Revisiting video

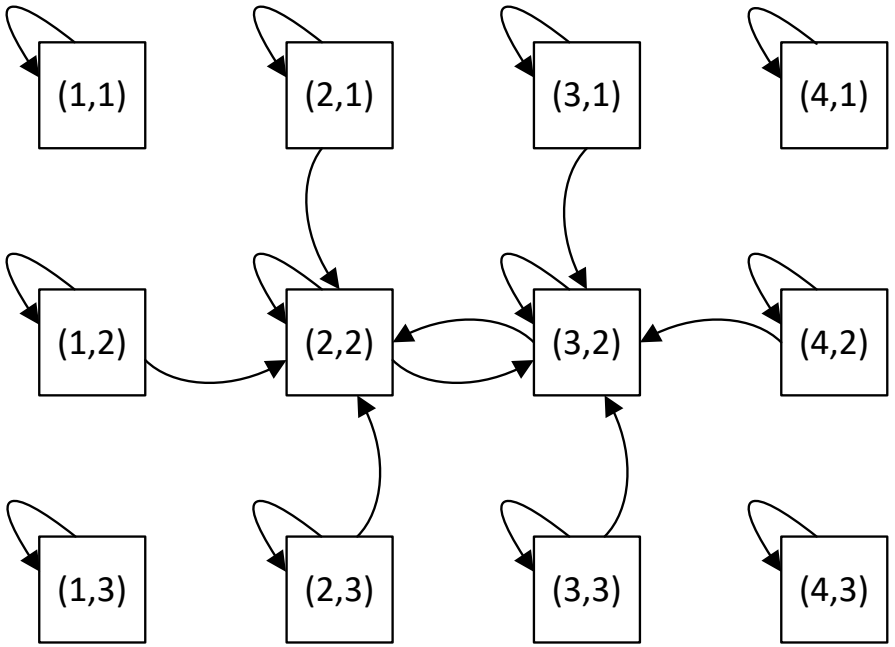
```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn // Receive frame (by value)
){
    // Handle n==0 case
    std::vector<uint8_t> fOut=fIn;

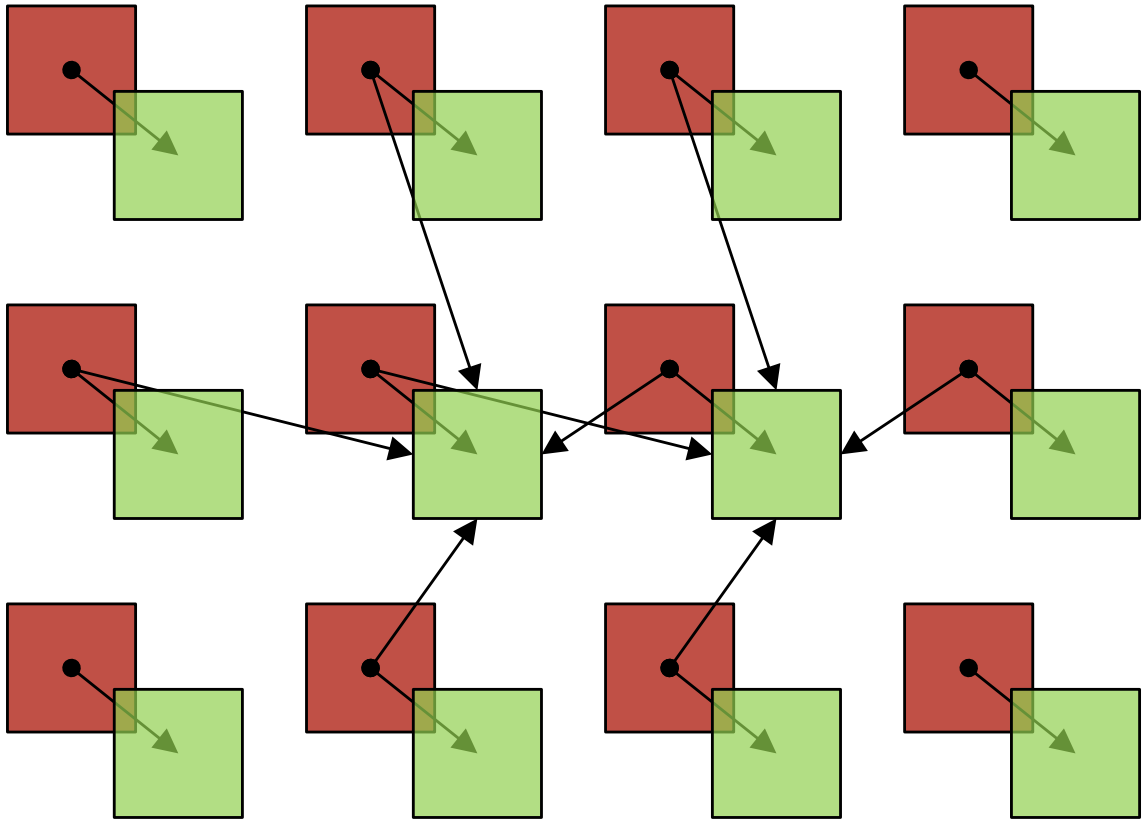
    for(int i=1; i<n; i++){
        fIn = fOut;

        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
){
