

# Building Applications out of Several Programs

Nick Maclaren

[nmm1@cam.ac.uk](mailto:nmm1@cam.ac.uk)

July 2009

# Purpose of Course

Most **large applications** are built of **multiple programs**  
There are **many reasons** you may want to do this

- May want to use **existing programs**, unchanged
- May want to integrate **incompatible** programs
- May want to change **interface** of existing programs
- May want to run programs on **multiple systems**
- And more ...

# Overview of Course

Mainly the **principles** of **whys**, **whens** and **hows**

Start with **elementary overview**

Then describe **chains**

And other **basic structures**

How to **design** and code **interfaces**

More **advanced** topics (optional)

Problems with **monolithic** programs

Some issues with **separate** programs

And more **advanced structures**

# Beyond the Course

<http://docs.python.org/> and/or books on it  
Go to **Python** courses if you don't know it

<http://www.perl.org/docs.html> and/or books on it  
E.g. **Programming Perl**, Third Edition, O'Reilly Media  
by Larry Wall, Tom Christiansen and Jon Orwant

# Using Separate Programs

Will give the most **common** reasons  
But there are **many, many** others

Golden rules:

- **KISS** – **Keep It Simple and Stupid**
- Only divide up in ‘**natural**’ ways
- Use a **simple, debuggable** structure
- **Interfaces** are **AS** important as **components**

# Basic Controller Model

- **Controller** ( $\equiv$  **harness**) does very little  
Controls **how** programs start and communicate  
Handles program failure (**return code** or **crash**)
- Programs run in **isolation** from each other  
No **communication** except as set up by the **controller**  
Components can be existing, free-standing programs  
They do **all** of the real work
- Not the **only** model – just the **simplest**

# Using Existing Programs

May want a different sort of **interface**

Simpler/clearer/area-specific/flexible/GUI

May need to **automate** some analyses

May need to **combine** several programs

Possibly in **binary**, **different languages** etc.

Avoids **mixed-language executable** problems

This is just industrial-strength **scripting**

- $\equiv$  **programming** using **processes**

# Splitting Up Programs

- Can often increase **debuggability**  
Provides **interfaces** to locate **bugs/problems**  
Can often **debug** components **separately**  
Can use to avoid **library incompatibilities**
- Sometimes **critical** for **efficiency**:  
Run in **parallel** on **multi-core** systems  
Can even use some components **remotely**  
**GUI** code will **'poison'** **HPC** (**SMP** or not)



# Choice of Languages

- **Components** can be written in anything  
**Process** interface is **language-independent**  
**Binaries** are usually in no known language!  
And, yes, **each program** can be **different**

Controlling programs: **Python**, **Perl** etc.

**C++**, **Fortran 90** etc. are OK but more **effort**

Complex **shell scripting** is for **masochists**

- **Python** is the recommended tool

# Advanced Controllers

Iris Explorer (from NAG)

Data Explorer (ex-IBM)

Many others used in commerce

Mostly GUI-based, hard to learn

A few users in the University

Worthwhile for very heavyweight tasks

Some job schedulers fall into this category

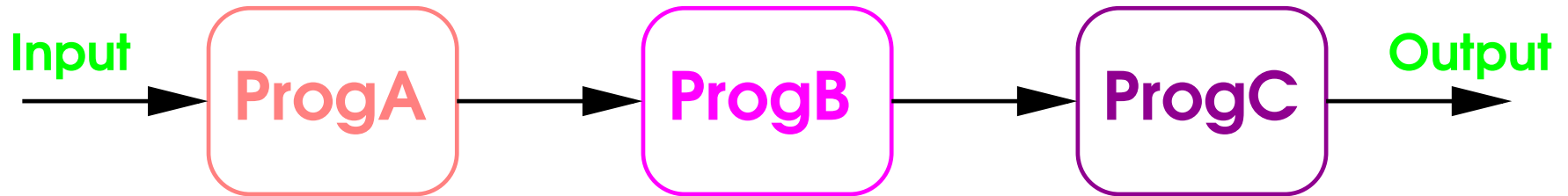
# Basic Structures

Some **structures** are easy to **use/debug**  
Can even **prove** mathematically **correct**  
**90%** of applications can use **one** of them  
**99%** can use a clean **combination**

Will mention **places** where **problems** occur  
But mainly to say “**don't go there**”

Remember, one **golden rule** is about these

# Simple Chains



**ProgB** must wait for **ProgA**

**ProgC** must wait for **ProgB**

**Data/control flow is from input to output**

# Basic Simplex Chains

A.k.a. pipes, streams, FIFOs, queues, sockets  
Serial from single input to single output  
Can use large buffers and many CPUs/systems  
Streaming I/O can be optimally efficient

Control and data flow are simply linear  
Done automatically by shell pipelines  
Very simple, very reliable, easy to test

Will return to interactive chains later

# Controlling Chains

- The **shell** creates a **pipe** (with **two ends!**)
  - Starts program **A** and feeds **output** into **pipe**
  - Starts program **B** taking its **input** from **pipe**
- But does **NOT** handle **errors** correctly!

**Controlling programs** should do the same  
No other **synchronisation** needed or wanted  
But should also **detect errors** . . .

