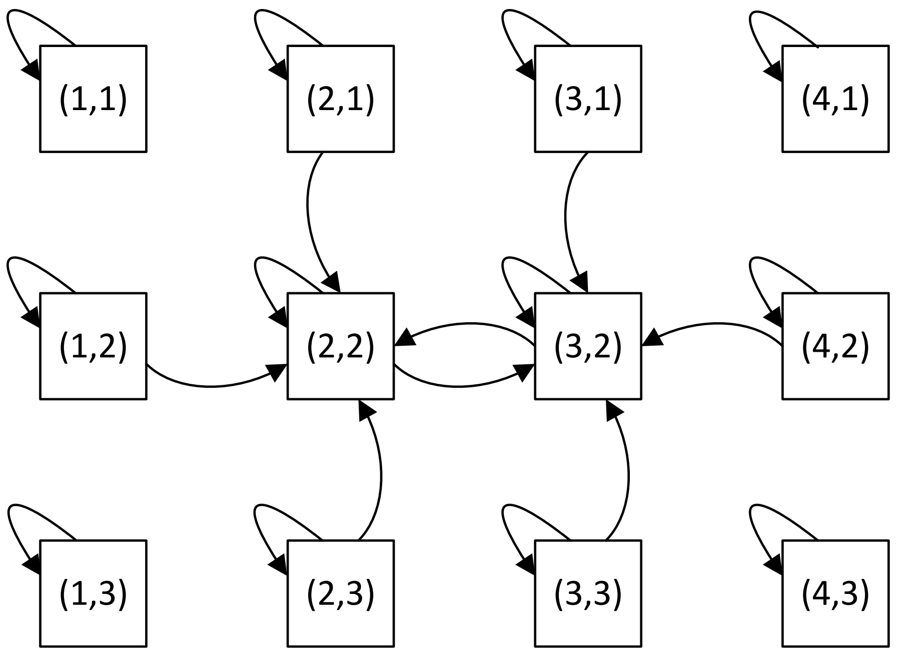
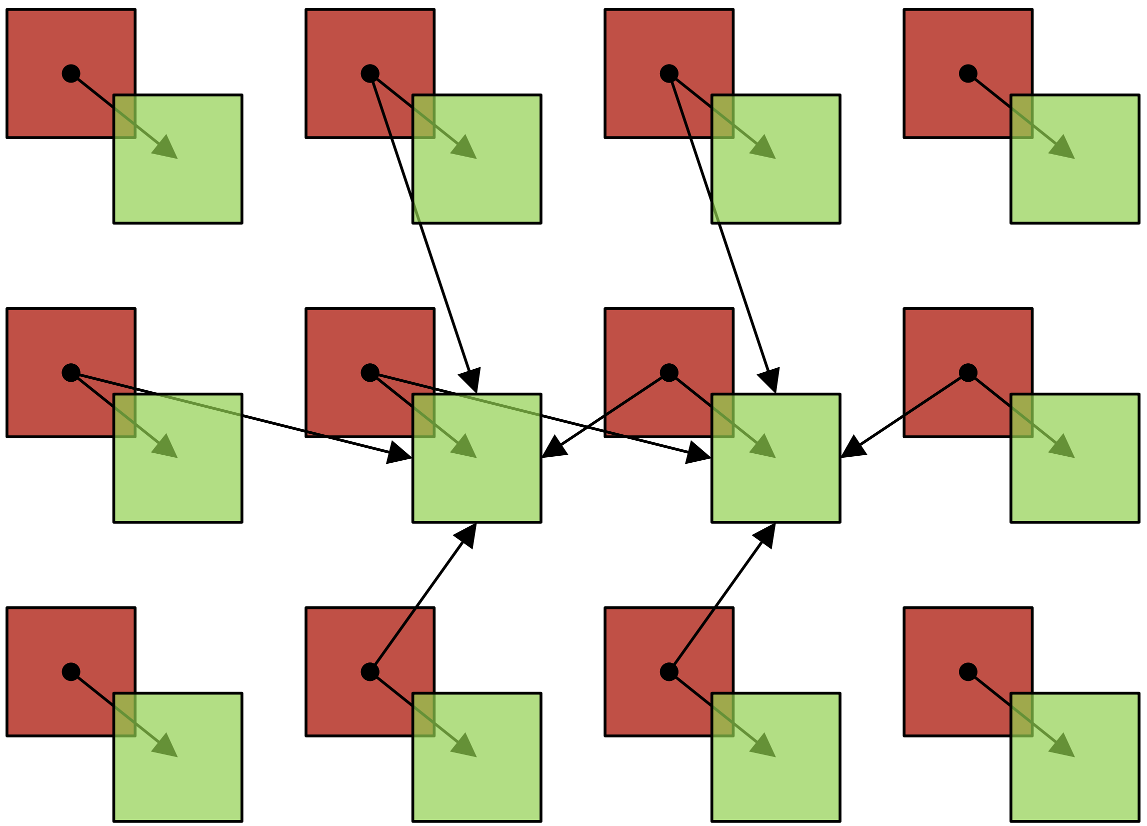
# Revisiting video

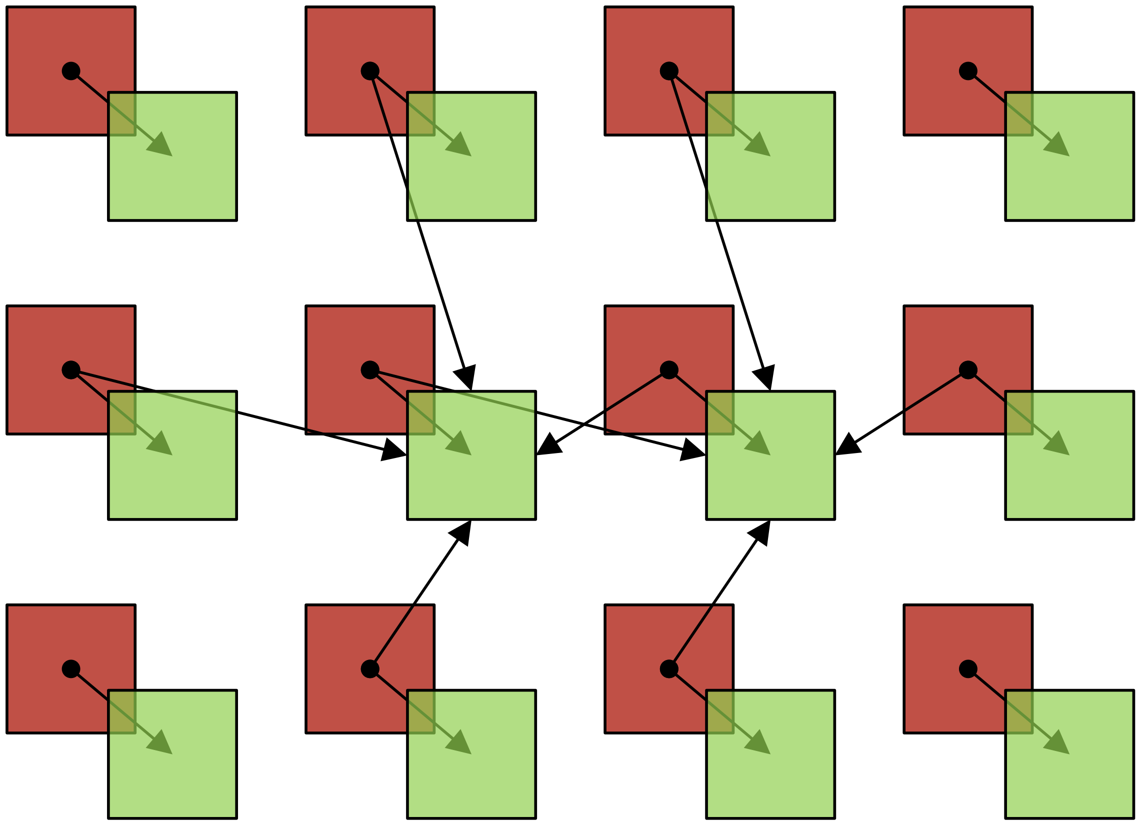
```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn // Receive frame (by value)
){
    // Handle n==0 case
    std::vector<uint8_t> fOut=fIn;

    for(int i=1; i<n; i++){
        fIn = fOut;

        for(int y=1; y < h-1; y++){
            for(int x=1; x < w-1; x++){
                uint8_t nhood [5] = {
                               fIn[(y-1)*w+x],
                fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                               fIn[(y+1)*w+x]
                };
                uint8_t value = min_of_array(5, nhood);
                fOut[y*w+x] = value;
            }
        }
    }
    return fOut;
}
```

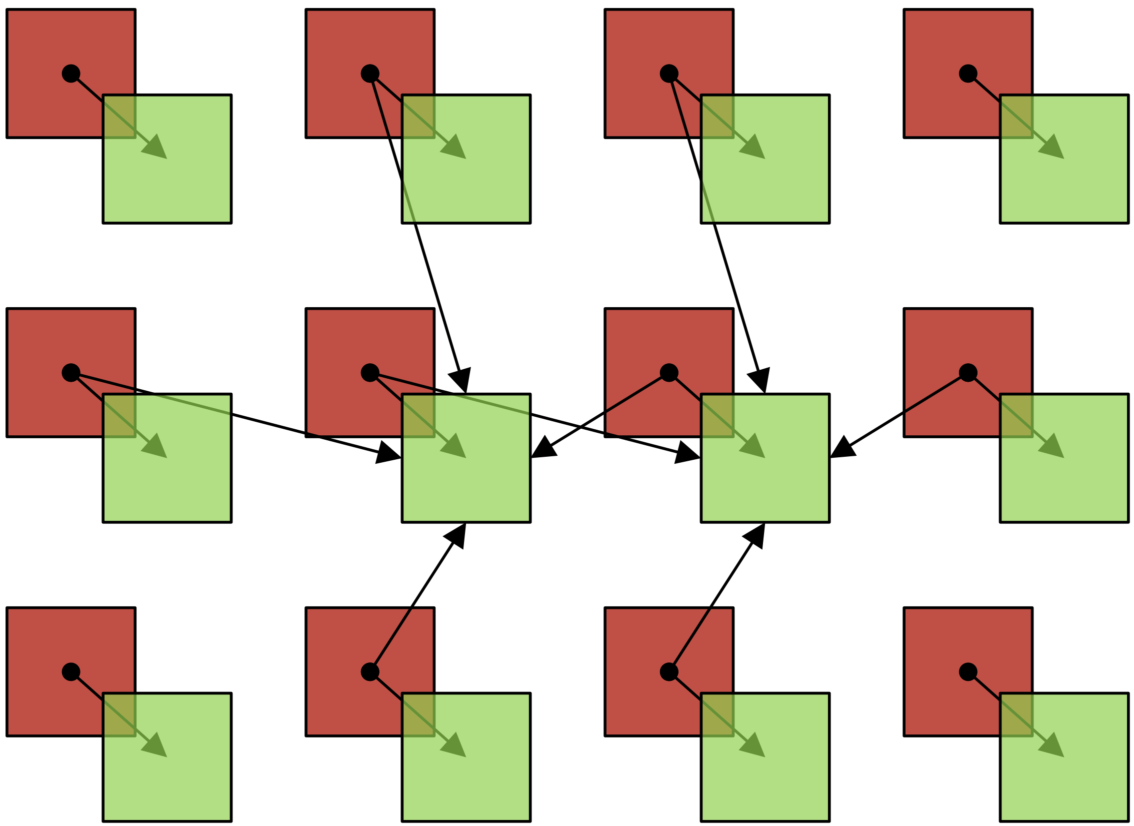
How do you parallelise?

