicore Software Development: Fact and Fiction I.

Karoly.Bosa@jku.at

- *The major challenge is refactoring existing software to achieve concurrency*. **NOT (ENTIRELY) TRUE!**
- Most (embedded) software system are already quite heavily multithreaded (simplify the management of the independent functions ):
  - On a unicore system, threads are logically concurrent.
  - On a multicore processor, these threads are naturally and truly concurrent, usually with **no change in the software required** (assuming an SMP) .
- Of course, not all systems make optimal use of all the hardware cores.
- Designers may indeed want to increase concurrency by refactoring the code.

# Multicore Software Development: Fact and Fiction II.

Karoly.Bosa@jku.at

- When refactoring software, maximize threads while minimizing processes. **NOT TRUE.**

- When deciding whether to map a new component to a thread (sharing memory space with other threads) or a process, consider:

  - Processes are memory protected

  - The cost (in terms of memory use and context switching time) of a process may be a bit higher

- Regardless of whether threads or processes are used, an SMP-capable operating system will automatically schedule the components onto the available cores.

# Multicore Software Development: Fact and Fiction III.

Karoly.Bosa@jku.at

- The industry is suffering from a lack of multicore standardization. **(MORE OR LESS)TRUE!**

- Multicore software needs the boost of pervasive standards.

- Only few parts of the problem area is covered by standards:

    - **Multithreading:** POSIX is a collection of open standard APIs specified by the IEEE for operating system services.

    - **Interprocess Communication (IPC):** MPI, POSIX, TIPC, LINX

- Beyond these standards are missing.

# Multicore Software Development: Fact and Fiction IV.

Karoly.Bosa@jku.at

- Multicore debugging tools are lagging. **NOT TRUE!**
- Although there are certainly a number of IDEs that have failed to adapt to the multicore evolution
- But leading IDEs have been focusing multicore support for a long time.
- ***On-Chip Debugging.*** Tightly-coupled multicore processors often provide a single on-chip debug port (e.g. JTAG).
- ***Multicore Event Analyzers.*** Many operating system vendors provide an event analysis tool.
- **Parallel (Multicore) Software Tools:** Probe, Multi IDE, TotalView, etc.

# Software Developer's Points of View

Karoly.Bosa@jku.at

The primary questions are:

- How do you know when your software application needs multicore programming?
- How do you know where to put the multicore programming in your piece of software?
- How to organize the interaction among the concurrent activities?

That's it. But…

# Challenges of Concurrency

Karoly.Bosa@jku.at

- Developers are faced with following challenges in the *Software Development Lifecycle* of parallel programs:

1. Software decomposition of tasks that need to execute simultaneously
2. **Non-deterministic** executions of concurrent activities
3. Communication between two or more concurrent tasks
4. Concurrently accessing or updating data
5. Identifying the relationships between concurrent tasks
6. Controlling resource contention when there is a many-to-one ratio between tasks and resource
7. Determining an optimum or acceptable number of parallel activites
8. Finding test (real/simulated) environment for parallel programs
9. Recreating a software exception or error in debugging.

# 1. Software Decomposition

Karoly.Bosa@jku.at

- Before decomposition you cannot decide about:
  - Whether to use concurrent activities?
  - How many parallel activities to use?
  - Whether use threads or processes, etc.?
- Degrade the complexity of the problem into its fundamental parts. But what are the fundamental parts of a problem?
- The answer depends on what model you use to represent the problem. Two major class can be mentioned:
  - Procedural models
  - Declarative models (e.g.: OO)
- According to the experiences procedural models are not able to scale over a certain limit (100 or 1000 parallel activities).
- Declarative should be used in the decomposition process(!).

# 2. Non-determinism

Karoly.Bosa@jku.at

- What is concurrency?
- Two or more tasks executing over the same time interval are said to *execute concurrently*:
  - On two or more CPU this can mean concurrent activities at the same time.
  - On CPU this always mean ***interleaving*** of executing activities.
- In both cases the execution order of instructions of different task is not predefined.
- Most of the challenges are derived from the non-deterministic behavior.

