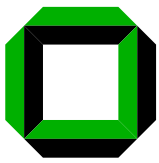

University of Karlsruhe
System Architecture Group
Gerd Liefländer

Teaching Assistant:
Philipp Kupferschmied

System Architecture 2008/09
Assignment 6

Question 6.1: Portable Barrier Synchronization

Consider the following code for barrier synchronization of two threads (from the lecture).

```
module synchronization
    export synchronize;
    import BLOCK, UNBLOCK;
    type sync = record
        S : signal = reset;
        W : thread = nil;
    end;

    procedure synchronize (var SY:sync)
    begin
        if (SY.S = reset) then begin
            (* first thread to arrive *)
            SY.S := set;          (* arm barrier *)
            SY.W := myself;       (* tell my partner who is waiting *)
            BLOCK(myself)         (* wait for partner *)
        end else begin
            SY.S := reset;        (* allow future reuse! *)
            UNBLOCK(SY.W)         (* reactivate partner *)
        end
    end
end
```

1. Assume `synchronize` is used for synchronization of two KLTs that run on an operating system with preemptive scheduling. Will the code shown above lead to correct results? If not, what needs to be done to fix it?
2. Does the solution place any constraints on the scheduling policy? Does it work both with time-slice–based scheduling and with static priorities?
3. Improve the code so that it reliably works for $N \geq 2$ threads.

Question 6.2: Terms and Definitions

1. What is the difference between a *critical section (CS)* and a *critical region (CR)*[1]?
2. Enumerate and explain the requirements for a valid synchronization solution.
3. What is the difference between a weak and a strong counting semaphore?

Question 6.3: Concurrent Modifications of Shared Memory

Consider the following Pascal program:

[1] This definition is KIT-specific

```
const n = 50;
var tally : integer;

procedure total;
var count : integer;
begin
  for count := 1 to n do tally := tally + 1
end;

begin (* main program *)
  tally := 0;
  parbegin
    (* two threads executing in parallel *)
    total; total
  parend;
  write(tally)
end.
```

1. Determine the lower and upper bounds of the final value of the shared variable `tally` as printed by the main program. Assume that threads can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

2. Suppose that an arbitrary number $t > 2$ of parallel threads performing the above procedure `total` is started within the `parbegin...parend` clauses of the above main program. What (if any) influence does the value of t have on the range of the final values of `tally`?

3. Now suppose `parbegin` created and started PULTs for each function called before the closing `parend`. Would this change make a difference to the output?

4. Finally consider a modified `total` routine:

   ```
   procedure total;
   var count : integer;
   begin
     for count := 1 to n do begin
         tally := tally + 1;
         yield
     end
   end;
   ```

   What will be printed now, if PULTs or KLTs are used?

Question 6.4: Protecting Shared Data Structures

Consider $n > 1$ threads concurrently accessing (a) a doubly linked list and (b) a binary tree. Each node in the data structures describes a customer record containing first and last names, home and work addresses, phone numbers, ... The contents of the customer records will never change. Nodes are added to the data structures, maintaining sort order by last name. Before a new node is added to the data structure, it is visible only to the thread that creates the customer record, but after the record is added, it becomes visible to all other threads, since they can follow the pointers in the tree or list.

1. Describe how the data structures are vulnerable to race conditions and how to avoid the race conditions.

2. Must a customer record have completely valid data before it is added to a data structure? Why or why not?

Question 6.5: Towards High-Performance Spinlocks

Assume you had a symmetric multi-processor system (SMP) consisting of 4 processors $P_1, \ldots, P_4$ with common main memory and dedicated L1 and L2 caches per processor.
Three cooperating threads T with a medium sized working set—each fitting into each L2 cache—run in parallel on processors $P_1, \ldots, P_3$. These threads synchronize with each other via spinlocks:

```
do
    reg := myThreadId;
    (* atomically exchange shared variable 'spinlock' and reg *)
    swap(&spinlock, reg);
until (reg = 0);
(* critical section *)
spinlock := 0;
```

Independently of those 3 threads, processor P_4 executes a grandmaster chess program with a large working set not fitting into the L2 cache.

1. Why does the chess program run substantially slower when the threads T execute concurrently?
2. How can you modify the software so that the chess program's performance is only hardly affected by the execution of T?

Question 6.6: Software-Based Mutual Exclusion

The following code fragment shows Peterson's solution for mutual exclusion of two threads:

```
(* Peterson's CS protocol for 2 threads Ti and Tj *)
forever do begin
  (* non-critical code *)
  (* entry code for thread Ti *)
  flag[i] := true;
  turn := j;
  while (flag[j] and (turn = j)) do begin (* wait *) end;
  (* critical section *)
  (* exit code for thread Ti *)
  flag[i] := false;
end
```

1. Is the variable `turn` really required, or would it be sufficient to only use the array `flag`?
2. What is the advantage of Peterson's algorithm over a simpler solution that uses only a single `turn` variable (as in "algorithm 1" in the lecture)?
3. Describe a scenario where Peterson's solution can lead to starvation.