General Preliminaries

- Welcome!
- We have some unallocated time in technical sessions later in the week
  - if there are things you would like to hear about, tell us now!
- If we are paying some or all of your expenses
  - get a form from me
  - keep receipts!

Exception and Error Handling in GAP

Steve Linton

Centre for Interdisciplinary Research in Computational Algebra

University of St Andrews

What Happens when Something Goes Wrong?

- When the GAP kernel or library encounters some condition it can't deal with “normally” it enters the break loop.
  - mathematical errors
  - soft out of memory
  - ^C
  - infinite recursion
- A great environment for debugging
  - not so good in other environments

Hooks and Fudges

- Existing GAP has quite a few hooks and controls for changing break loop behaviour:
  - -T command-line option
  - QUITTING global variable
  - OnBreak
  - OnBreakMessage
  - OnQuit
- all wrapped around the basic behaviour of break loop and quit or return

Why change?

- We want to be able to respond to errors in software
  - eg a GAP server should redirect (most) error messages to the client
- We want to introduce new kinds of interruption
  - external signals, CPU time limits, even softer memory limits
- Always good to move control from the kernel to the library
  - more transparent, more maintainers

Necessary Kernel Support

- A way of passing control out of the function where something went wrong, back to somewhere that can deal with it
  - CALL_WITH_CATCH(<func>, <args>)
  - JUMP_TO_CATCH(<payload>)
    - CALL_WITH_CATCH returns [true*[, <result>]*] if the function complete or [false, *<payload>*] if JUMP_TO_CATCH was called.
- A way to run an interactive shell (the break loop)
  - SHELL(<lots of arguments>)
    - returns fail (after quit, QUIT or EOF) or [ ] or [<returned>]

Putting it Together – Error in GAP

```
BIND_GLOBAL("ErrorInner",
        function( arg )
    local   context, mayReturnVoid,  mayReturnObj,  lateMessage,  earlyMessage,
            x,  prompt,  res, errorLVars, justQuit;
    context := arg[1];
    justQuit := arg[2];
    mayReturnVoid := arg[3]; mayReturnObj := arg[4];
    lateMessage := arg[5];   earlyMessage := arg{[6..Length(arg)]};
   # argument checks omitted.
    ErrorLevel := ErrorLevel+1;
    errorCount := errorCount+1;
    errorLVars := ErrorLVars;
    ErrorLVars := context;

    if QUITTING or not BreakOnError then
        PrintTo("*errout*","Error, ");
        for x in earlyMessage do
            PrintTo("*errout*",x);
        od;
        PrintTo("*errout*","\n");
        ErrorLevel := ErrorLevel-1;
        ErrorLVars := errorLVars;
        JUMP_TO_CATCH(0);
    fi;
```

Points to Note

- ErrorInner is called to handle a range of kernel level errors as well as the Error() function
  - hence it has lots of options to tell it whether return or return <obj> are allowed, what messages to print when, etc.
- Real code checks the arguments
  - ommitted here for brevity

Error Continued

the Actual brk loop

```
PrintTo("*errout*","Error, ");
for x in earlyMessage do
    PrintTo("*errout*",x);
od;
PrintTo("*errout*","\n");
if IsBound(OnBreak) and IsFunction(OnBreak) then
    OnBreak();
fi;
if IsString(lateMessage) then
    PrintTo("*errout*",lateMessage,"\n");
elif lateMessage then
    if IsBound(OnBreakMessage) and IsFunction(OnBreakMessage) then
        OnBreakMessage();
    fi; fi;
if not justQuit then
     if ErrorLevel > 1 then
         prompt := Concatenation("brk_",String(ErrorLevel),"> ");
      else
          prompt := "brk> ";
     fi;
    res := SHELL(context,mayReturnVoid,mayReturnObj,1,false,
       prompt,false,"*errin*","*errout*",false);
else
    res := fail;
fi;
```

Error in GAP part 3 -- Cleanup

```
    ErrorLevel := ErrorLevel-1;
    ErrorLVars := errorLVars;
    if res = fail then
        if IsBound(OnQuit) and IsFunction(OnQuit) then
            OnQuit();
        fi;
        SetUserHasQuit(1);
        JUMP_TO_CATCH(3);
    fi;
    if Length(res) > 0 then
        return res[1];
    else
        return;
    fi;
end);

BIND_GLOBAL("Error",
        function(arg)
    CallFuncList(ErrorInner,
            Concatenation([ParentLVars(GetCurrentLVars()), false, true, false,
   true],arg));
end);
```

Side benefit

- Having implemented SHELL it was easy to extend it so it could handle the main GAP read-eval-view loop as well
  - so now init.g actually includes lines to run the main loop
    - another big chunk of code out of the kernel
    - less repetition
- SHELL could be used for other interactions.