# Task to Task Communication

Karoly.Bosa@jku.at

- In some cases, the operating system keeps the resources of concurrent task separate (e.g.: processes).
- Consequently these tasks use separate memory address spaces, too.
- In this case special Inter Process Communication (IPC) mechanisms provided by the OS are required (e.g. in POSIX):
  - Command-line arguments
  - Environment variables
  - Files with locking facilities
  - Pipes
  - Shared memory
  - Message queues
  - Sockets

# Concurrent Access to Data or Resources

Karoly.Bosa@jku.at

- Synchronization problems imply the competition for some resources by two or more tasks at the same time
- Resources can be:
  - **Software resources:** files, records within files, fields within records, shared memory, program variables, pipes, sockets, and functions.
  - **Hardware resources:** interrupts, physical ports, and peripherals such as printers, modems, displays, storage, and multimedia devices.
- It can result data loss, incorrect program results, system failure, and in some cases, device damage.
- Some common types of synchronization problems are:
  - Race Condition
  - Deadlock

# Race Condition

Karoly.Bosa@jku.at

- If two or more tasks attempt to change(!) a shared piece of data at the same time
- The final value of the data depends simply on which tasks get there first (non-determinism)..
- For instance, let a=5 and b=12 are shared variables:

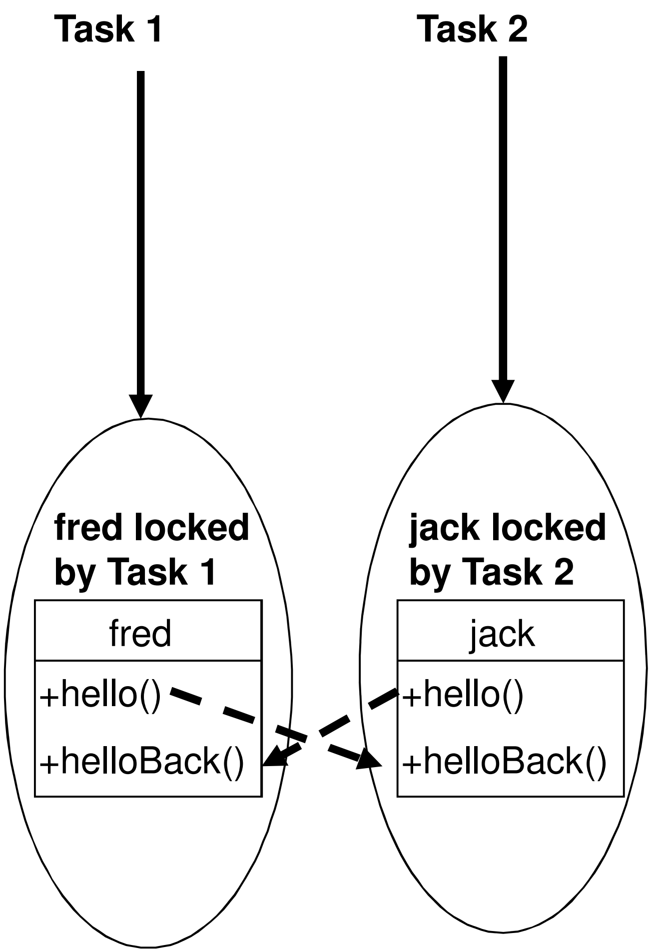
| | |
|---|---|
| a = a+b;<br><br>//Let b equal the previous value of a<br>b = a-b;<br><br>//Let a equal the previous value of b<br>a = a-b; | a--;<br>b--; |

# Deadlock

Karoly.Bosa@jku.at

- A waiting-type pitfall: It describes a situation where two or more concurrent tasks are **blocked forever**, waiting for each other.

- A typical example is:
    - When two concurrent activities lock (have exclusive access to) some distinct resources.
    - But each of them need to access to the resources belonging to the other task before it can released its own resources.
    - Both of them will wait for the release of the corresponding resources (forever).