    if(n==0){
        return fIn;
    }else{
        std::vector<uint8_t> fOut = fIn;

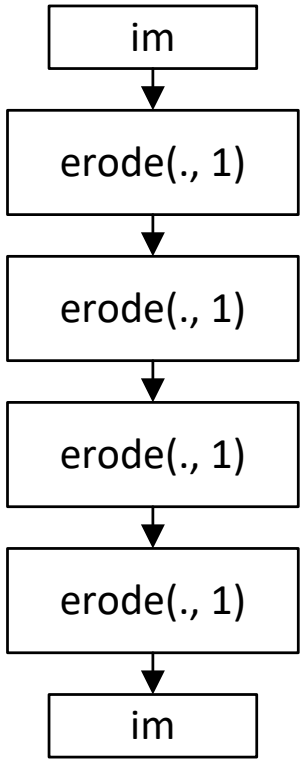
        tbb::parallel_for( ... );

        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);

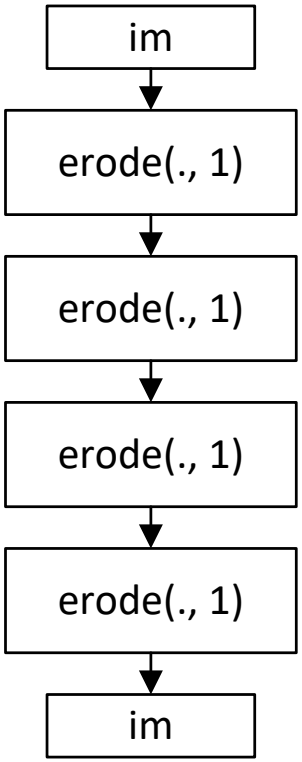
       return process_frame(
          n-1, w, h,
          im
       );
    }
}
```