Using **default I/O** almost always OK

# Python Chain Controller

huey | dewey | louie

```
from sys import stdin, stdout
from subprocess import Popen, PIPE
p1 = Popen(["huey"],stdout=PIPE)
p2 = Popen(["dewey"],stdin=p1.stdout, \
           stdout=PIPE)
p3 = Popen(["louie"],stdin=p2.stdout)
rc = p3.wait()
```

# Python Error Handling

- Wait for or kill **all** subprocesses
- Print **subprocess name** and **error code**
- Possibly trap exception **OSError**

See example in **Python** library manual

Only **8** not-very-complex **lines**

Check return codes from **all** subprocesses

Best programs check have **reached EOF** on **input**

And get **EOF** when **all output** has been **read**



# Python Error Code

```
from __future__ import print_function
try:
    rc = call(cmd+args, shell=True)
    if rc < 0:
        print("sig", -rc, file=stderr)
    else:
        print("exit", rc, file=stderr)
except OSError as e:
    print("fail:", e, file=stderr)
```

# Perl Chain Controller

See “Programming Perl”, chapter 16

Some very simple cases are easy

In general, not much easier than C

Very little error handling by default

Remember to clean up environment

- Not advised unless you know Perl already

# C/C++/POSIX Controller

Too complicated for this course

- Avoid this if you possibly can

Need `pipe()/dup2()/fork()/exec?()/waitpid()`

Plus cleaning up `programming environment`

`Not doing so` can cause `confusion/chaos`

Example code is shown later

And that's `just` for the `simple case`!

# Fortran Controller

Calls **C** to do actual **process control**  
**Advanced logic** can be in **Fortran**

Not worthwhile for **simple chain** control  
Starts being so for **master/worker**

Please ask for help if doing this

# Python Component

This is what 'cat' looks like:

```
from sys import stdin, stdout
while 1 :
    line = stdin.readline()
    if not line :
        break
    stdout.write(line)
```

## Or Perl?

```
while (<STDIN>) {  
    print $_;  
}
```

But you will need to add **error handling!**  
**Perl** includes very little **automatically**

# Or Fortran?

```
character, len=big_enough :: buffer
do
    read (*,'(a)',end=10) buffer
    write (*,'(a)') buffer
enddo
10 continue
```

Fortran errors default to fatal, as in Python

## Or C++? Or C?

```
string s;  
while (cin >> s) cout << s << std::endl;
```

```
char buffer[big_enough];  
while (fgets(buffer,sizeof(buffer)-1,stdin) {  
    buffer[sizeof(buffer)-1] = '0';  
    fputs(buffer,stdin);  
}
```

Remember about **error handling** here, too



# Golden Rules of I/O

- Use streaming I/O – allow reblocking
- Don't reposition/handshake in any way
- For performance use binary/unformatted  
Use large buffers (64+ KB) if possible
- Check but distrust all error codes  
Close explicitly and check return code

# Another Method

Program **A** spawns program **B** (i.e. **fork+exec**)

Program **A** waits for program **B** to finish

First orders start of **B**, second orders end of **B**

Or controlling program runs **A**, and then **B**

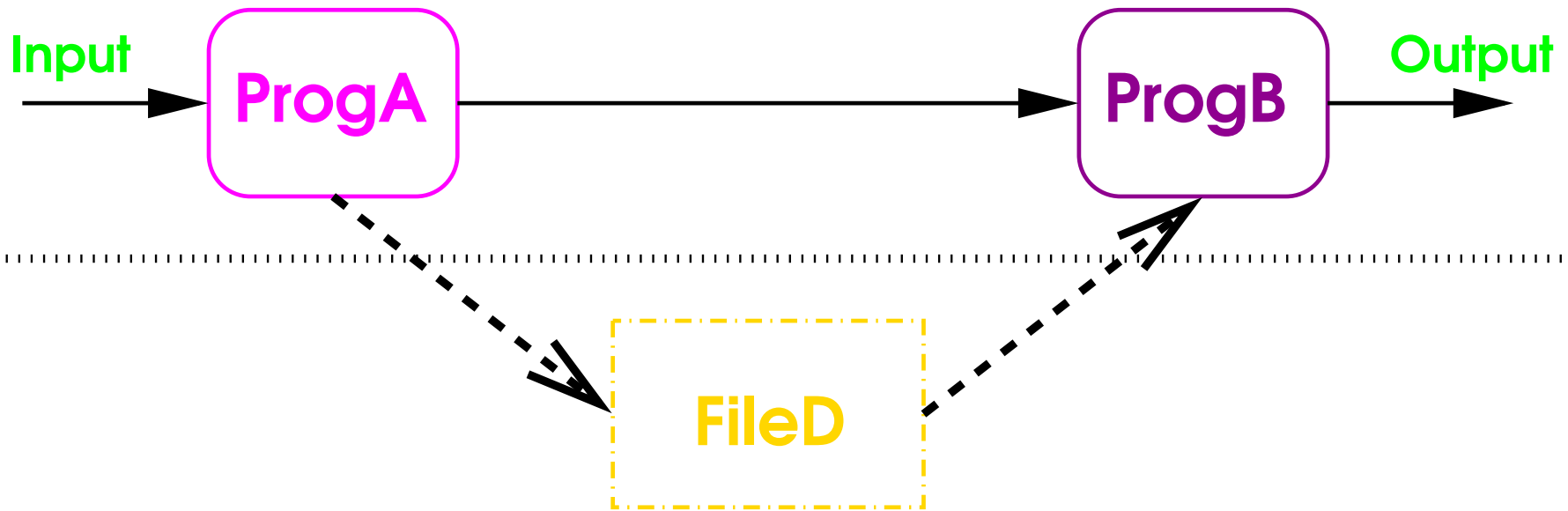
Can also send messages down pipe

Or by using signals (not recommended)