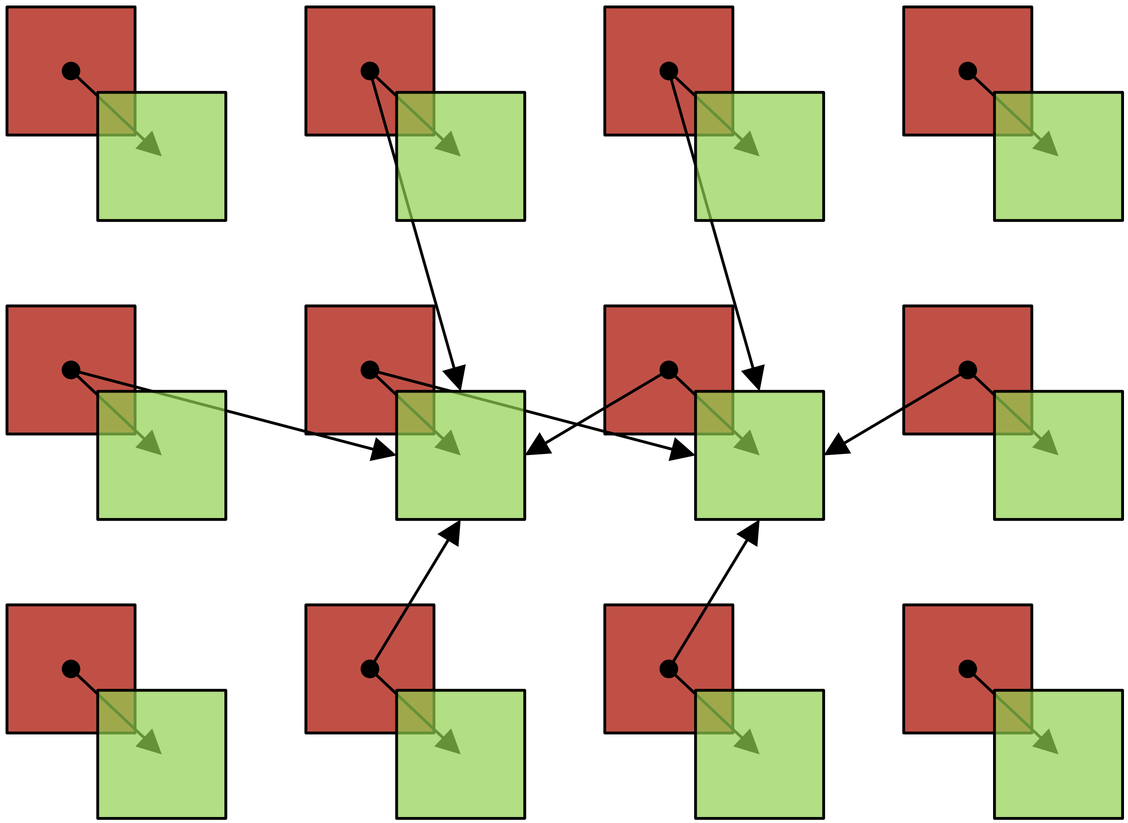
```
uint8_t nhood[5] = {
                 fIn[(y-1)*w+x],
  fIn[y*w+(x-1)],fIn[(y+0)*w+x],fIn[y*w+(x+1)],
                 fIn[(y+1)*w+x]
};
```

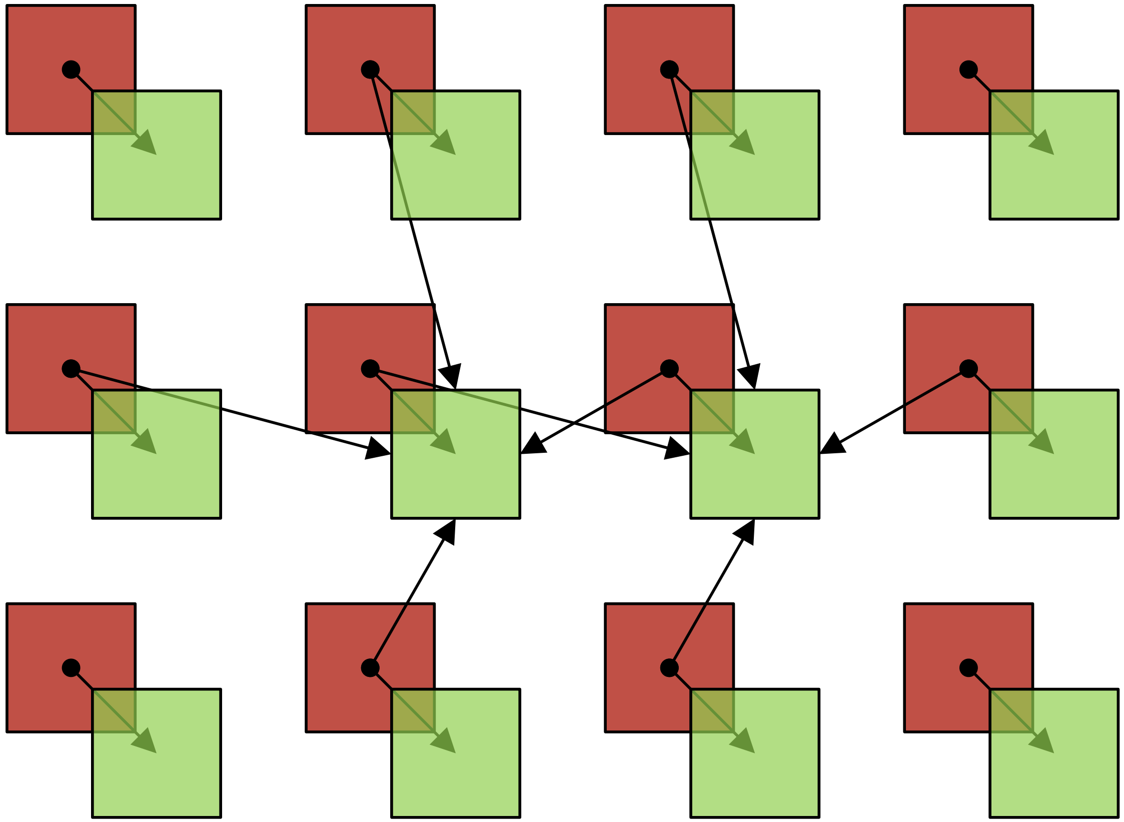


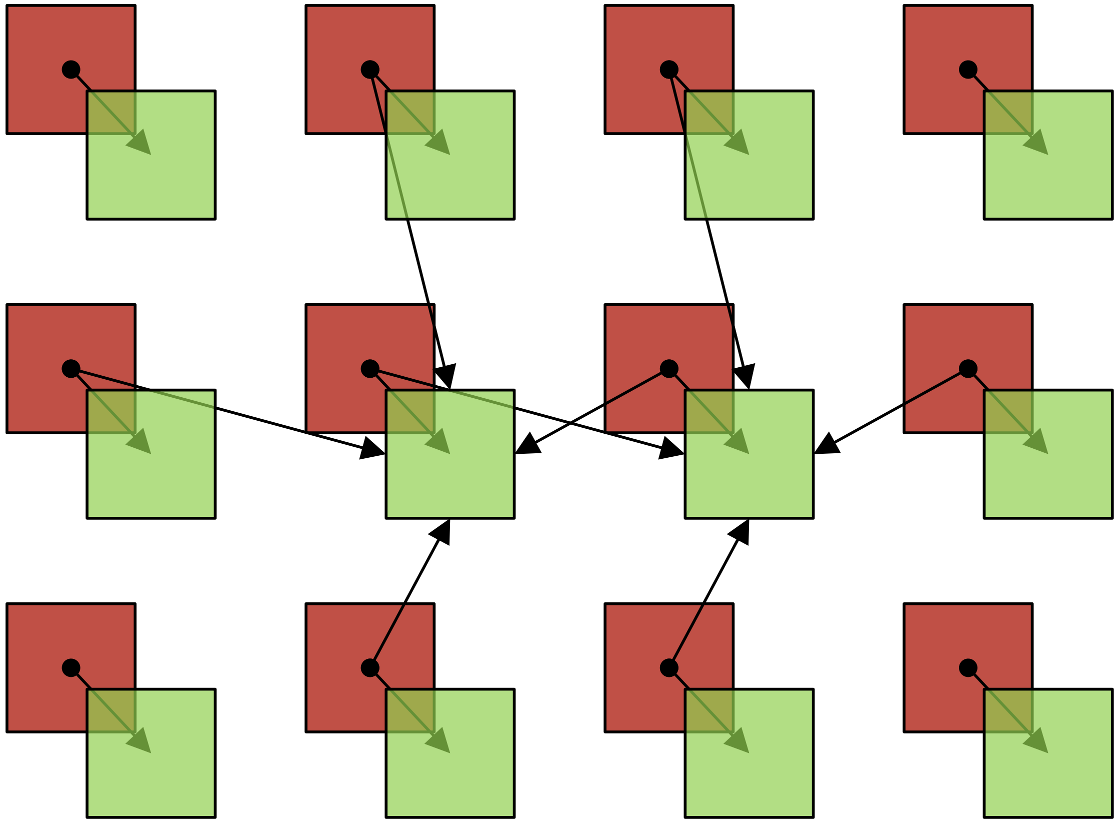












```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    std::vector<uint8_t> fOut = fIn;

    for(int i=0; i<n; i++){
        fIn = fOut;

        tbb::blocked_range2d<int> r( 1,h-1, 1,w-1 );

        tbb::parallel_for( r, [&](const tbb::blocked_range2d<int> &xy) {
            for(int y=xy.rows().begin(); y < xy.rows().end(); y++){
                for(int x=xy.cols().begin(); x < xy.cols().end(); x++){
                    uint8_t nhood [5] = {
                                fIn[(y-1)*w+x],
                    fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                                fIn[(y+1)*w+x]
                    };

                    uint8_t value = min_of_array(5, nhood);

                    fOut[y*w+x] = value;
                }
            }
        });
    }
    return fOut;
}
```

- Strict loop carried dependency
- Pure cyclic chain
  - Impossible to break
- No parallelism...?