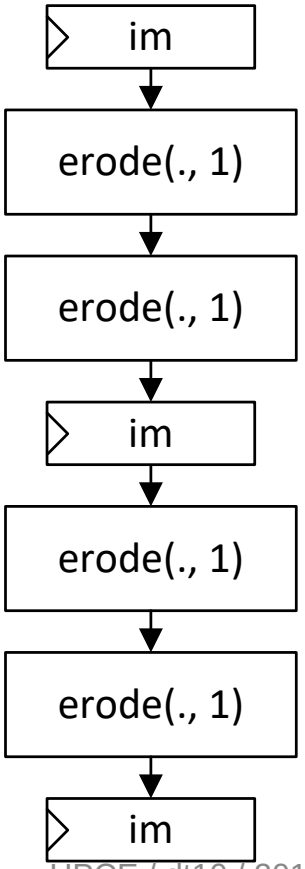


Can chunk function calls together

```
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(fIn,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 $y_i=f_1(f_2(...(f_n(x_i)...))$, $i=1,2,...$
- **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

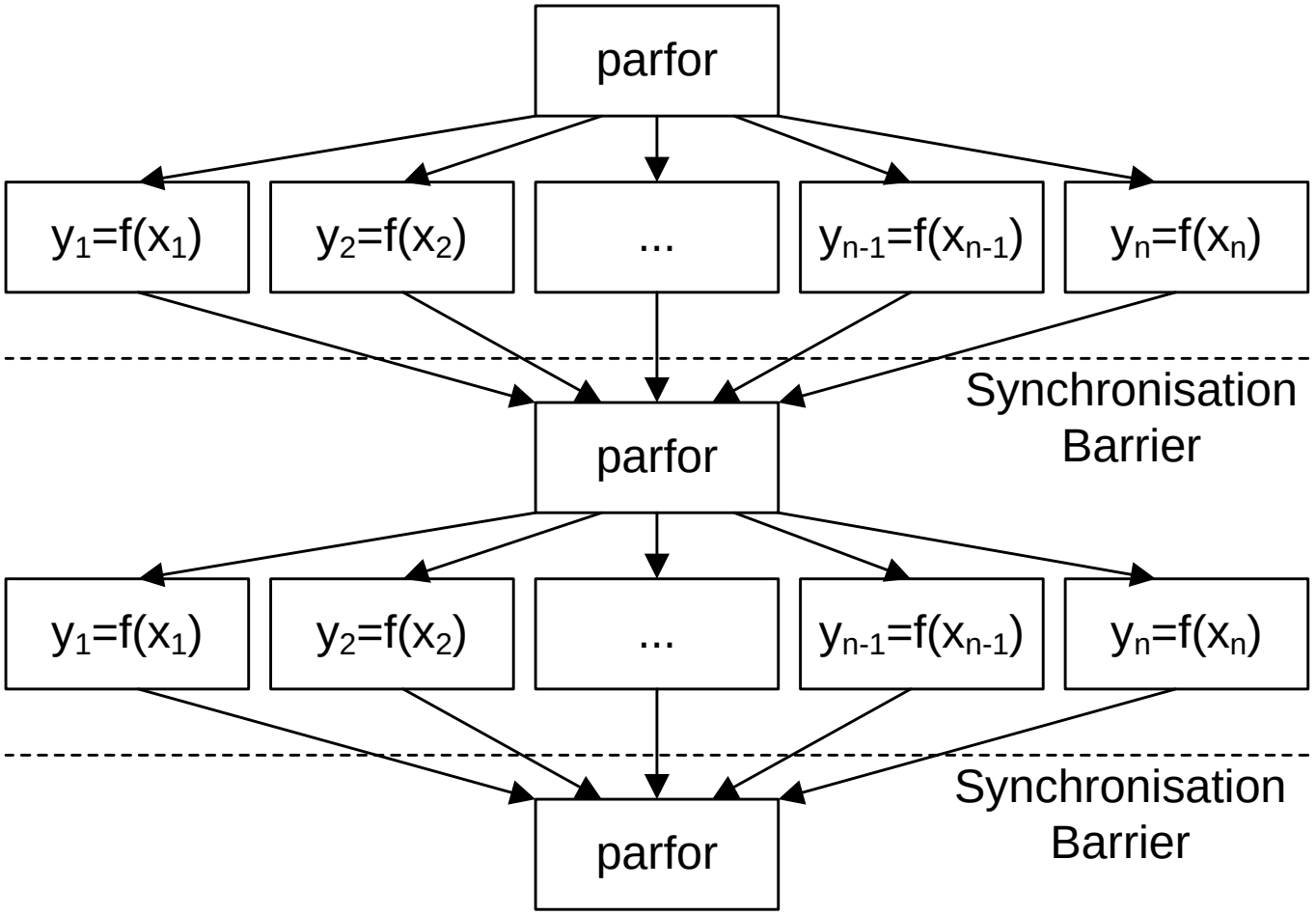
Data parallelism