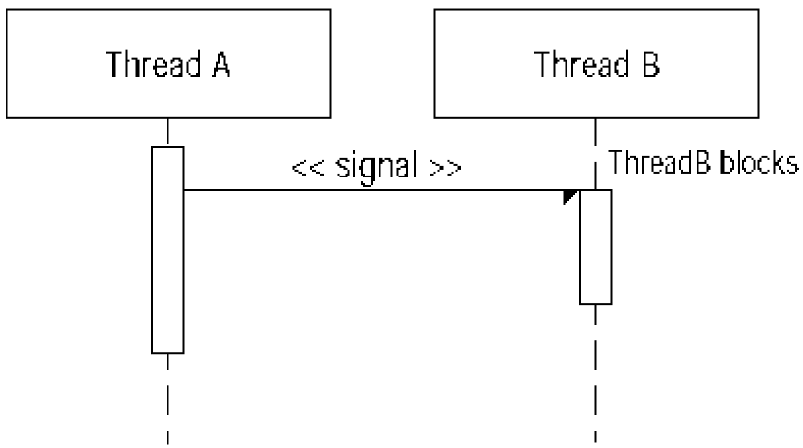


# Identifying Relationships

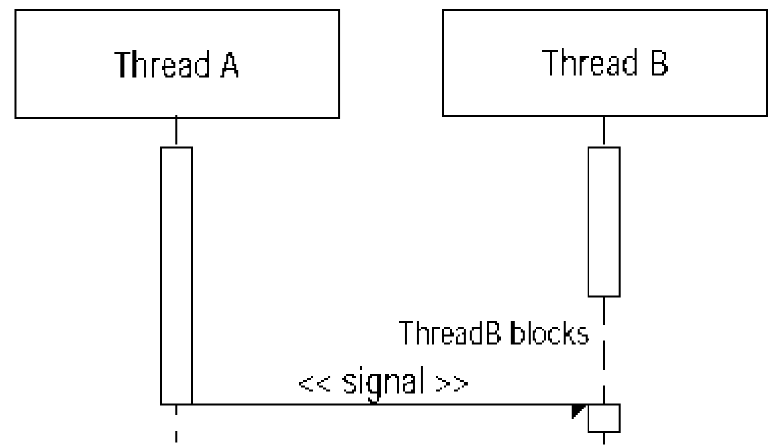
Karoly.Bosa@jku.at

- ***Basic Synchronization Relationships:***

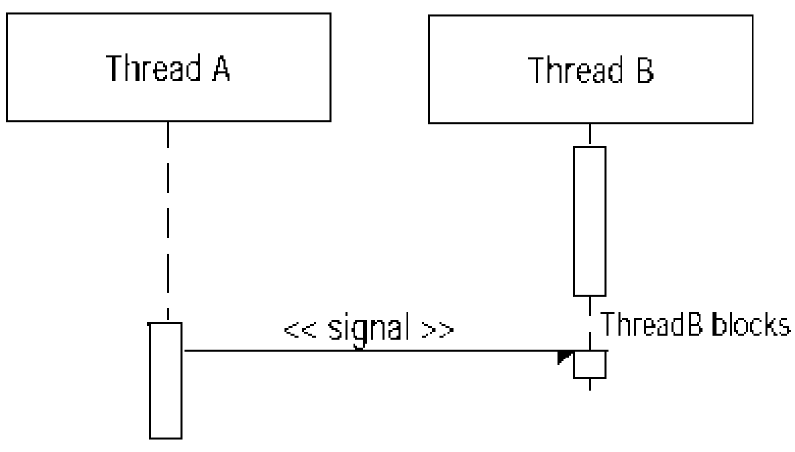
START-TO-START



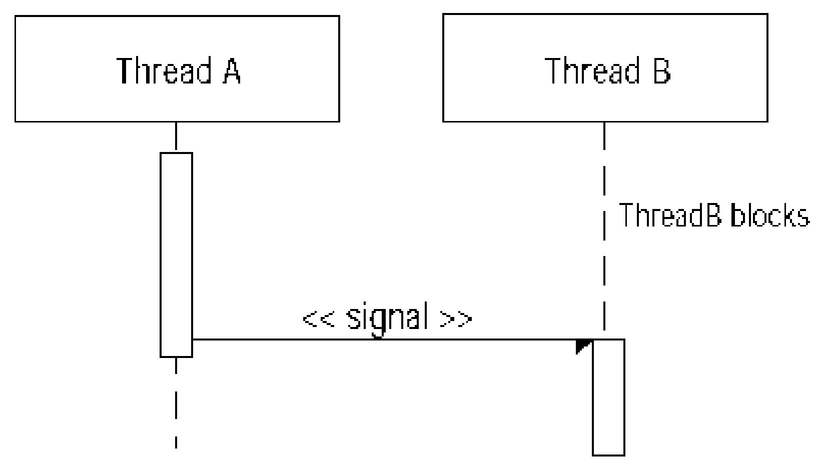

FINISH-TO-FINISH




FINISH-TO-START




START-TO-FINISH



# How Many Concurrent Activities Are Enough?

Karoly.Bosa@jku.at

- The old adage “ you can never have enough processors ” is simply **not true**.
- Extra overheads which may outweigh the speed improvement and other advantages gained from parallelization:
  - Managing multiple concurrent activities (creation, start, clean up, etc.)
  - Communication or synchronization between concurrent activities.
- The question is whether is there an optimal number of processors (concurrent tasks) for any given parallel program? The limitation can caused:
  - Either by the managing of software component
  - Or by the available hardware resources