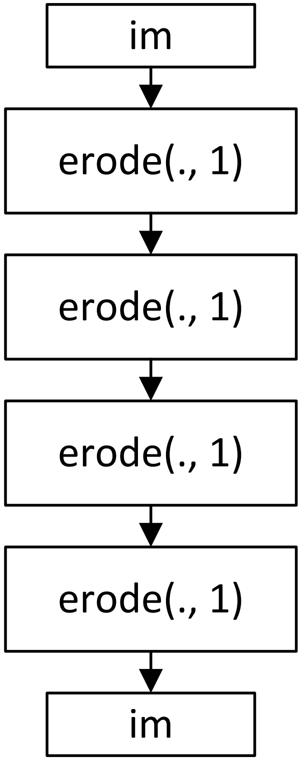
```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    std::vector<uint8_t> fOut = fIn;

    for(int i=0; i<n; i++){
        fIn = fOut;

        tbb::parallel_for( ... );
    }

    return fOut;
}
```

```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    std::vector<uint8_t> fOut = fIn;

    for(int i=0; i<n; i++){
        fIn = fOut;

        tbb::parallel_for( ... );
    }

    return fOut;
}
```

```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    if(n==0){
        return fIn;
    }else{
        std::vector<uint8_t> fOut = fIn;

        tbb::parallel_for( ... );

        return process_frame(
            n-1, w, h,
            fOut
        );
    }
}
```

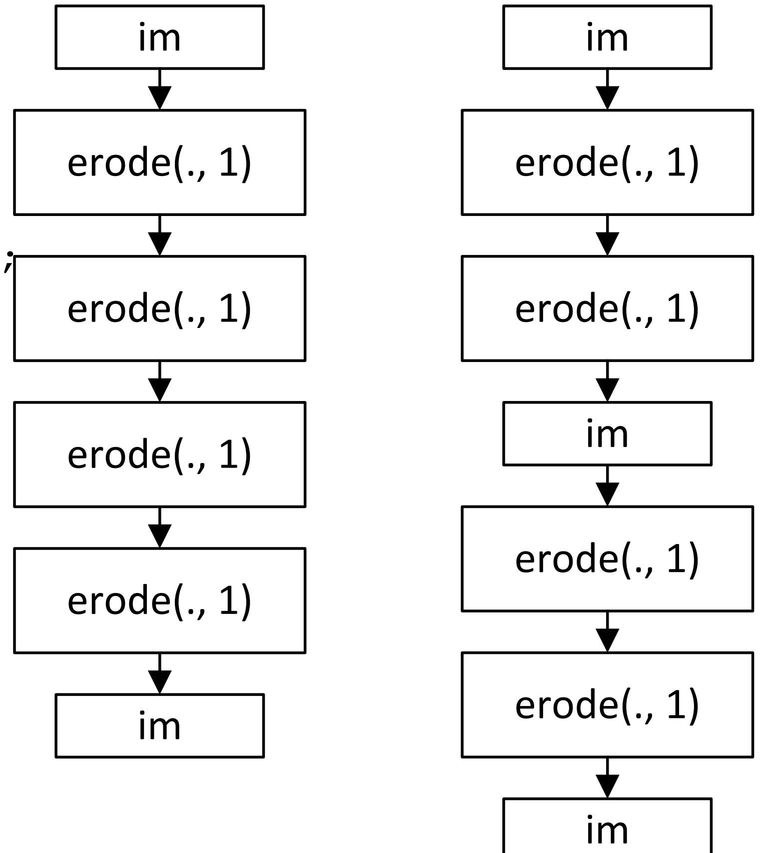
# Repeated function application

```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    if(n==0){
       return fIn;
    }else{
       auto im = erode(fIn, 1);

       return process_frame(
          n-1, w, h,
          im
       );
    }
}
```



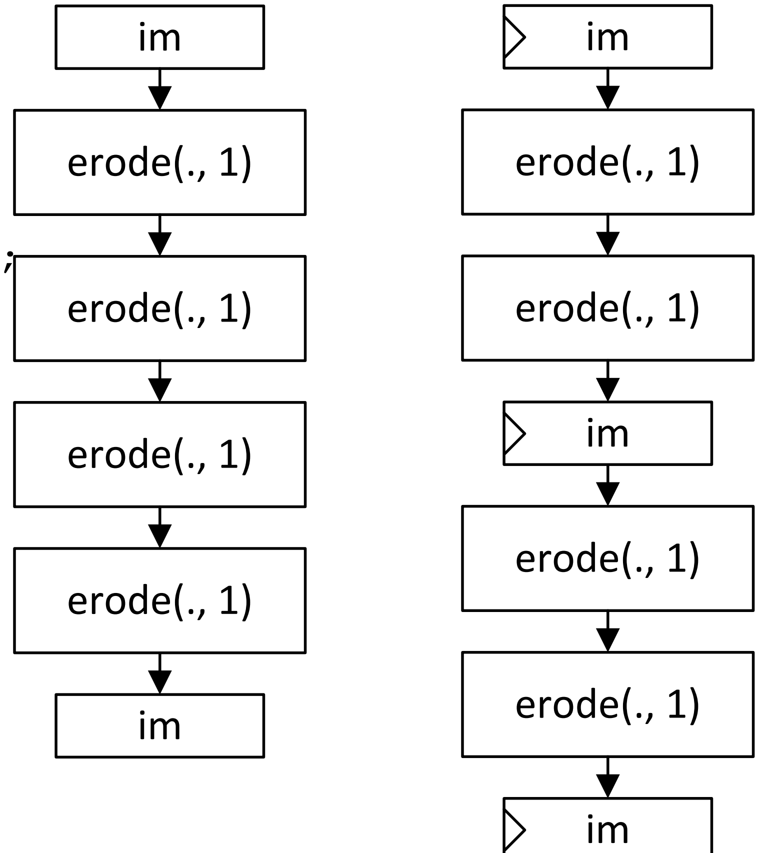
# Can chunk function calls together

```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    if(n==0){
       return fIn;
    }else{
       auto im = erode(erode(fOut1,1),1);

       return process_frame(
          n-2, w, h,
          im
       );
    }
}
```



# Can chunk function calls together

```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn
){
    if(n==0){
       return fIn;
    }else{
       auto im = erode(erode(fOut1,1),1);

       return process_frame(
          n-2, w, h,
          im
       );
    }
}
```



# Adjust solution space for the problem

- Video often has interesting performance requirements
  - Time to process any one frame is usually irrelevant
  - Main performance metric is usually frames/second
  - Latency is not important in many situations – buffer freely
- Need to determine application performance metrics
  - ***Latency***: time from start to end of processing
  - ***Throughput***: average frames per second
  - ***Jitter***: difference between desired and actual time frame shown
  - ***Dropped frames***: tolerance for frames which don't make it
  - ***Distortion***: acceptable pixel-level errors within each frame
- If we are allowed some latency, pipelining is possible

# Pipeline parallelism

- **Problem**: want to calculate $y_i=f_1(f_2(\ldots(f_n(x_i)\ldots))$, $i=1,2,\ldots$
- **Goal**: high throughput only, maximise outputs / sec
- **Solution**:
  - Multiple tasks each handling one function in parallel
  - Synchronise all tasks at the end of each round
- **Requirements**:
  - $f_1..f_n$ are side-effect free, so can safely call them in parallel
  - Application is latency tolerant
  - Intermediate memory usage is not a problem