This logic is needed if using files for data

**Must** close output before opening for input

# Using Files in Chains



**ProgA:** write & close **FileD** THEN prod **ProgB**

**ProgB:** wait for **ProgA** THEN open **FileD**

# Python Example (1)

```
from subprocess import Popen
rc1 = Popen(["A"]).wait()
rc2 = Popen(["B"]).wait()
```

## Program A:

```
output = open("fred","w")
output.write(some_data)
output.close()
```

## Program B:

```
input = open("fred","r")
...

```

# Python Example (2)

## Program A:

```
output = open(filename, "w")
output.write(some_data)
output.close()
p1 = Popen(["B"])
p1.wait()
```

## Program B:

```
input = open(filename, "r")
...
```

# Python Example (3)

## Program A:

```
output.write(some_data)
output.close()
stdout.write(filename)
```

## Program B:

```
name = stdin.readline()
input = open(name)
...
```

# GUIs - X and MS Windows

Most **common** requirement for **splitting** programs

Mandatory **event loop** with no long delays

Does **horrible** things with **networking**

Often demands specific **compiler options**

**Name clashes** and other **problems** abound

**Foul** to debug – **repeatability?** **evidence?**

May even **lock up console** and **force reboot**

Solution: **separate off** and **Keep It Simple**

# GUI Input and Output

- GUI component creates/checks input files
- Analysis program runs non-interactively
- GUI component displays/selects results

Many commercial/production programs do this

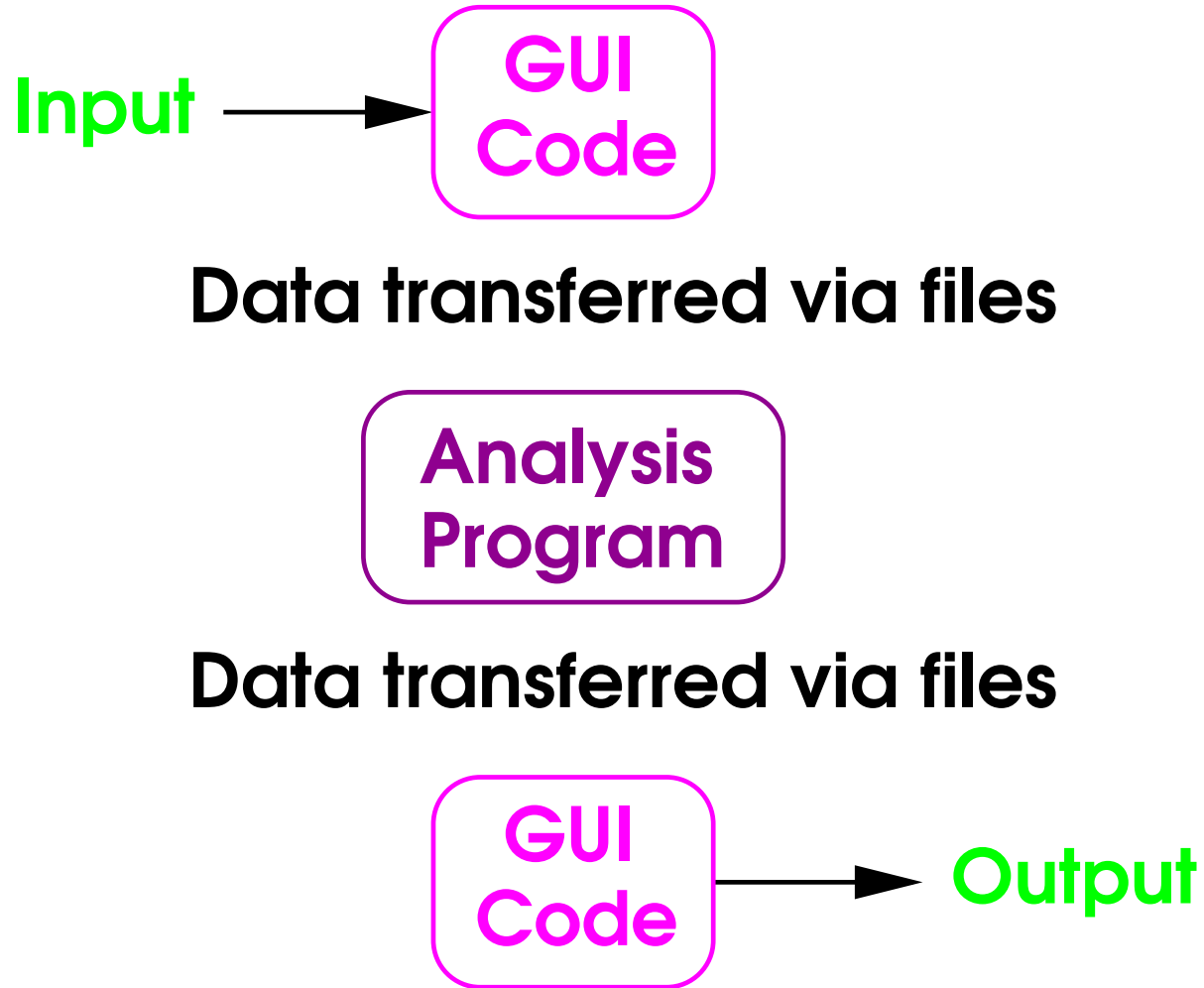
Almost universal in HPC environments

4 decades of experience supports this design

It can save a LOT of debugging time!



# Simple GUI Design



# Why Do This?

Can **rerun any stage** if it fails

**Very** useful for **debugging** etc.

Or if you just need to go to bed!