Customising Error Handling

- Previous customisations still work
- For more complete control, you can overwrite ErrorInner and do whatever you want
  - SHELL and JUMP_TO_CATCH are there if you want to fall back to traditional behaviour

Handling Other Events

- I plan soon to allow installation of GAP functions as handlers for
  - external signals
  - soft out-of-memory
  - some kind of timeout
- Defaults will maintain current behaviour
- JUMP_TO_CATCH will allow these handlers to interrupt program flow.

Questions and Future Directions

- QUESTION: would it be useful to provide other options for customising error handling? if so what?
- What events would it be useful to be able to trap?
- Longer term, the binary choice of returning fail or Error() when something can't be handled nicely in the library could be replaced by a more sophisticated Exception mechanism (cf Java or Python or clu).
  - any thoughts on design of such a setup?

Introduction to the GAP Kernel and Compiler

Steve Linton

Centre for Interdisciplinary Research in Computational Algebra

University of St Andrews

The GAP Kernel

- The 150K line C program that
  - provides the run-time environment and UI
  - interprets the GAP language
  - does arithmetic on basic types
  - speeds up some things
- One access to kernel functionality is kernel functions

```
gap> Print(KERNEL_INFO,"\n");
function ( )
    <<compiled code>>
end
```

- Simplest form of kernel programming is to add kernel functions

Levels of Kernel Programming

- Four levels of sophistication
  1. Add kernel functions manipulating existing data types
  2. Use the “data object” type to add new binary data structures in the kernel
  3. Add new primitive data types (eg Float)
  4. Add syntax to the language
- Try and give an idea of the first two today.

Adding a Kernel Function

- Hello World
  - a kernel function to print “Hello World!”
    - takes no arguments, returns nothing
- The C handler – add somewhere in string.c:

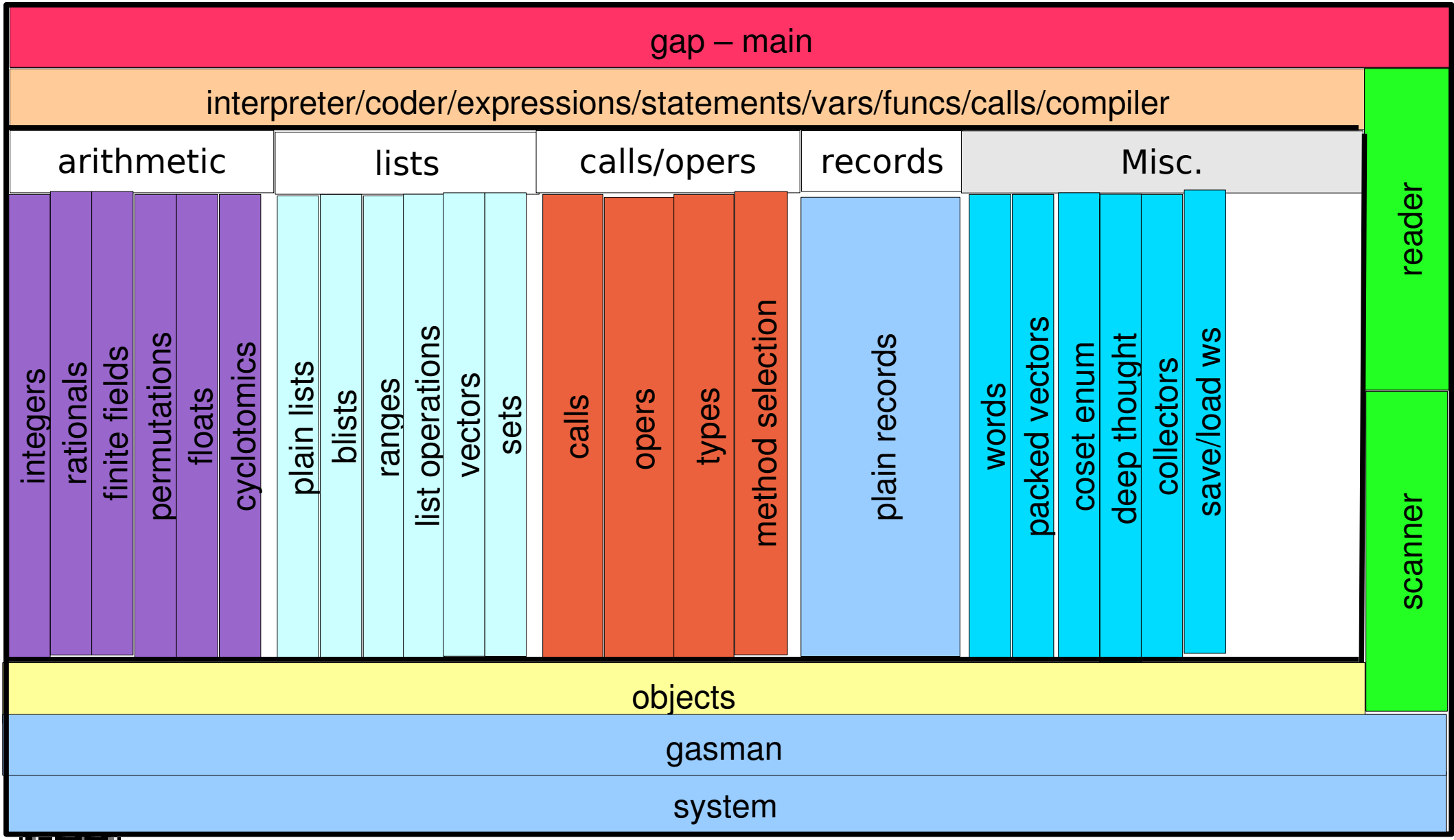
```
Obj FuncHELLO_WORLD( Obj self) {
  Pr("Hello World!\n",0L, 0L);
  return (Obj) 0;
}
```