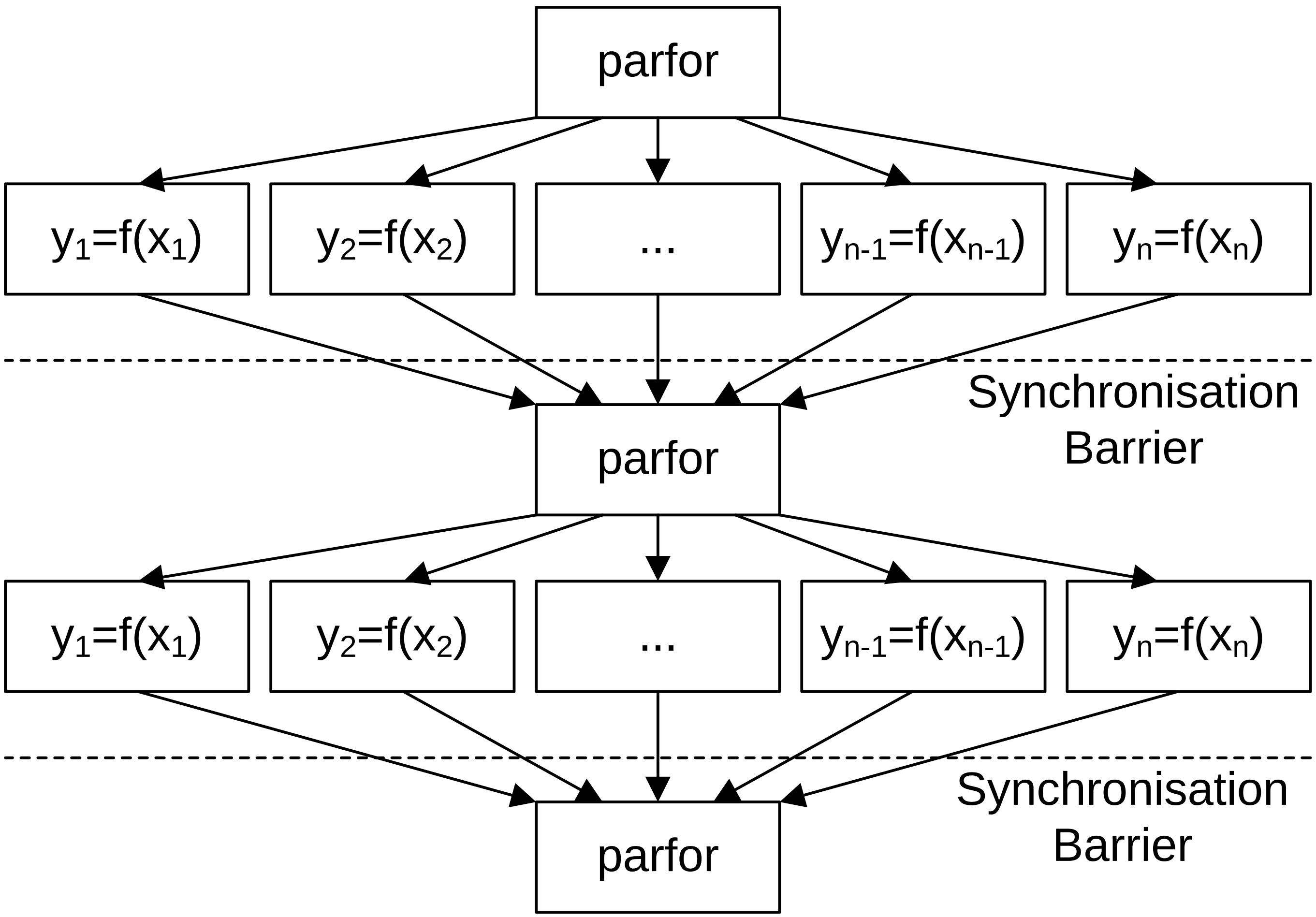
# Data parallelism

- **Problem**: want to calculate vector $y_i=f(x_i)$, $1 \le i \le n$
- **Goal**: low latency, minimise total execution time
- **Solution**:
  - Multiple tasks each handling one piece of data in parallel
  - Synchronise all tasks at the end of each round
- **Requirements**:
  - f is side-effect free, so can safely call them in parallel

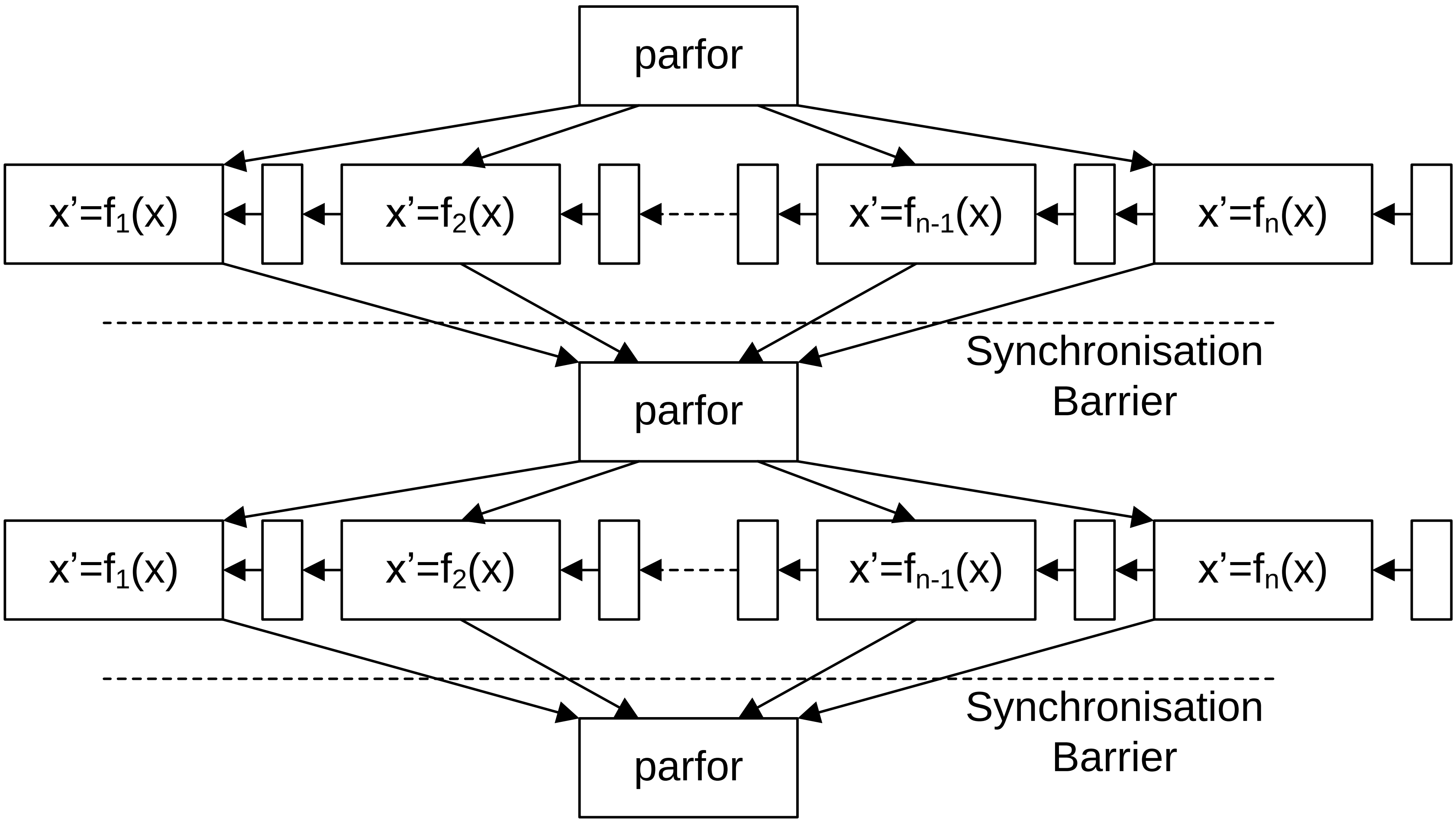
# Our first two design patterns

- Data-parallelism : *apply one function to lots of data*
  - Simple but powerful form of parallelism, natural in HW and SW
  - Widely applicable, many applications allow SIMD operation
  - Can be used to build some other types of parallelism
- Pipeline parallelism : *apply lots of functions to a stream*
  - Often well supported in hardware; less so in software
  - Fewer applications, as must be able to tolerate latency
  - Difficult to use as a primitive for other forms of parallelism
- Both are restricted in scope
  - Must know amount of data / number of functions at start-up
  - Very simple dependency model based on barriers
  - *Future design patterns: relax these restrictions*

# Control dependency view : data par.



# Control dependency view: simple pipeline



# Practical pipelining

- tbb::parallel_for can be used to construct pipelines
  - Not what it is designed for
  - Abusing one design pattern to implement another

- Many libraries and approaches support it natively
  - Unix Pipes: one of the simplest general purpose tools
  - Threaded Building Blocks: allows complex pipelines
  - OpenCL 1.2: use events to build pipelines
  - OpenCL 2.0: builtin support for FIFOs between kernels
    - ~~Too new for us to look at~~ NVidia still don’t support it
    - Designed to allow hardware-level pipeline parallelism
  - Lots of video and audio-processing streaming APIs

# Unix pipes as pipeline parallelism

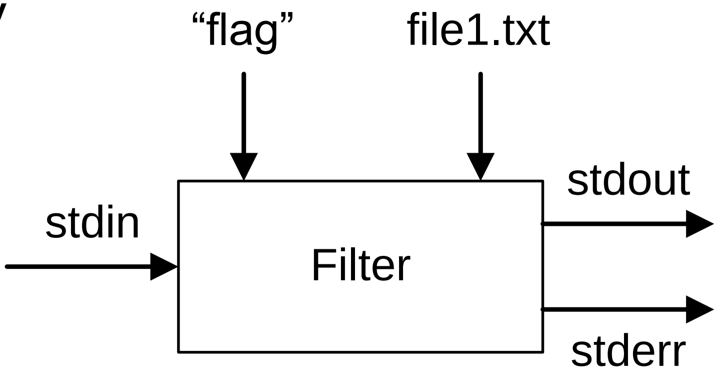
Mike Gancarz: "*The UNIX Philosophy*":

1. Small is beautiful.
2. Make each program do one thing well.
3. Build a prototype as soon as possible.
4. Choose portability over efficiency.
5. Store data in flat text files.
6. Use software leverage to your advantage.
7. Use shell scripts to increase leverage and portability.
8. Avoid captive user interfaces.
9. Make every program a filter.

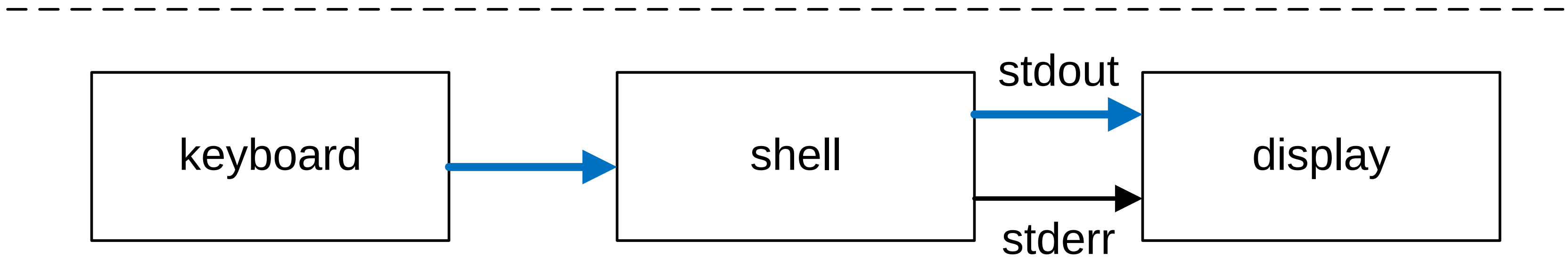
There are multiple re-statements, but this one I prefer.