Files provide **proof** of where **errors lie**

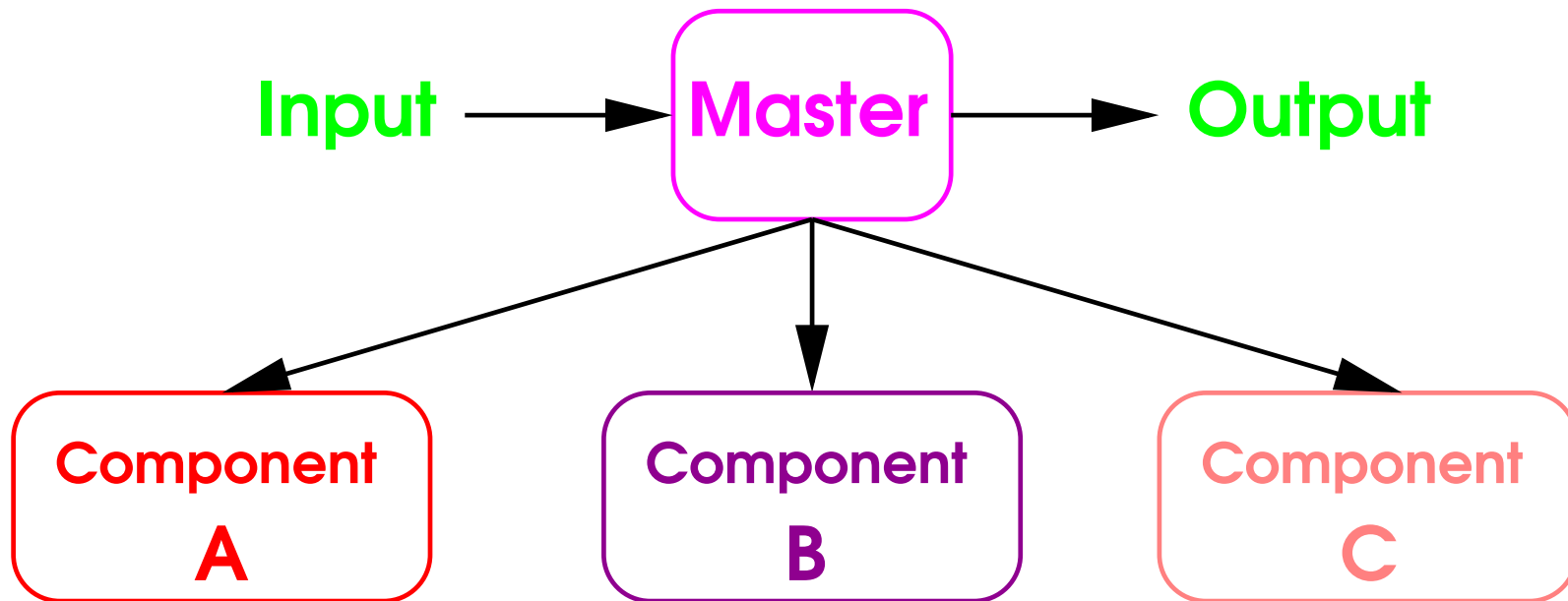
Can **automate** (script) **creation of input**

Or **changes**, or **analysis of output** . . .

**Analysis** may take days or **need restarting**

Or need to be run on **another system**

# Master/Worker



**The master may just do control, or may also do processing (but not in parallel to workers)**

**Workers may run serially or in parallel**

# Serial Master/Worker

The **master** runs the workers **serially**  
Possibly **interleaved** with its **own work**  
Simple, reliable, but **not parallelisable**

**Spawn** and **wait** for **component A**  
Do some **computation** in the **master**  
**Spawn** and **wait** for **component B**  
**Spawn** and **wait** for **component C**  
And so on . . .

Return to the **parallel** version later

# Warnings

Don't be **clever** when sharing **descriptors**  
There are some evil '**gotchas**' lurking

Watch out for **environment pollution**  
Far more of this than most people realise  
E.g. **signal handling** and **limits**

# Tree Structures

Serial master/worker can make a tree  
Just function calls to separate programs

Don't expect recursion to work!

Only real problem is handling failures  
Killing a process doesn't kill children

# More Complex Structures

Key concept is a transaction (coming next)  
Effectively an atomic message+reply

Streaming I/O can be used – with care  
But remember pipes have finite capacity  
Using files for bulk data is much safer