- **Problem**: want to calculate vector $y_i=f(x_i)$, $1 \leq i \leq n$
- **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

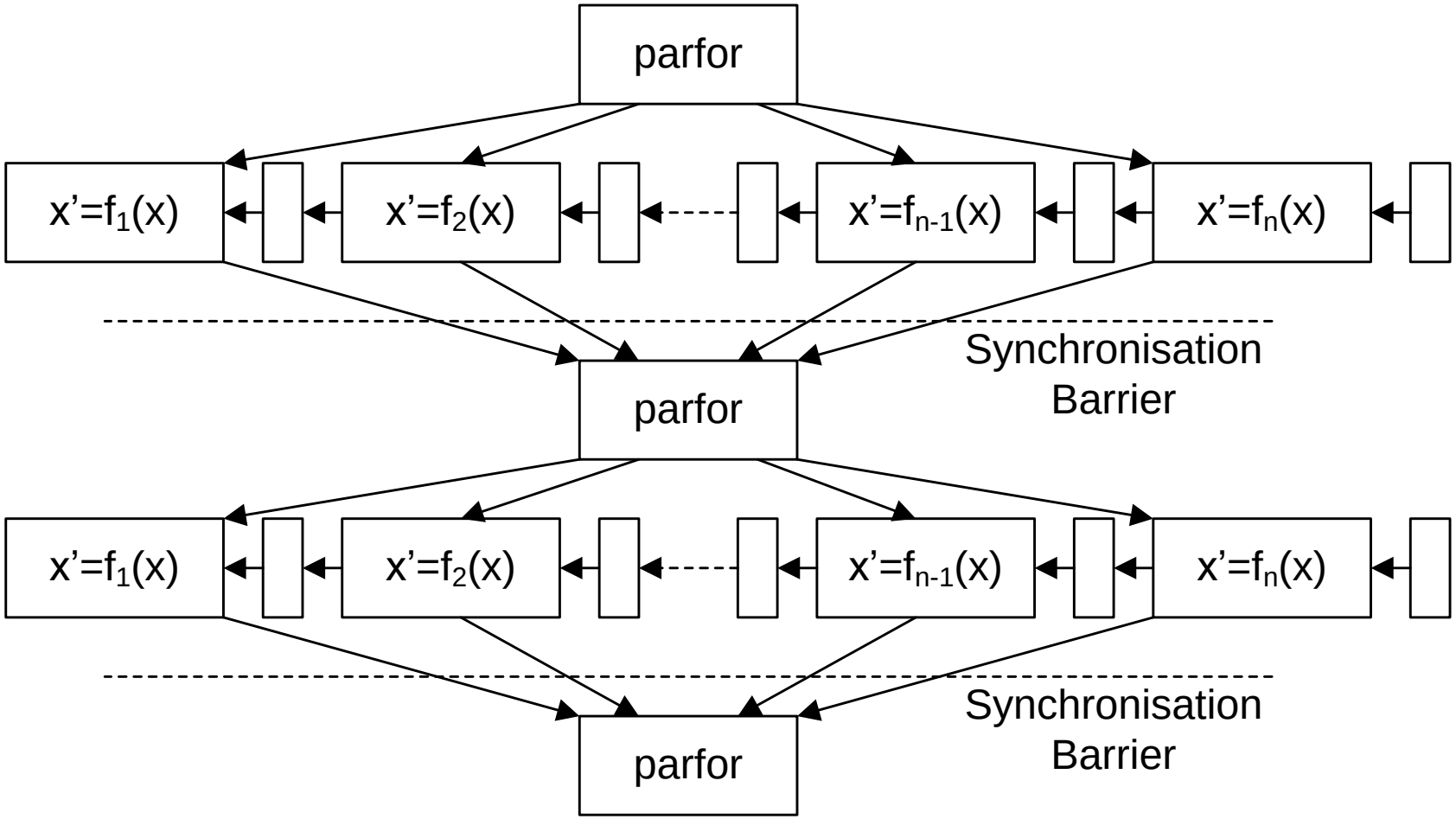
Our first two design patterns

- 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

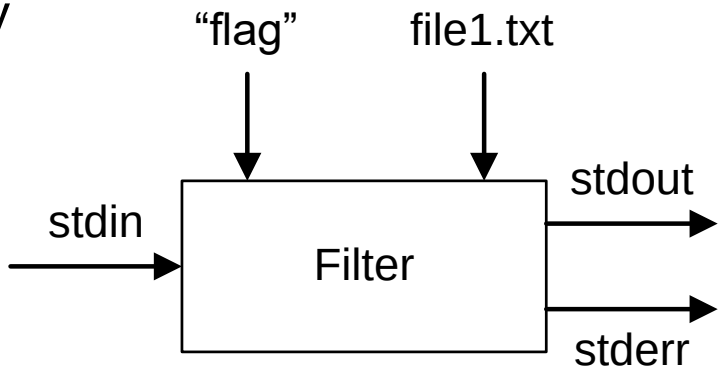
Mike Gancarz: "*The UNIX Philosophy":*

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.