# Anatomy of a filter program

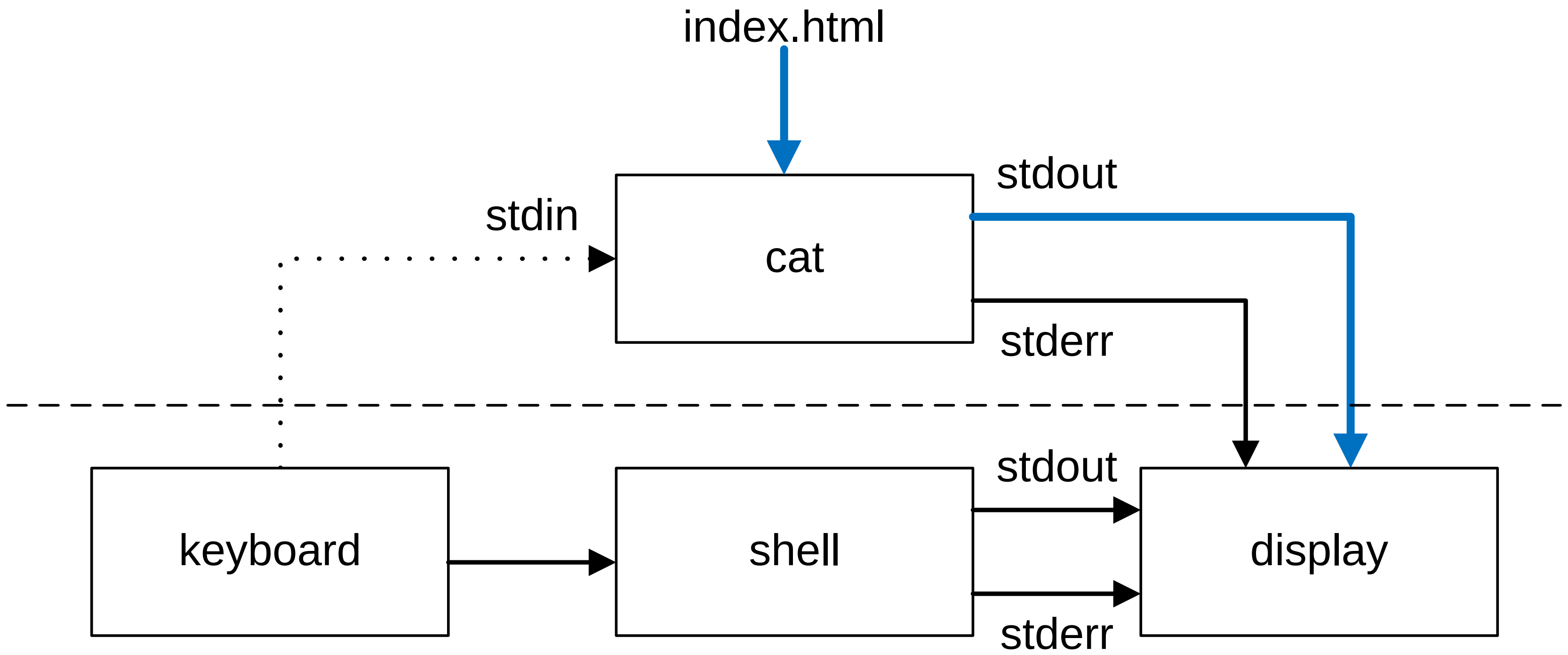
- Most OS's and languages have standard streams
  - stdin : Input text or binary data being passed to the program
  - sdout : Output text or binary data being produced by program
  - stderr : Diagnostic information produced during execution
- Streams are initialised when program starts
  - Arguments are passed to main by shell or OS
  - Standard streams are automatically connected, e.g. to keyboard/display
  - Program has to deal with the extra arguments, may open files, ...