Will give guidelines for safe use  
Experts can and do break the rules

# Simple Transactions

Program **A** writes **all** of its **request**

Program **B** reads **all** of the **request**

Program **B** writes **all** of its **reply**

Program **A** reads **all** of the **reply**

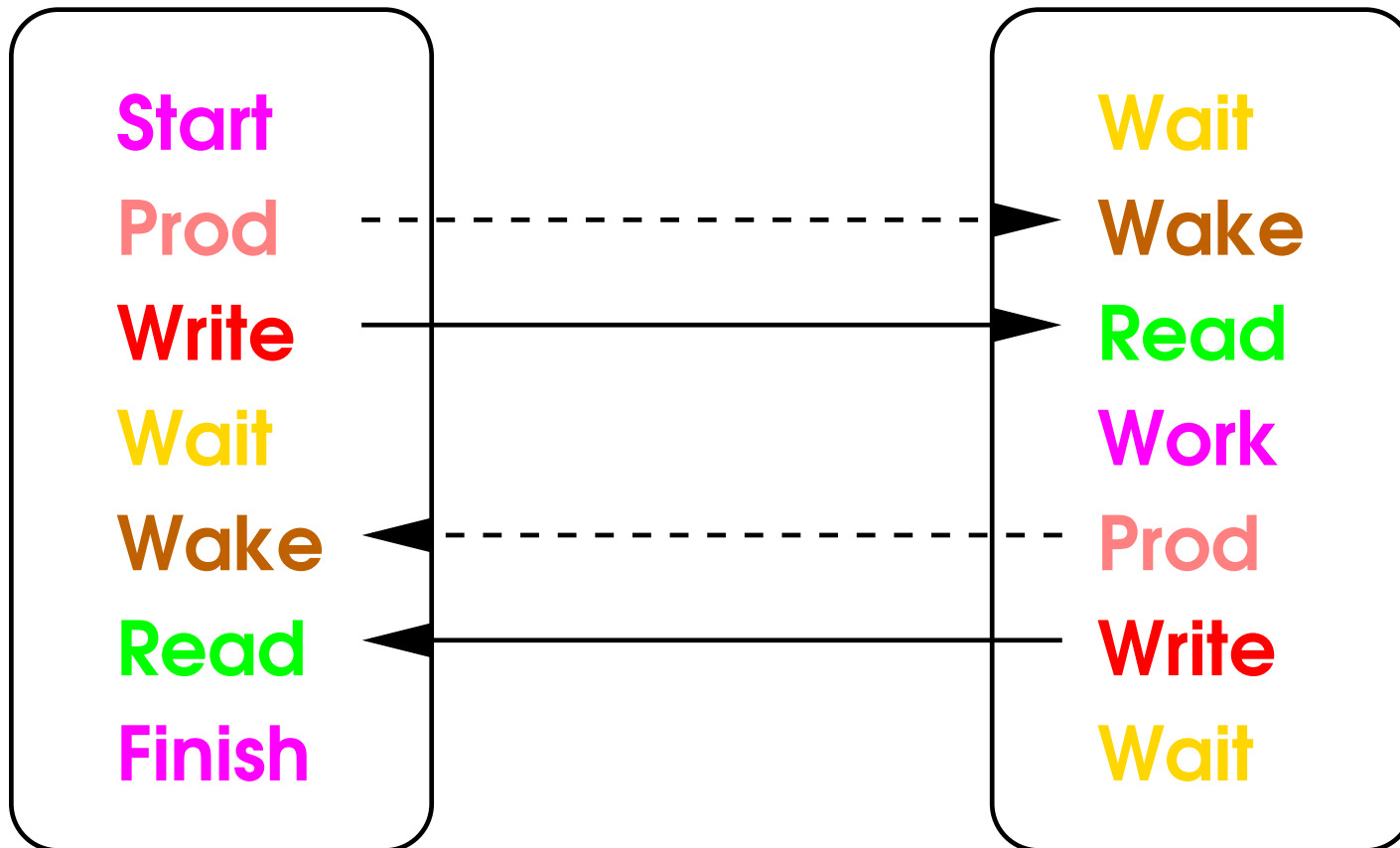
**NO** other **communication** during that

Don't **start reply** while **reading request**

**Read reply** before sending **next request**



# Transactions



# Parallel Master/Worker (1)

The **master** runs the **workers** in **parallel**  
**Workers** talk only **to/from** their **master**  
Very good for **SMP** systems and **clusters**

**Read** input and **divide up** work

**Spawn** all of the **workers**

**Wait** for all of the **workers**

**Collect** their **work**

**Combine** it and **write** output

## Parallel Master/Worker (2)

Master/worker **communication** can be a **problem**

Easiest if master supplies **input initially**

And then just collects **results at end**

Usually **safest** when files are used for this

Sometimes **ongoing** communication is needed

See later on **duplex pipes**

But **avoid** it if you **possibly** can

Use **simple transactions** if you must do it

# Simple Client/Server

The **server** runs as a **daemon** (indefinitely)  
It **waits** for **requests** and responds **serially**

**Clients** gather input and send **requests**  
Wait for **reply** and then produce output  
And **possibly** do this repeatedly

**msntp** is a **very** simple example  
**exim** is a more realistic one

# Combination Structures

Can **combine** above **structures** in many ways

A **component** can be a **combination**

But remember to **KISS!**

Beyond that, **really** don't go there

Virtually **impossible** to debug

Some **distributed applications** do this

Schedulers, desktops, Grid software

**Administrators** curse them, vigorously

# Data Interfaces

Design like **external** interfaces

You don't make **errors**? – I do, often

Good way of **simplifying debugging**

Programs should **check input** for validity

**Checking output** can be worthwhile, too

Be **thorough**, but no need to be **paranoid**

It really will **save you time**, overall

# Specific Checks

Check **formats** – bad ones may mean **wrapping**

Check **validity** – “NaN” is **Not A Number**

Check values are in **plausible range**

Check **consistency** – e.g. **number count**

**Failures** can mean source program **crashed**

**Any check** may pick up **data corruption**

And **anything else** you can think of

# Designing Formats

**KISS**, and include **cross-checks**

Include **some values** just for **checking**

Counts, maxima, sums, whatever makes sense

Not just **N values**, but **count & N values**

Or **N values & terminator**, or both

5 1.2 2.3 3.4 4.5 5.6

1.2 2.3 3.4 4.5 5.6 -1.0e30

5 1.2 2.3 3.4 4.5 5.6 -1.0e30



# Document It!

A **block comment** in your code is easy and good  
For example:

```
# All main items and rows start on a new line
# Extra spaces and newlines in numbers ignored
#
# Title and author in free text
# Date in format 01 / Apr / 2006
# Row and column sizes
# Data by row appended with -1.0e30
```

# Structured Data

Object = < Vector | Matrix >

Vector = Size Newline \  
Values(Size) Newline

Matrix = Row\_size Column\_size Newline \  
Rows(Column\_size)

Row = Values(Row\_size) Newline

Value = < 'Missing' | Floating-point >

You can **spot problems** in the **format** you use

Advantage is your **program** can **decode it**

And, with care, can **detect** and flag errors

# You're Now Using BNF!

Read up about **BNF** (Backus–Naur Form)

**Wikipedia** is easier than **textbooks**!

[http://en.wikipedia.org/wiki/Backus–Naur\\_form/](http://en.wikipedia.org/wiki/Backus–Naur_form/)

It is **NOT** complex, and very **useful**

Don't worry about **notation** – anything goes

You want it **mainly** to keep your **thoughts clear**

And to **ensure** that your **code can parse it!**

Some **Fortrans** and **C90** didn't and ...