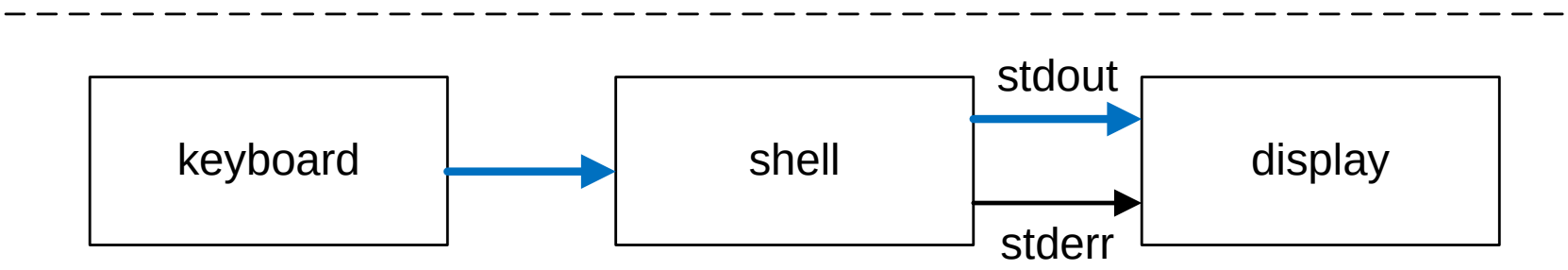
There are multiple re-statements, but this one I prefer.

Anatomy of a filter program

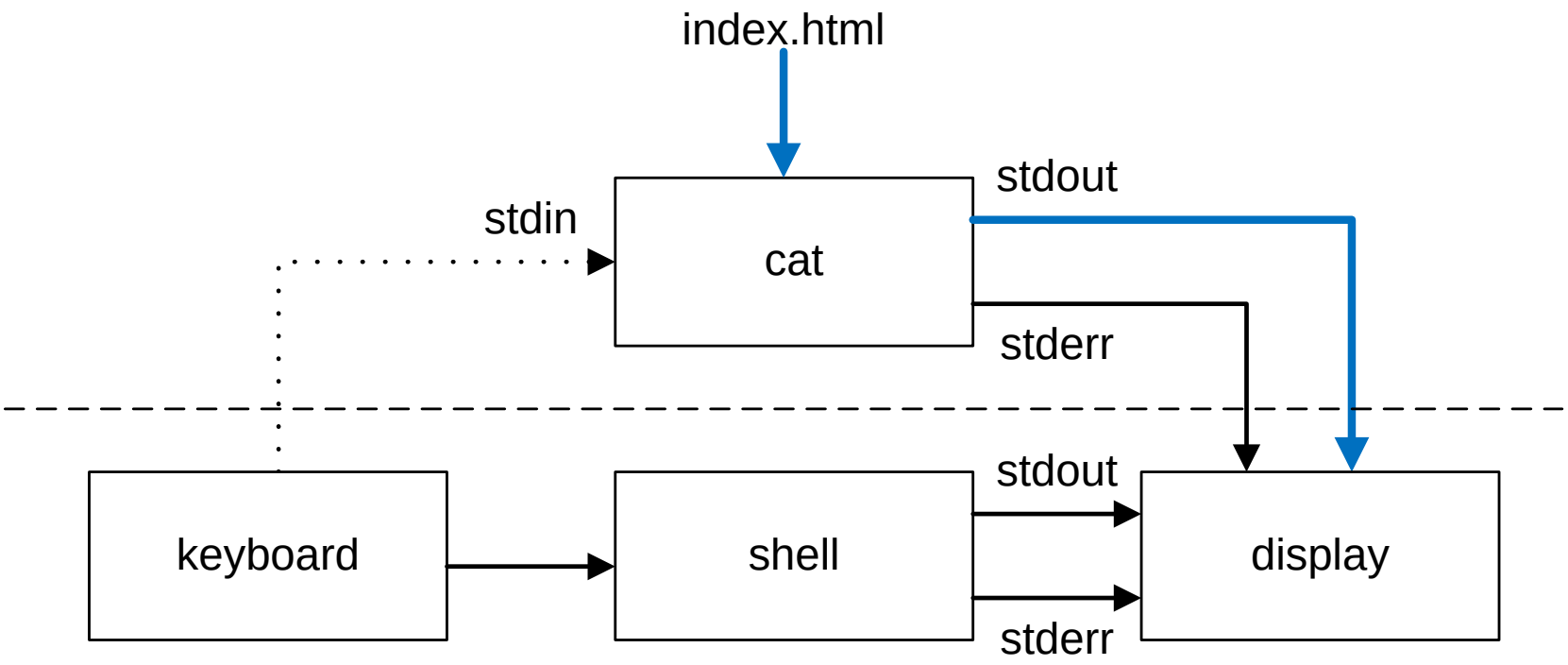
- Most OS's and languages have standard streams