# Stream re-routing

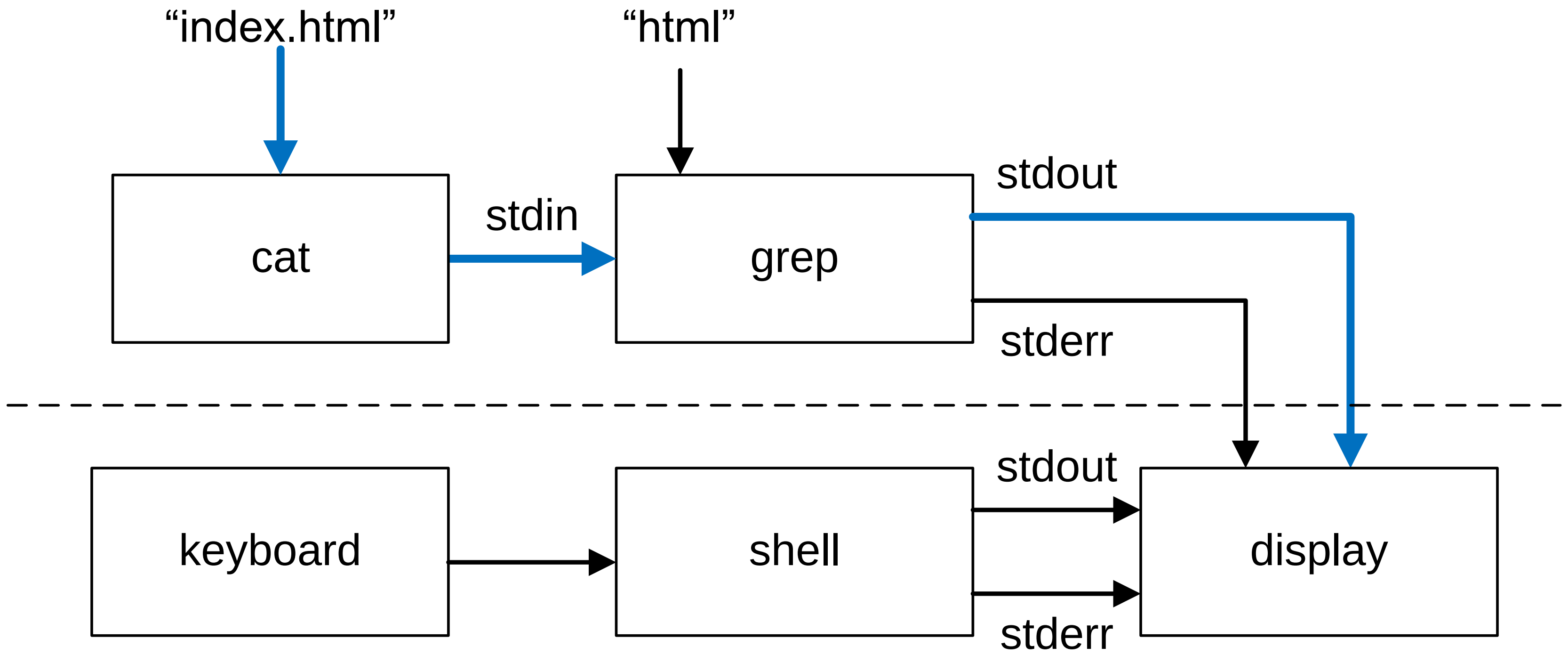


# Stream re-routing



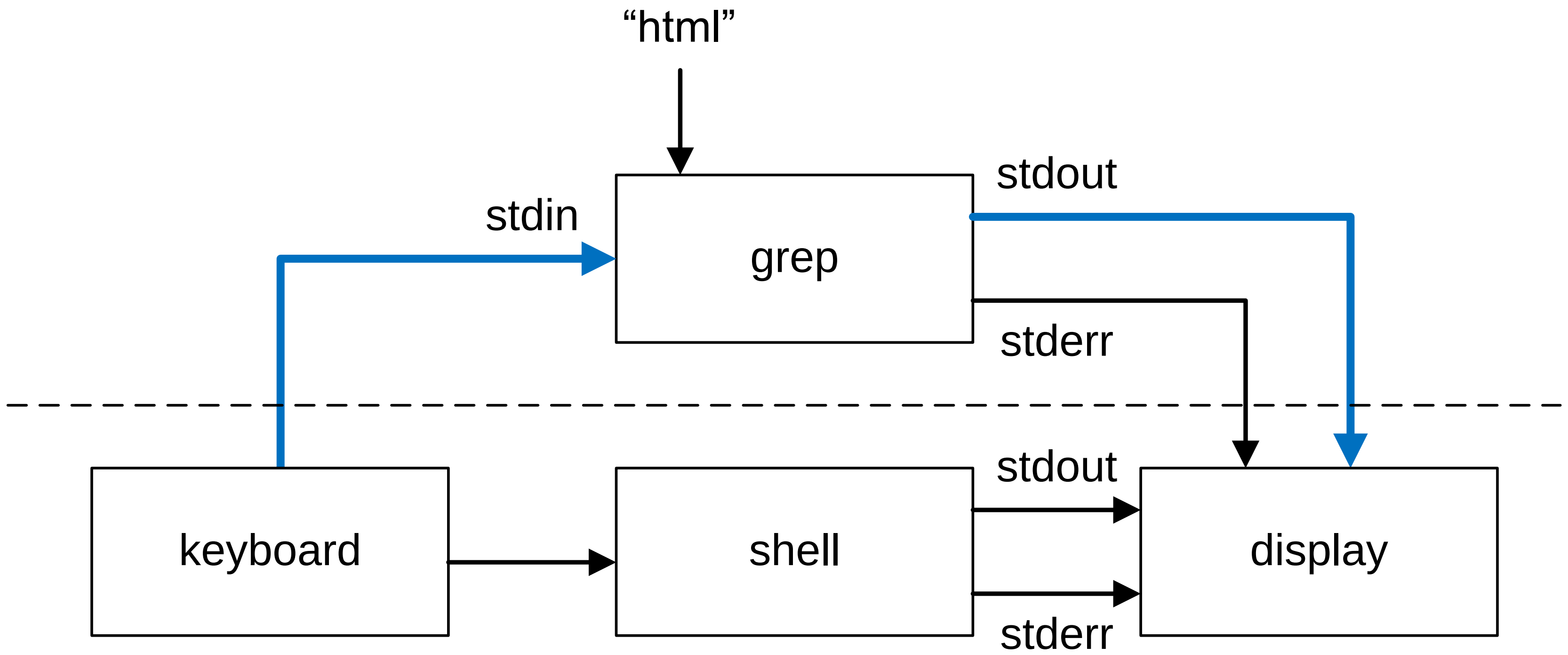

`cat index.html`

# Stream re-routing



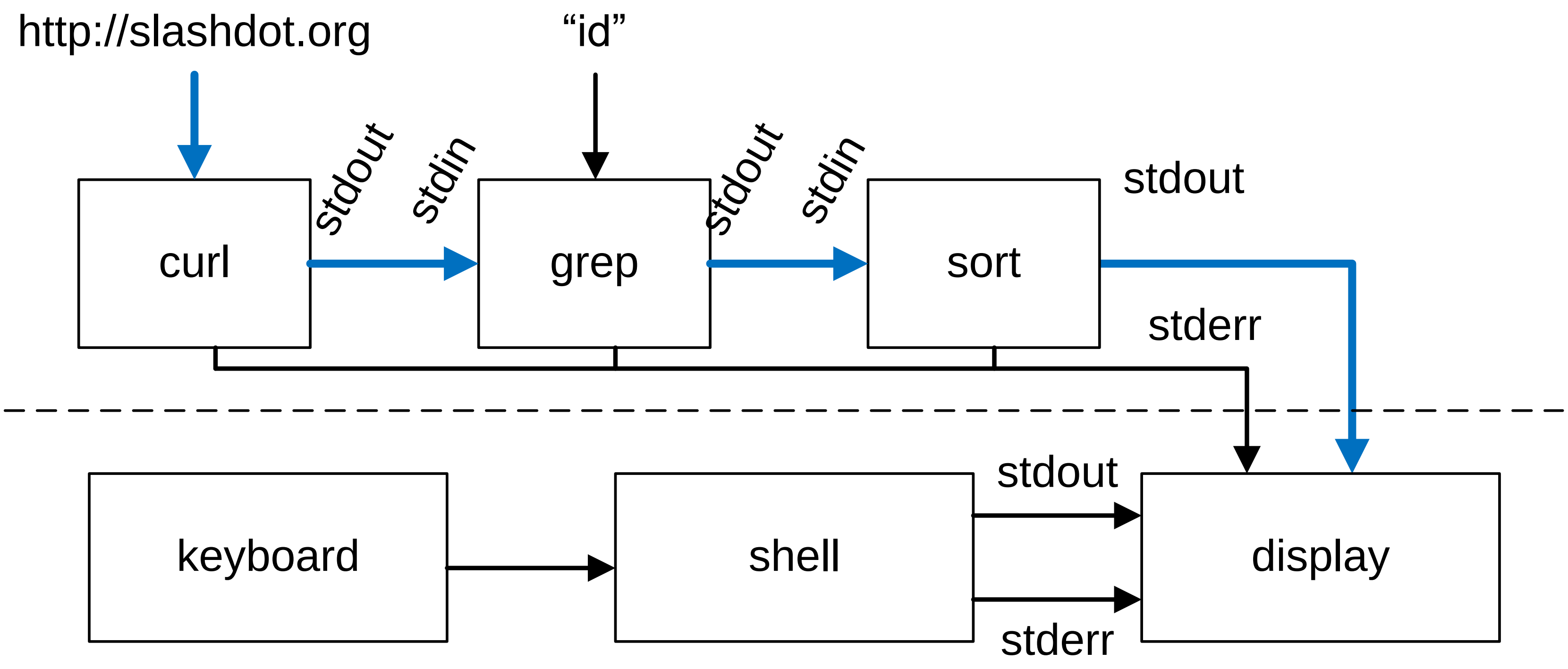

`cat index.html | grep html`

# Stream re-routing




`grep html`

# Stream re-routing




```
curl http://slashdot.org | grep id | sort
```

# Advantages of streaming data

- Intermediate data never has to touch disk
  - IO is expensive; we are often limited by disk bandwidth
  - When processing terabytes of data there is not enough space
    - For “Big-Data” can often only store compressed version
    - High performance computing is increasingly data-limited
- Parallel processing comes for free
  - Each stage in the pipeline is its own parallel process
  - OS will block processes when they are waiting for data
- Synchronisation is local, rather than global for pipeline
  - Block when there is not enough data on stdin
  - Block when there is not enough buffer space on stdout
  - Apart from that: *process away!*

# Disadvantages?

- Limited to linear chains?
  - ***No***; can create merge then split – works very well.

- Can't create cyclic graphs?
  - ***Yes***; need to worry about loop carried dependencies
  - Can sometimes get round it in hardware with C-Slow

- No reconvergent graphs? (i.e. split then merge)
  - ***Somewhat***; danger of deadlock, unless some conditions are met

- High communication to compute ratio?
  - ***Somewhat***; large communication overhead with small tasks

# Merge operations

- Take n distinct streams and merge to a single stream
  - Compositing video streams
  - Mixing audio streams
  - Correlating event data streams
  - Merge together two streams of data

- Interleave columns of two csv files
  - `pr -m -t -s\ data1.csv data2.csv |`
    `gawk '{print $1, $4, $2, $5, $3, $6}'`

# Fork operations

- Take the same stream and duplicate to many sinks
  - Generate expensive source stream once, process in parallel
  - Take a live stream and pass to multiple consumers

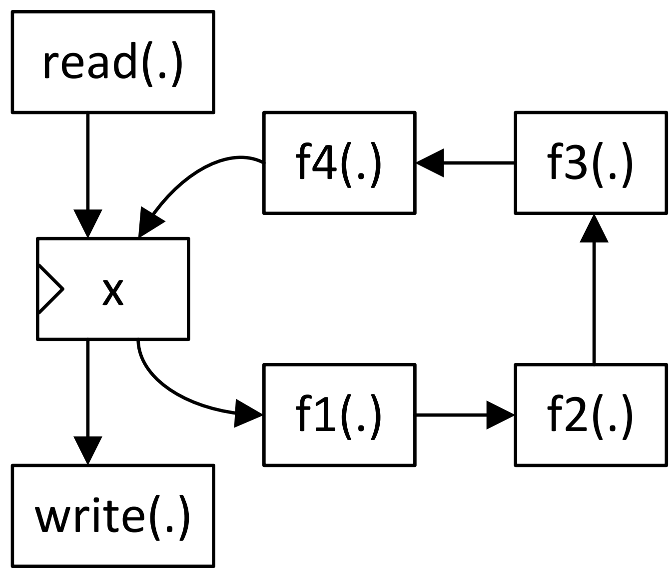
- Supported in shell via ‘tee’ and process substitution
  - e.g. Compress file to multiple types of archive

```
cat /dev/random | tee >(gzip  -9 > rnd.gz) \
                | tee >(bzip2 -9 > rnd.bz) \
                | tee >(lzip  -9 > rng.lz) \
                | tee >(lzop  -9 > rng.lzop) \
                | tee >(zip   -9 > rnd.zip) \
                > /dev/null
```

# Cyclic graphs

- There is not a lot we can do with cyclic graphs
  - Same sort of loop carried dependencies we saw before
- If we have multiple streams we might be able to C-Slow
  - Add “C” pipeline registers, and process “C” separate streams

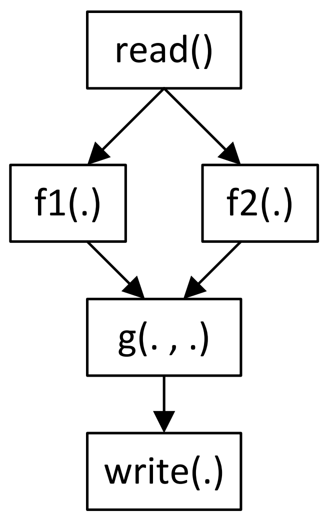
```
x=read();
for i=1:n
  x=f1(x);
  x=f2(x);
  x=f3(x);
  x=f4(x);
end
write(x);
```



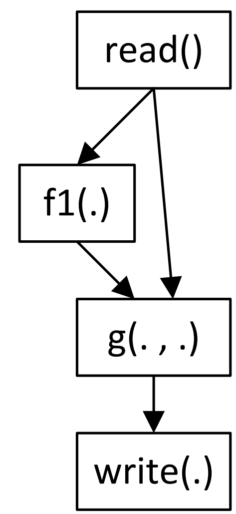
# Reconvergent graphs

- Very common to split then merge from a single source
  - *Image processing*: apply horizontal and vertical filters
  - *Audio processing*: apply filter then check distortion
- It can definitely be done (in unix shell, elsewhere)
  - We can construct it using FIFOs

```
while(1){
  x=read();

  t1=f1(x);
  t2=f2(x)
  z=g(t1,t2);

  write(z);
}
```



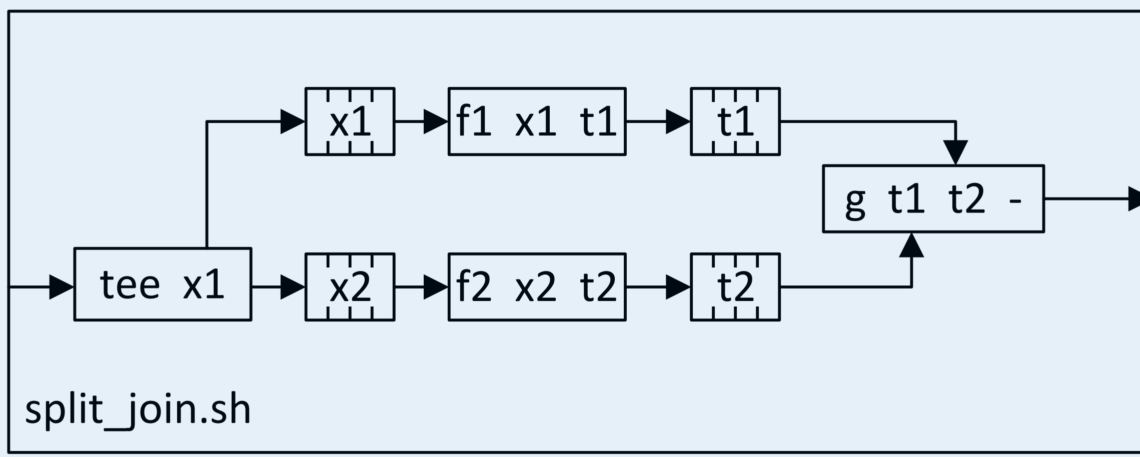

```
while(1){
  x=read();

  t1=f1(x);
  z=g(t1,x);

  write(z);
}
```