# Advanced Topics

Start with **problems** of **monolithic** programs  
And **some** that can arise with **separate** programs

It is worth knowing what the **issues** are  
Mainly to know what examples **not to follow**  
And **when** to take a **different approach**

- Beyond here is background information only  
I.e. **why** do the above, and what **not to do**

# Monolithic Program Issues

Can be **avoided** by using **separate** programs  
Don't **panic** over them, but **recognise** them  
**Split up** if it makes development **simpler**  
But **interfaces** need **design and coding**, too!

Don't **mince** applications for the **sake** of it  
**1970s** (and later) **computer science dogma**  
'**Software Tools**', **S** tend to follow this dogma

**KISS** and be **cautious**, and all will be well

# Common Incompatibilities

Only a **few** languages can be **linked** together  
**Python, Perl, C++, Fortran 90** must be 'master'

**External name clashes** (not easily soluble)

**Incompatible** use of **stdin** and **stdout**

**Run-time systems** often **incompatible**

**Two garbage collectors** is **Bad News**

Worst is basic **paradigm incompatibilities**

E.g. are **exceptions**, **longjmp**, **signals** allowed?

# HPC, OpenMP etc.

Exactly the **converse** of **GUI** requirements

Can dive into **libraries** for hours

Needs **aggressive optimisation** and more

Often need special **scheduling options**

Very often want to run in **background**

Or even on a **remote** (and different) system

Solution: create input and **'run in batch'**

# 'Interactive' Chains

Buffered output need not appear until end

In extreme case, not until input is closed

Not a problem for 'batch' processing

But very confusing if you aren't expecting it

Need an end-to-end flush/push – don't have one

Can 'solve' with non-blocking/unbuffered I/O

Details are very messy and system-specific

Avoid if you can – much less efficient



# Duplex Pipes

I.e. ones where messages are being sent **both ways**

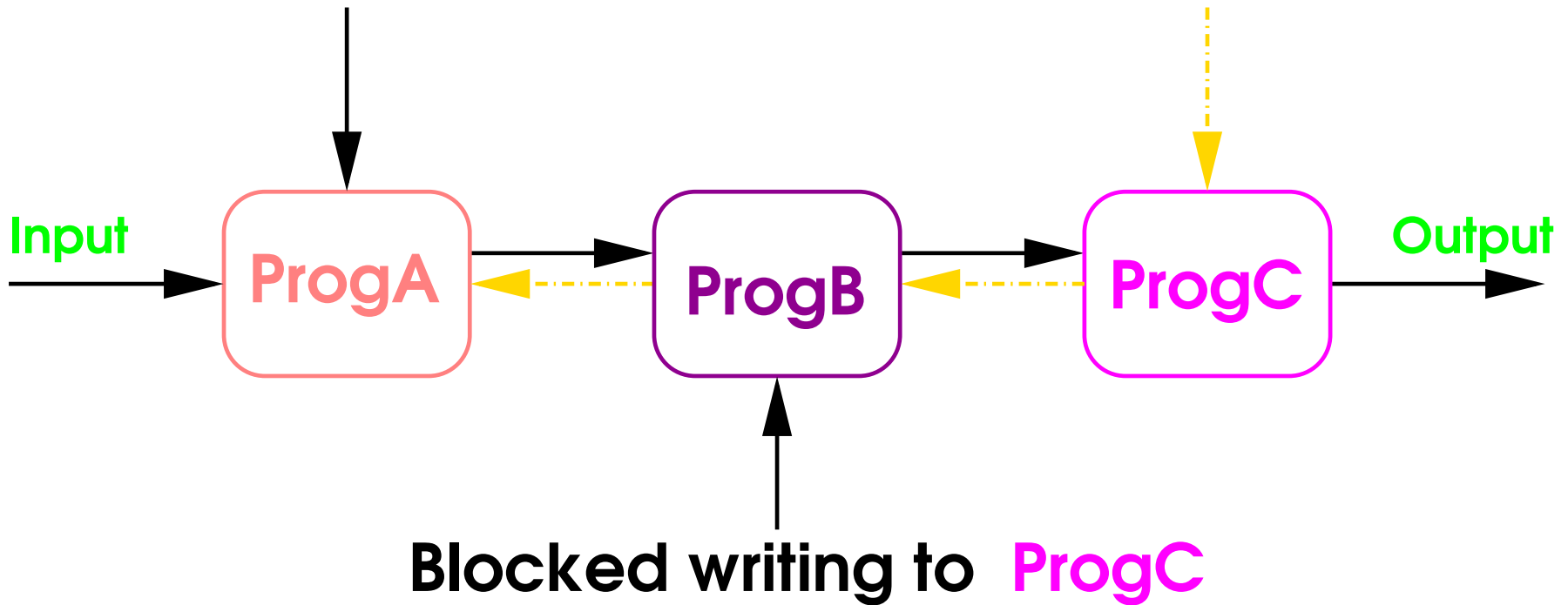
Look **simple**, but **aren't** (even theoretically)  
Seriously **misdesigned** in **POSIX** (**Microsoft?**)  
OK if **careful** – easy to cause **deadlock**  
Don't mix at all well with **streaming**

Solution:

Communicate using **simple transactions** only  
All **messages** are **short** – won't **block pipe**  
Use **files** if any danger of doing that

# Duplex Pipes

Blocked writing to ProgB      Trying to write to ProgB



**If buffers fill up, the application can deadlock!**

# File Access

Don't **trust** consistency **guarantees** too far

Can **read** arbitrarily often in **parallel**

**Or update** (write) from **one** component

Be careful when **changing** between these

Close **all** uses in **all components**

Handshake to **all** components that **use file**

Only **then** can safely open file again

# File/Pipe/Socket/Memory I/O

Flush/push/fsync unreliable even for local files

Don't trust blocking/non-blocking

POSIX rules are not what they appear

Don't use asynchronous I/O

Shared memory can be very efficient

Treat it like I/O – i.e. handshake

Don't assume consistency by magic

# Specific Unix Problems

Some things pass through **fork+exec**  
File descriptors, signal mask,  
environment vars, limits . . .