  - The table entry (in GvarFuncs in string.c).

```
{ "HELLO_WORLD", 0, "",
    FuncHELLO_WORLD, "src/string.c:HELLO_WORLD" },
```

- That's it! Compile and HELLO_WORLD(); will work.

The Big Picture



Big Picture – Points to Note

- In the region within the thick lines kernel code sees the same world as GAP code
  - objects, with automatic memory management
  - no unevaluated expressions or fragments of code
- The only difference is that kernel code can see into the binary contents of the objects if it wants to
- Adding a kernel function is easy if it stays in that box
  - ie uses interfaces provided in white and yellow areas
  - these interfaces provide kernel equivalents to basic GAP functionality.

Another Function

- Return a list containing an object, its square and cube
  - x -> [x,x^2,x^3]

```
Obj FuncFoo(Obj self, Obj x) {
        Obj x2;
        Obj l = NEW_PLIST(T_PLIST_HOM+IMMUTABLE, 3);
        SET_ELM_PLIST(l,1,x);
        CHANGED_BAG(l);
        x2 = PROD(x,x);
        SET_ELM_PLIST(l,2,x2);
        CHANGED_BAG(l);
        SE2
```

Bags

- The GAP memory manager GASMAN provides an API dealing with “Bags”
  - areas of memory with types and sizes that have stable handles (of C type Bag) and can be resized
  - when the heap is full inaccessible bags are automatically reclaimed and live bags may be moved, but the handles don't change
    - handle is a pointer to a pointer to the actual data
  - Bag references from C local variables are found automatically, references from C statics and globals must be declared.

Objects

- The kernel view of every GAP Object is an object of C type Obj
  - most are Bags, represented by their handles
  - small integers and small finite field elements are represented by values that could not be valid handles
    - note IS_INTOBJ, INTOBJ_INT and INT_INTOBJ for small integer conversions
  - Lots of kernel API for working with objects
    - TNUM_OBJ (first level type), SIZE_OBJ, ADDR_OBJ (C pointer to data), TYPE_OBJ (full type)
    - IS_MUTABLE_OBJ, MakeImmutable,.....

Other Kernel APIs

- The second example uses several more kernel interfaces
  - NEW_PLIST, SET_ELM_PLIST, etc. for creating plain lists
  - PROD for arithmetic (see also SUM, DIFF, etc.)
- These are flexible (PROD will multiply anything).
  - If you know what objects you will have it will be a bit faster to call the multiplication directly.
- There are lots more – strings, general lists, calling functions, ... too many to talk about them all.

Rules of Kernel Programming what was that CHANGED_BAG?