# Debugging and Testing I.

Karoly.Bosa@jku.at

- Sequential programs:
  - Tracing of the logic of a program in a **step by step manner**.
  - Starting with the same input  (and with the same system state), then the outcome or the flow of the logic is **predictable.**
- It is difficult to reproduce the exact context of concurrent tasks because of :
  - operating system scheduling policies,
  - dynamic workloads on the computer,
  - processor time slices,
  - process and thread priorities,
  - communication latency and
  - the random chance involved in parallel contexts.

# Debugging and Testing II.

Karoly.Bosa@jku.at

- Non-determinism in cross platform development:
  - Different treatment of processes and threads in different OS and hardware.
  - For instance, thread priorities in different systems:
    - high, medium, low
    - User-defined priority levels
    - Mission critical priorities
    - Real-time priorities
    - Normal priorities
    - Background priorities, etc.
  - Different types of schedulers in different OS
  - Different implementations of IPC mechanisms in different OS
  - Different implementations of kernel threads versus user threads.

# Processor Architecture Challenges

Karoly.Bosa@jku.at

- Different architectures translate to difference set of compiler switches and directives
- In some cases (e.g.: CBE), multiple types of compilers are needed to generate a single executable program.
- Different linker specific features (e.g.: CBE)
- If you take the advantage of particular architecture can make the software **non-portable(!).**

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

- According to some commentators that *“efficiently and correctly porting existing code onto platforms with four or more cores is beyond the capabilities of many engineers.”*

- Others simply state that *“this is a solved problem and that mature SMP operating systems and threading libraries already exist and are well understood.”*

- So who is right? The answer is *“it depends.”*

- The OS can make life easier:
  - With full SMP OS support such as Linux.
  - In case of refactoring, by supplying multithread support and APIs, particularly POSIX Threads.



By Barry O’Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

- A number of different applications fall into the "easy" category.
- **Example I: In communication and networking**
- A single program needs to deal with a large number of clients.
- A new thread is created for each incoming connection, and that thread exists for the duration of the connection.
- The operating system deals with assigning these threads to a processor and scheduling their execution.

```
...
do {
    ...
    clientSocket = accept(listenSocket,
                          (struct sockaddr
*)&clientAddress,
                          &addressSize);
    if (clientSocket == -1) {
        perror("Unable to accept connection");
        exit(1);
    } else {
        if (pthread_create(&threadHandle,
NULL,
                           server, &clientSocket)) {
            fprintf(stderr, "Unable to spawn a
new thread");
        }
    }
}
...
```



By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

- **Example II: Data Parallel approach**

- Loop unrolling and auto parallelization: highly optimized code into finer grain threads (by the compiler).

```
...
for (i = 0; i < SET_SIZE; i++)
  aData[i] = process(aData[i]);
...
```

- Each iteration of the loop is independent of the next

- On a four core system, four iterations of the loop can be executed in parallel with the expectation of achieving a better than **3x acceleration**.



By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

- **Example III:** The main loop now steps through the data in strides of PROCESSOR_COUNT(=4). In each iteration, we create a set of threads and then wait for them to complete.

```
void wrapped_process (data_t* pData)
{
  *pData = process(*pData);
  pthread_exit(NULL);
}
...
for (int i=0; i<SET_SIZE; i+=PROCESSOR_COUNT)
{
   int num_threads = (SET_SIZE – i) < PROCESSOR_COUNT ?
                              (SET_SIZE – i) : PROCESSOR_COUNT;
  for (int n=0; n<num_threads; n++)
    pthread_create(&thread[n], NULL, (void *) wrapped_process,
                              &aData[i+n]);
  for (int n=0; n<num_threads; n++)
    pthread_join(thread[n], NULL);
}
...
```

- The performance is barely better than the sequential implementation.