**Shells** have some hacks to **reset them**

You may need to **do the same**

**Critical** when calling **unclean components**

**Microsoft** probably has similar **gotchas**

# Socket/etc. Problems

Sockets are very 'active' objects

Any access can affect other uses

Stray open descriptors can delay close

In extreme cases, can hold up output

Can be prone to unexpected temporary hangs

Time-dependent code using them is tricky

Look at code of OpenSSH for examples

# Using (Avoiding) Threading

Threads are **NOT** the solution!

Solve one problem, add half-a-dozen more

Details are beyond scope of this course

Only real use is to avoid blocking problems

Particularly relevant for duplex pipes etc.

Need considerable experience, even so

OpenMP implementations use them – don't ask

Rumours are that some GUI libraries do, too

# C/POSIX fork+exec

This is the C code for the chain controller  
Complete with tolerable error handling

It is shown mainly to put you off  
Please ask for it if you really need it



# C Chain Controller (1)

```
/* We start in the parent */  
if (pipe(in) != 0 || pipe(out) != 0 ||  
    (pid = fork()) < 0) fatal();
```

There are now **two processes** running **this code**

## C Chain Controller (2)

```
if (pid > 0) {
/* This is in the parent */
    if (close(in[1]) != 0 ||
        close(out[0]) != 0) fatal();
    if (write(out[1],...) < 0 ||
        close(out[1]) != 0) fatal();
    if ((len = read(in[0],...)) < 0 ||
        close(in[0]) != 0) error();

    if (waitpid(pid,&status,0) < 0) fatal();
    if (! WIFEXITED(status) ||
        WEXITSTATUS(status) != 0) error();
```

## C Chain Controller (3)

```
} else {  
/* This is in the child */  
    if (close(in[0]) != 0 || close(out[1]) != 0 ||  
        dup2(in[1],STDOUT_FILENO) < 0 ||  
        close(in[1]) != 0 ||  
        dup2(out[0],STDIN_FILENO) < 0 ||  
        close(out[0]) != 0)  
        fatal();  
    for (i = 0; i <= 63 /* Sigh */; ++i) signal(i,SIG_DFL);  
    if ((k = sysconf(_SC_OPEN_MAX)) <= 0) k = 63;  
    for (i = 3; i <= k; ++i) close(i);  
    execl(spawned_program);  
    fatal();  
}
```

# More Advanced Structures

Don't go there – **really** don't go there  
But you **already use** programs like this  
And you may well **curse them**, vigorously

# DAGs

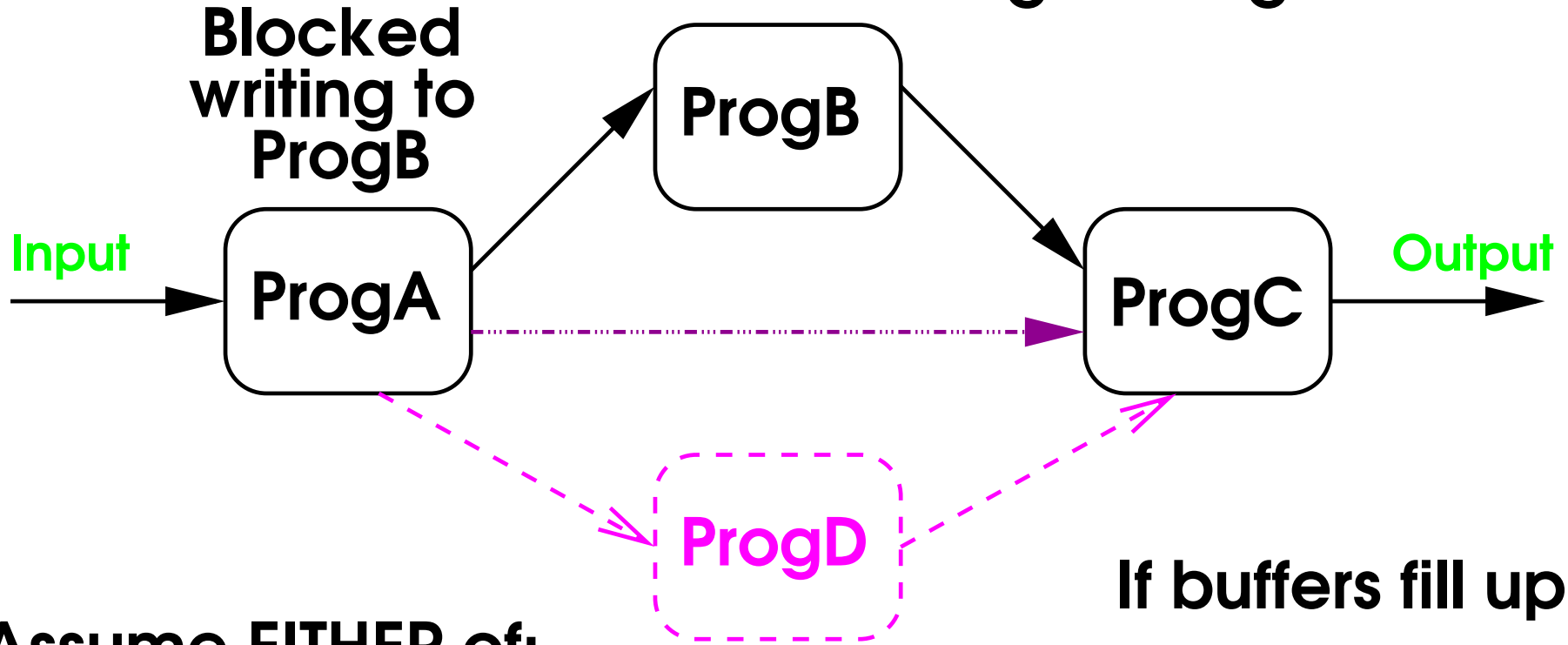
Directed acyclic graphs – ones without loops  
Can be very useful, but easy to deadlock  
Exactly the same problems as duplex pipes