- Three golden rules:
  - Real C pointers into objects (returned by ADDR_OBJ) must **not** be held across anything that could cause a garbage collection (GC)
  - If you add a new object to another one (eg put it in a list) you must call CHANGED_BAG on the container.
    - otherwise the new object may get lost in a GC
  - Don't use malloc
    - actually using it a little bit is usually safe, and it's safe if you don't ever want to expand the GAP workspace

Common Kernel Gotchas

- More things can cause a garbage collection than you expect
  - printing (might be to string stream)
  - generic list or record access (might be handled by GAP methods)
  - integer arithmetic (might overflow to large integers)
- Be careful of things like
  - ELM_PLIST(l, 3) = <something that might cause GC>
  - This expands in C to
    - *((*l)+3) = <something>
  - The compiler is allowed to follow the inner * then evaluate the RHS then the outer *.
    - A GC breaks this

Data Objects

- Positional and Component objects are made from lists and records using Objectify
  - they contain their Type and data accessible with ![] and !.
- Data objects also contain their Types, but the data is only accessible via kernel functions
  - Data can be anything you like **except** bag references.
    - the garbage collector doesn't see inside them
  - At a minimum, construction and basic access functions need to be written in the kernel.
- Compressed vectors are done this way.

The GAP Compiler

- Converts GAP code into kernel functions.
  - still has to do lots of checks, so not usually as fast as hand-written C
  - compiled code can be loaded into a running kernel (on UNIX or Mac OS)
- Performance gain is significant for code that spends a lot of time in loops, small integer arithmetic, etc.
  - not significant if code spends most of its time in the kernel or elsewhere in library.

Compiler, Example

```
{caolila:17}cat foo.g
foo := x -> [x,x^2,x^3];
{caolila:18}gac -d -C foo.g
 .....
{caolila:19}gac -d foo.g
.............
{caolila:20}ls -l foo.so
-rwxr-xr-x 1 sal 158 4999 2007-09-11 10:19 foo.so*
{caolila:21} gap -b
GAP4, Version: 4.dev of today, ....
gap> LoadDynamicModule("./foo.so");
gap> Print(foo);
function ( <<arg-1>> )
    <<compiled code>>
endgap> foo(3);
[ 3, 9, 27 ]
```


Compiled Code
Key lines from foo.c

```
/* return [ x, x ^ 2, x ^ 3 ]; */
t_1 = NEW_PLIST( T_PLIST, 3 );
SET_LEN_PLIST( t_1, 3 );
SET_ELM_PLIST( t_1, 1, a_x );
CHANGED_BAG( t_1 );
t_2 = POW( a_x, INTOBJ_INT(2) );
SET_ELM_PLIST( t_1, 2, t_2 );
CHANGED_BAG( t_1 );
t_2 = POW( a_x, INTOBJ_INT(3) );
SET_ELM_PLIST( t_1, 3, t_2 );
CHANGED_BAG( t_1 );
RES_BRK_CURR_STAT();
SWITCH_TO_OLD_FRAME(oldFrame);
return t_1;
```

Original GAP code (more or less) appears as comments

Exploiting the Compiler

- Apart from just compiling your code, you can
  - compile your code and then hand-optimize critical sections
    - eg replace calls to POW by calls to PROD or even to the product function for particular kinds of objects
      - risk a segfault if you call your function with wrong arguments
  - just use the compiler to generate a shell that can be dynamically loaded then fill in your own C code
    - done in browse, IO and EDIM packages
    - need to make sure your code complies with rules

Interfacing to GAP

Steve Linton

Centre for Interdisciplinary Research in Computational Algebra

University of St Andrews

From the GAP FAQ

8. Programming GAP

8.1: Can I call GAP functions from another programme?

The short answer is no. To explain a little more fully, essentially all the algebraic functionality of the GAP system is written in the GAP language, and so needs the GAP interpreter to run. The interpreter is written in C, but does not coexist happily with other code in the same process for a number of reasons, so there is no sensible way to link GAP into a C, Java or other program as a subroutine library.

What you can do is to run GAP in a child process and communicate with it using pipes, pseudo-ttys, UNIX FIFOs or some similar device. We have done this successfully in a number of projects, and you can contact the support list for more detailed advice if you want to go down this route.

The GAP Group Last updated: Fri Apr 29 13:52:12 2005

The Problem

- To run a GAP process as a slave of some other (local or remote) process
  - might be a general purpose slave executing arbitrary commands
  - might be a specialist repeatedly doing a specific job (ie checking whether a node in a search tree can be excluded on symmetry grounds)
- To do this robustly

Zeroth Approach

- Run a perfectly normal GAP process controlled through pipes
- Send GAP commands through one pipe (do remember to send semicolon-newline at the end!)
- Try to separate the displayed results from the prompts, echos, info messages newlines and anything else being returned
- Parse the results and work out what values they represent (making sure that you have them all).
- Make sure the pipes don't fill up!
- Try to stay out of the break loop!
- Horribly unstable

Refinements to Zeroth approach

- Command-line options
  - -b -q -n -T

    {caolila:11} gap -b -q -n -T

    1+1;

    2
- Carefully crafted commands
  - Print("+++++++",FormattedResult(....),"******\n");
  - don't rely on the read-eval-view loop and mark the start and end of the results unambiguously.
- This can work. Still issues with errors, random messages, pipes filling up.