- **Serious load balancing issues:**
  - the program does not treat all of the elements in the data set equally.
  - there is only four threads per iteration.
- More frequent thread creation than is necessary.



By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

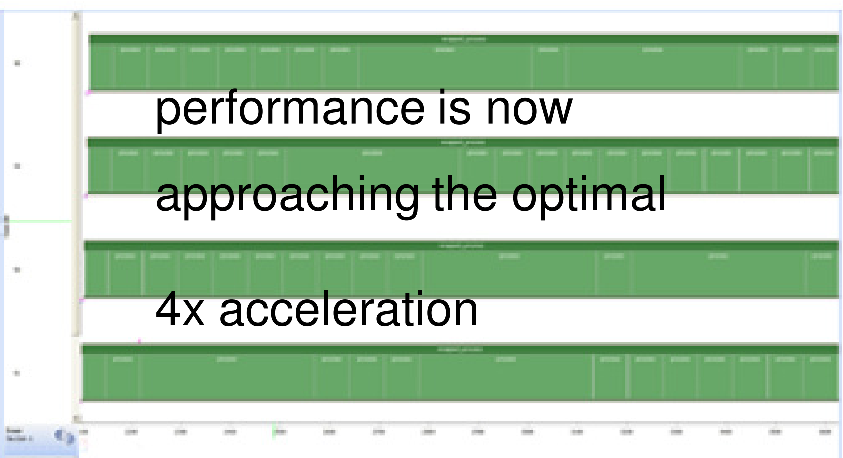
- **Example III 2nd version:**
  - the loop becoming distributed across the threads,
  - each thread responsible for updating the index of data set,
  - we now have this shared index variable, gDataIndex,
  - it is necessary to **to prevent a data race**.

```
#define THREAD_POOL_SIZE 25
int fetchAndIncrementIndex(void)
{
  int aLocalIndex;

  pthread_mutex_lock(&gDataIndex_mutex);
  aLocalIndex = gDataIndex++;
  pthread_mutex_unlock(&gDataIndex_mutex);

  return aLocalIndex;
}
void wrapped_process (data_t* pDataSet)
{
  int aIndex;
  while ((aIndex = fetchAndIncrementIndex()) < SET_SIZE)
    pDataSet[aIndex] = process(pDataSet[aIndex]);
  pthread_exit(NULL);
}
...
  gDataIndex = 0;
  for (int n=0; n<THREAD_POOL_SIZE; n++)
    pthread_create(&aThreadSet[n], NULL,
                   (void *) wrapped_process, (void *) gDataSet);

  for (int n=0; n<THREAD_POOL_SIZE; n++)
    pthread_join(aThreadSet[n], NULL);
...
```



performance is now approaching the optimal 4x acceleration

By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

- **Thread Safety:** A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.
- If we do not apply mutual exclusion the test of the program can give correct result (for a while).
- Small changes in the execution time of one thread or the other could change the access order to the variable suddenly triggering a previously unnoticed data race.

This following code fragment is not thread-safe:

```
…
 if (gDataIndex < SOME_LIMIT)
   return doThis(pData);
 else
   return doThat(pData);
…
```



By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN

# Multicore Programming: Easy or Difficult?

Karoly.Bosa@jku.at

**Deadlock:** It can occur when two separate thread-safe resources are accessed together.

- **Example IV.:** The two functions are executed by different threads.
- If a thread executes both locks before the second thread executes the locks then everything will work correctly.
- When both threads reach their respective function at the same time that problems occur.
- The best approach is to enforce a programming standard on a project (everyone knows what order to lock and unlock ).

```
void function_in_thread_1(void)
{
  ...
  pthread_mutex_lock(&gLock1);
  pthread_mutex_lock(&gLock2);
  aSharedVariable1 += aSharedVariable2 * 4;
  pthread_mutex_unlock(&gLock2);
  pthread_mutex_unlock(&gLock1);
  ...
}
void function_in_thread_2(void)
{
  ...
  pthread_mutex_lock(&gLock2);
  pthread_mutex_lock(&gLock1);
  aLocalVariable = aSharedVariable1
                            - aSharedVariable2;
  pthread_mutex_unlock(&gLock1);
  pthread_mutex_unlock(&gLock2);
  ...
}
```



By Barry O'Rourke, CriticalBlue -- 3/25/2009 EDN