Avoiding deadlock is harder than for duplex  
Needs careful design of data/control flow

Very similar problems to interactive I/O

# Directed Acyclic Graphs (1)

**Blocked writing to ProgC**



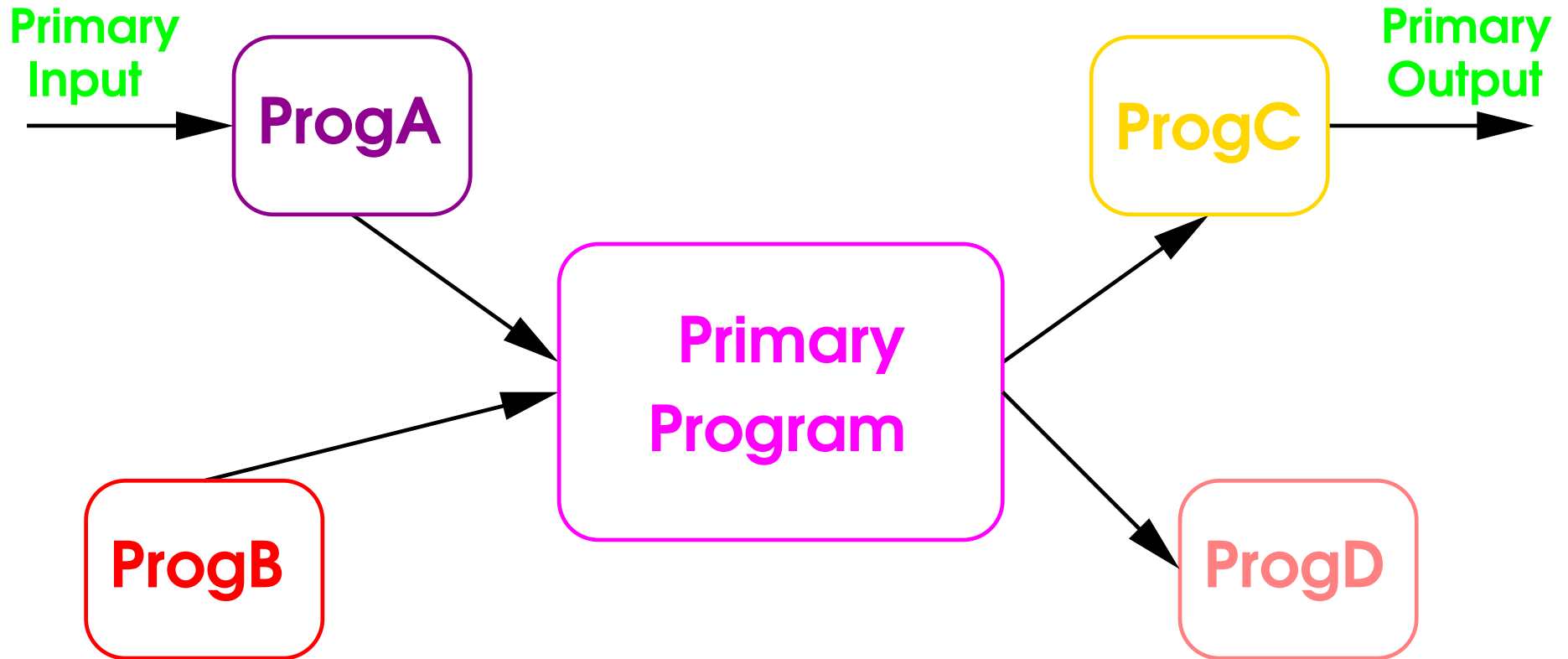
**Assume EITHER of:**

ProgC reads from ProgA

ProgC reads from ProgD

**If buffers fill up,  
the application  
can deadlock!**

# Directed Acyclic Graphs (2)



**Beyond this gets very confusing, very fast**

# Multiple Interactive I/O

Use one primary input and one primary output

If two programs reading, which gets the input?

You have NO way of directing input

Use a single input program (GUI?) to control this

Output is generally easier, but still confusing

And output can be merged in the middle of lines!

Causes confusion when piping through grep etc.

Chaos with full-screen (character addressing)



# Multiple GUI Components

Theoretically, windows are entirely separate  
In practice, this is not quite true

KISS KISS (Keep It SEPARATE, Stupid!)

Focussing, fonts, colours etc. are global  
Some programs handshake via X properties

Don't even think of using threading  
There can be some EVIL socket issues  
⇒ Close all unneeded descriptors

# Really Advanced Use

Existing schedulers, desktops, databases etc.  
Multiple independent daemons, interacting

Design is a DAG with time-ordering on messages  
Need a directed temporarily acyclic graph  
Even major vendors don't get those right

Point gun at foot; pull trigger; **BANG!**