# Reconvergent graphs using pipes

- We can express arbitrary DAGs using programs & pipes
  - DAG = Directed ***Acyclic*** Graph
- Not always appropriate, but useful in special cases

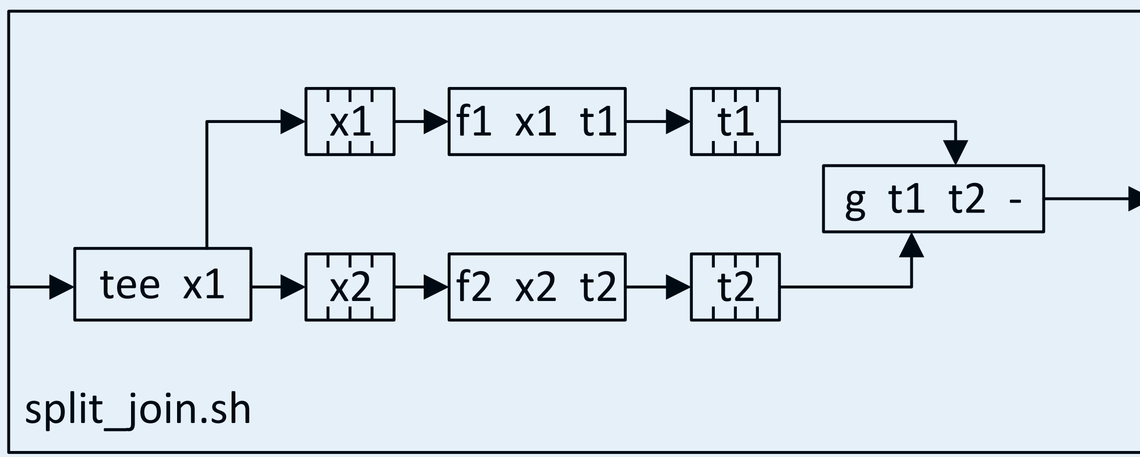
```
while(1){
  x=read();
  t1=f1(x);
  t2=f2(x)
  z=g(t1,t2);
  write(z);
}
```



# Reconvergent graphs using pipes

- Can capture the graph in a script
  - Like creating a function, except arguments are streaming
- Allows ***composition*** of parallel components

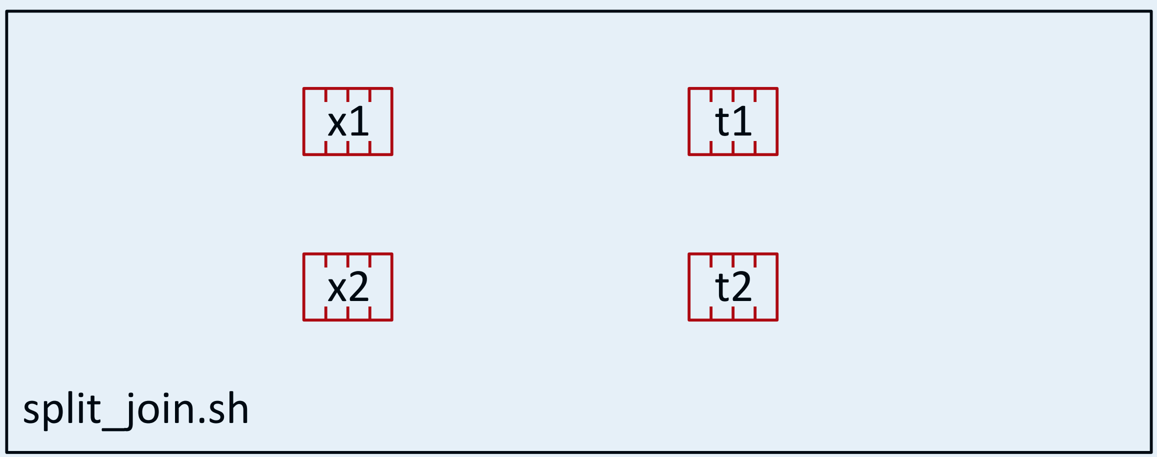
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 1 – Create FIFOs

- Need to be able to name the intermediate states
- FIFOs act a bit like variables
  - One process can write to FIFO, another can write
  - FIFOs have a fixed buffer size (operating system dep. ~4K-64K)

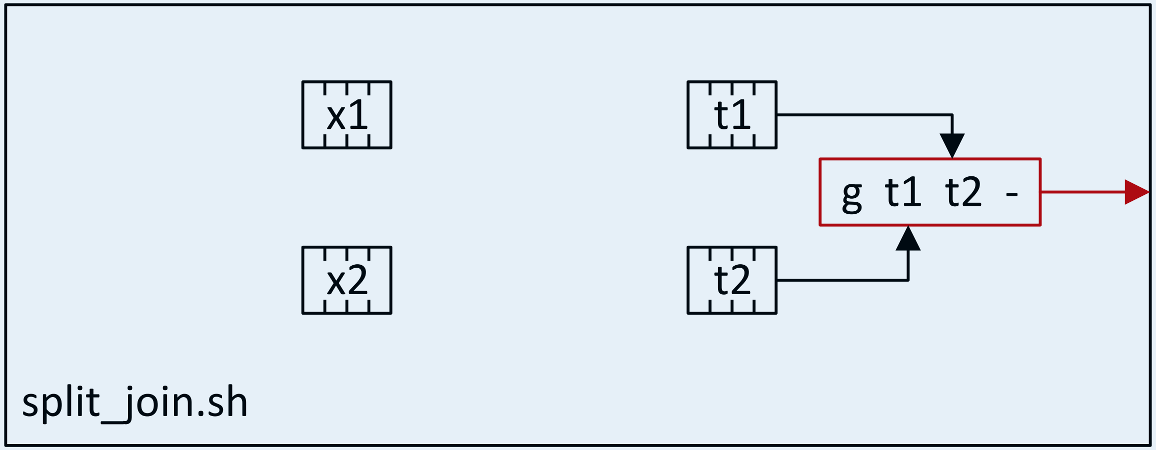
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 2 – Build backwards from the output

- Ampersand (&) means to launch task in parallel
- Task inherits the stdout of the parent (i.e. the script)
- No writers on t1 and t2, so it blocks

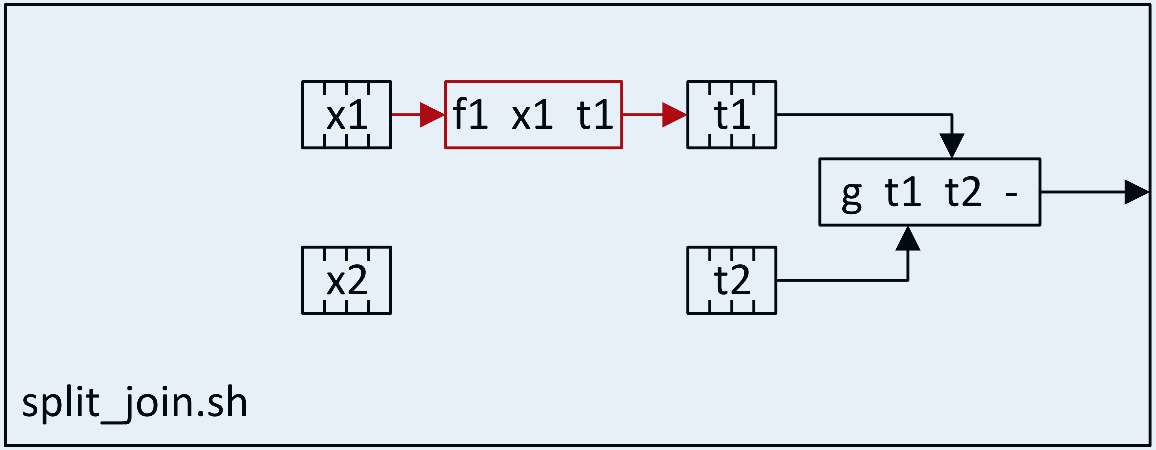
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 3 – Connect filters

- Add more parallel tasks, building backwards
- Tasks will stay blocked, as there is no input yet

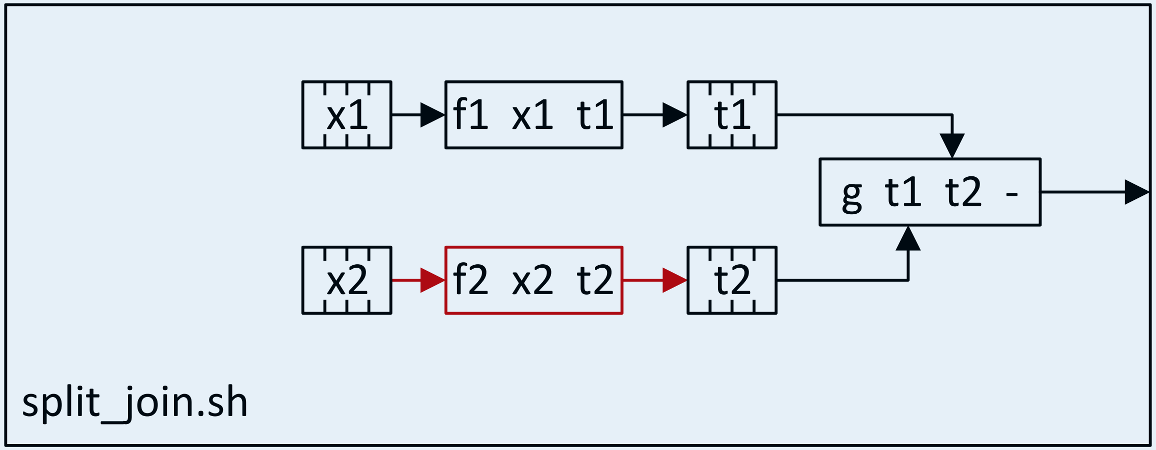
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 3 – Connect filters