First (historically) approach

The xgap solution

- Use the -p commnd-line option
  - invented for xgap
  - causes GAP to put special sequences (starting with @) in the output stream
  - indicates whether output is
    - echo
    - results
    - prompt
    - error message
    - help text
    - garbage collector numbers

XGAP mode Ctd

- @ codes are documented in pkg/xgap/src.x11/pty.

```
**    'pX.'                 package mode version X
**    '@'                   a single '@'
**    'A'..'Z'              a control character
**    '1','2','3','4','5','6'   full garbage collection information
**    '!','"','#','$','%','&'   partial garbage collection information
**    'e'                   gap is waiting for error input
**    'c'                   completion started
**    'f'                   error output
**    'h'                   help started
**    'i'                   gap is waiting for input
**    'm'                   end of 'Exec'
**    'n'                   normal output
**    'r'                   the current input line follows
**    'sN'                  ACK for '@yN'
**    'w'                   a window command follows
**    'x'                   the current input line is empty
**    'z'                   start of 'Exec'
```

XGAP mode 3

- It's fairly easy to write code on the client side that uses these sequences to sort out the GAP output stream and associate results with requests.
  - this is how the SAGE interface works (or worked in Jan 2006, anyway)
- Beware that the xgap package may try to autoload
- About the best solutions possible when talking to GAP through stdin/stdout.

A Better Approach

- Much better to separate communication with the client program from stdin/stdout completely
- Have an infinite loop running in GAP reading input from somewhere and writing results somewhere else
  - you gain complete control of input and output formats
  - especially good for a specialised server
  - errors may be a problem
    - see next talk

The GAP/eCLiPse Interface

- eCLiPse is a constraint solver – essentially a backtrack search
  - using GAP to break symmetry in search space
- Create two fifos (names pipes) – togap and fromgap
- GAP in infinite loop
  - reads data from togap (until end of file)
  - writes results to fromgap
  - closes fromgap
- Data can be in any format, opening and closing pipes synchronises

Future Plans

- See previous talk
- With Max's IO package, we can run GAP as a true server, using sockets (or fifos or anything else)
  - read input and send results in any format we choose
- New error handling functionality (see next talk) allows us to catch errors and report them meaningfully to the client, or recover locally.

GAP in the Multi-core World
Questions, not Answers

Steve Linton

Centre for Interdisciplinary Research in Computational Algebra

University of St Andrews

Background

- It is becoming hard to buy a single-core CPU
  - dual core is standard, quad core is easily available
  - Intel are talking about 80 cores! Sun processors are optimised for running hundreds of threads
- Only way to exploit this at the moment is to run multiple copies of GAP
  - or GAP and other programs
- Various ways of coordinating them to distribute a computation
  - ParGAP, dc, SCSCP

Problems

- You may not have that many independent problems to solve
- Modern multi-core processors share cache
  - two copies of the GAP interpreter and key workspace data will compete for the cache
- Passing data between processes with current tools is slow and inefficient
  - have to serialise
  - lose Attributes and other context
  - have to maintain two copies of possibly large data structures
- RAM may be the limit rather than CPU, so you can't have two large GAP processes running at once

Multi-threaded GAP

the dream

- Exploit multiple CPUs in a single GAP process
  - one workspace, one problem, solved sooner
- Ideally unchanged programs would compute correct results twice as fast on twice as many cores
  - rather optimistic
- Failing that, it should be really easy to adapt our programs to work correctly in parallel
  - going to depend on the programs (and on your notion of easy).

Parallel Programming Environments

- What little I know
- POSIX or Java threads
  - totally explicit control by user, user has to ensure safety, thread creation not cheap
- OpenMP markup
  - user annotates code to say things like
    - “iterations of this loop are independent”
    - “these two things can be done in any order”
- Lightweight thread approaches
  - very cheap to create, only become real threads if there is a CPU to run them.

Questions

- How important is this?
- Where do we put our effort?
  - into better communication between separate GAP processes and maybe some general algorithms designed for this?
  - into kernel (and maybe library) routines that can use multiple cores for specific jobs only
    - matrix multiply, Todd-Coxeter, collection, ...
  - into letting users program with threads at GAP level?
  - some more sophisticated parallel environment?

More Questions

- Does anyone know more than me about this stuff?
- Does anyone want to help?
- Where do we begin?
- Does anyone have a clue how to parallelise any interesting algebraic algorithms?