- Add more parallel tasks, building backwards
- Tasks will stay blocked, as there is no input yet

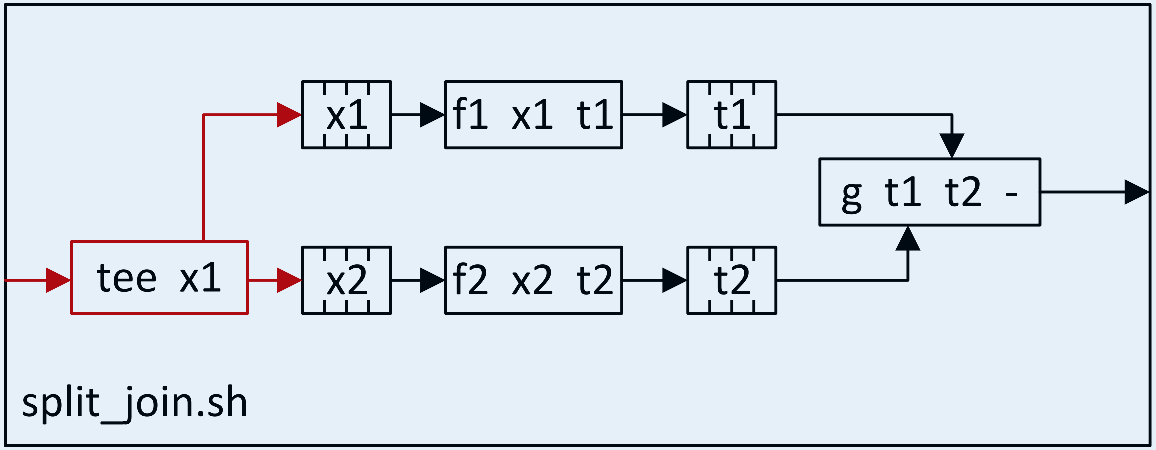
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 4 – Split input to each branch

- `tee` is a simple program that duplicates stdin
  - One copy is simply copied to stdout as normal
  - One or more copies are sent to named files
- Can use it to send the stream to two different FIFOs

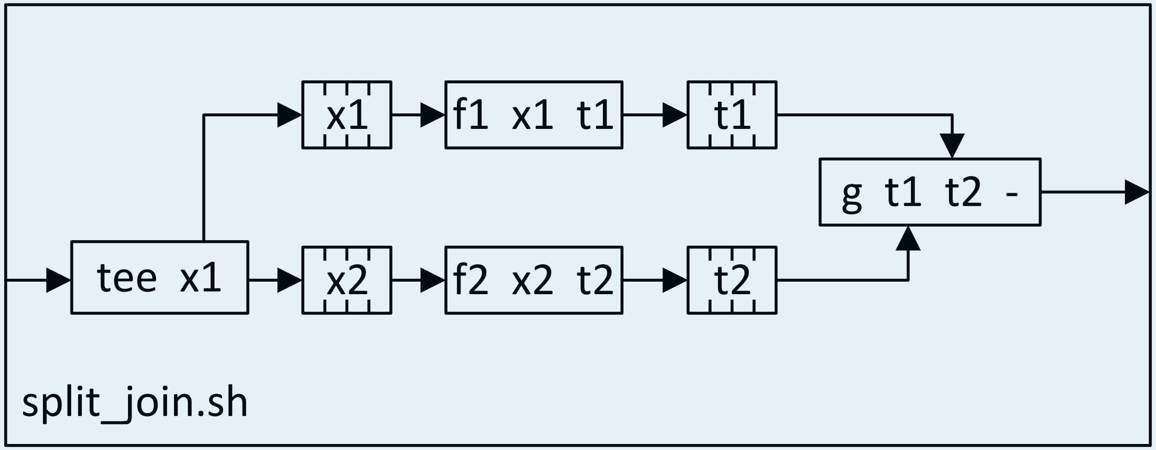
```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# 5 – Connect the inputs & outputs

- Graph within script is now complete, but blocked
  - Anyone trying to read stdout will also block
- Once stdin produces data, graph will start running

```
#!/bin/sh

mkfifo  x1 x2  t1  t2

g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



# Pipe streams are simple but powerful

- If you can draw the graph, you can build it
  - Parallelism is automatic and easy
  - OS will schedule multiple processes on fewer CPUs
- Makes it very easy to avoid touching disk
  - Can work with terabytes of data very easily
  - Can decompress and compress data on the fly
    - Can get ~500 MB/sec off SSD (compressed): ~ 2 TB / hour
- Ideal for progressive filtering of data
  - *Initial filter*: very fast, eliminate 90% of data at 200+ MB/sec
  - *Next filter*: more accurate, remove next 90% at 20 MB/sec
  - *Accumulation*: accurately detect items of interest, collect stats.
  - e.g. : search for string; then apply regex; then use parser

# *Disclaimer* : 1TB is not “Big Data”

- 1 TB data sets are routine – you just get on with it
  - e.g. 1 day of cross-market intra-day tick-level data is 100 GB+
  - Raw wikipedia is 40GB
- Big data (in the volume sense) is in the PB range
- Also have to worry about Velocity and Variety

Don’t tell people I said pipes were the solution to big data

# Problems with pipes

- Communications overhead
- Scheduling overhead
- Potential for deadlock
- Raw streams not a good data-type for many applications

# Communications overhead

- Every transfer potentially involves some IPC
  - Data has to move from one address space to the other
  - *An advantage when you want to move data between machines*
- Need frequent calls to the Operating System

```
uint8_t buffer[SIZE];
while(1){
  read(in, SIZE, buffer);
  // do something to buffer
  write(out, SIZE, buffer);
}
```

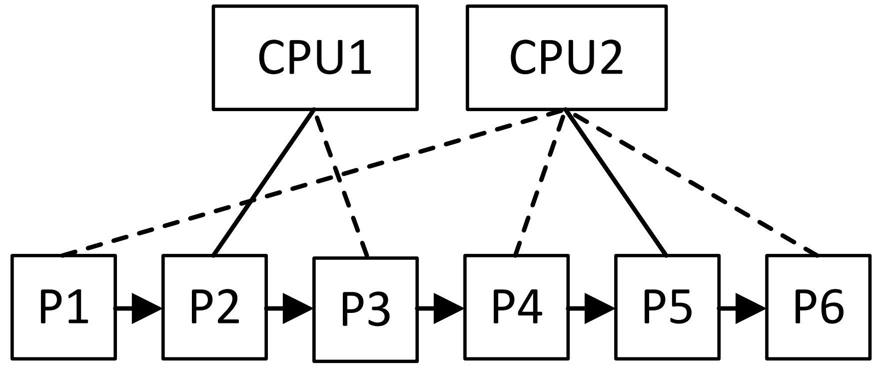
IPC

```
uint8_t buffer[SIZE];
while(1){
  read(in, SIZE, buffer);
  // do something to buffer
  write(out, SIZE, buffer);
}
```

IPC = Interprocedural Communication

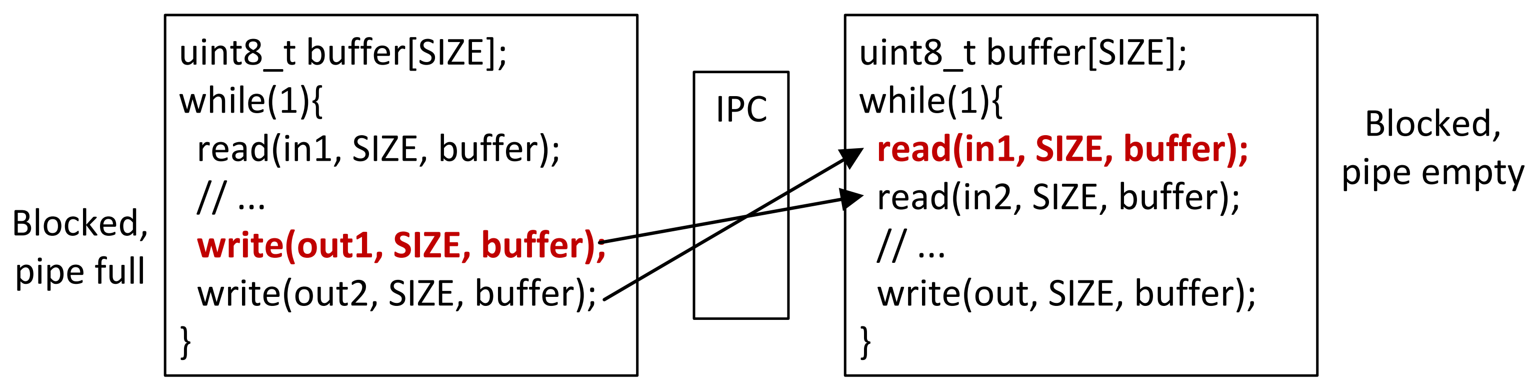
# Scheduling overhead

- Bigger pipelines have more processes for OS to schedule
  - May have many more active processes than CPUS
  - Lots of losses due to context-switches when all filters active
  - May be frequent blocking due to congestion
- Time-slicing is useful, but inefficient with 100s of tasks



# Potential for deadlock

- OS provided pipes have a fixed-size buffer
  - Typically a few tens of KB, e.g. 64K in modern linux
  - Large writes may block till consumer has drained buffer
  - Large reads may block till produced has filled buffer
  - ***Or***: may read/write less than the entire buffer's worth
- Reconvergent pipelines can be tricky
  - Filters need to be very well behaved and make few assumptions



# Raw binary streams are too low-level

- Lot's of data-types are naturally packets
  - Have a header, properties, payload, variable size...
  - e.g. video frames, sparse matrices, text fragments, ...
- Some data-types are very expensive to marshal
  - Passing graphs down a pipe is slow and painful

```
uint32_t width, height;

while(1){
  read(in, 4, &width);
  read(in, 4, &height);
  pixels=realloc(pixels, width*height);
  read(in, width*height, pixels);

  // ...
}
```