  - stdin : Input text or binary data being passed to the program
  - sdout : Output text or binary data being produced by program
  - stderr : Diagnostic information produced during execution
- Streams are initialised when program starts
  - Arguments 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

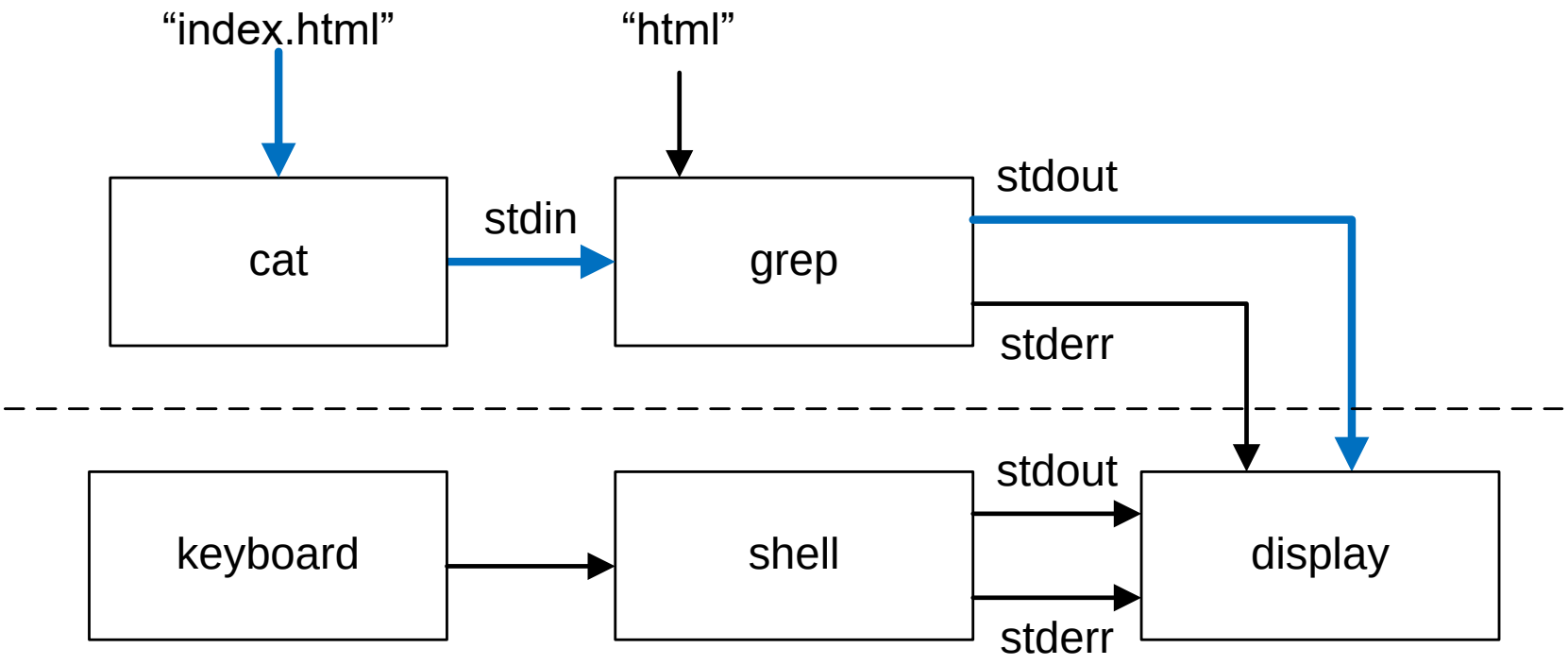


Stream re-routing



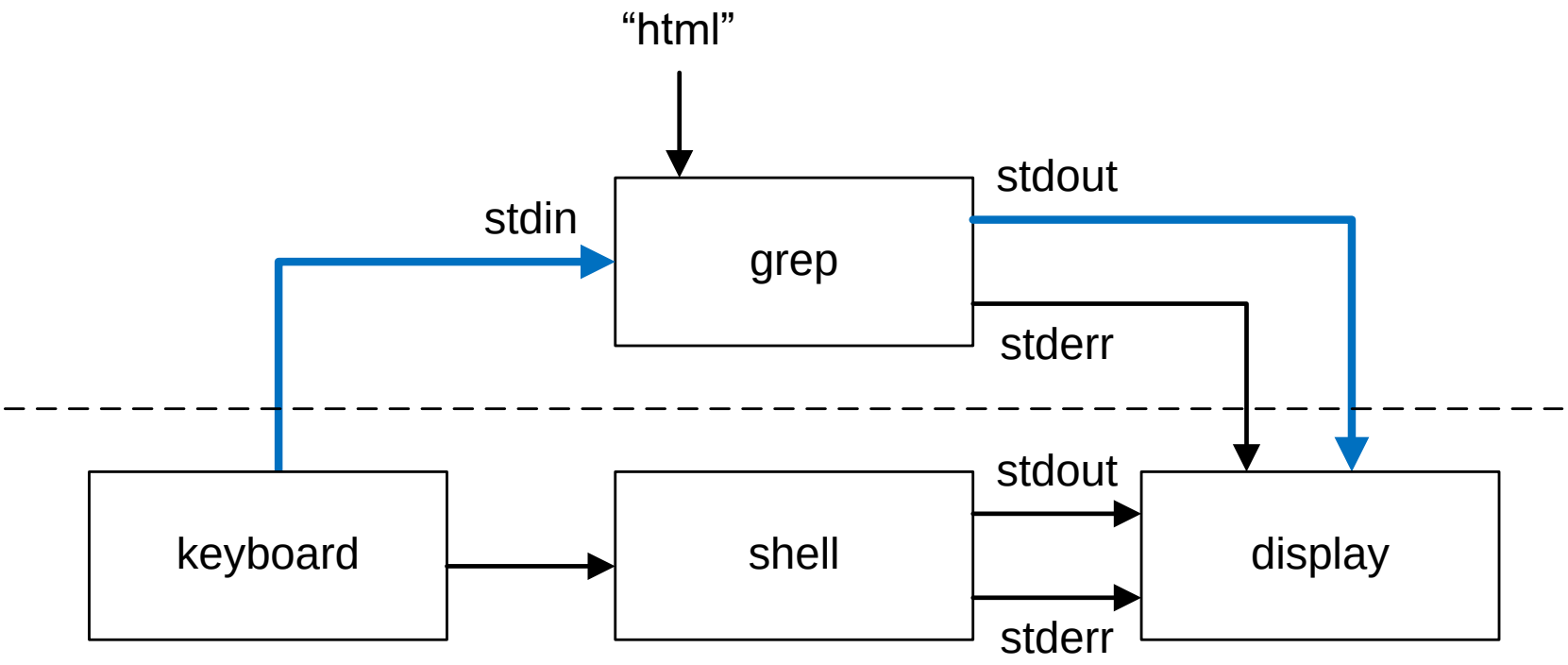

```
cat index.html
```

Stream re-routing



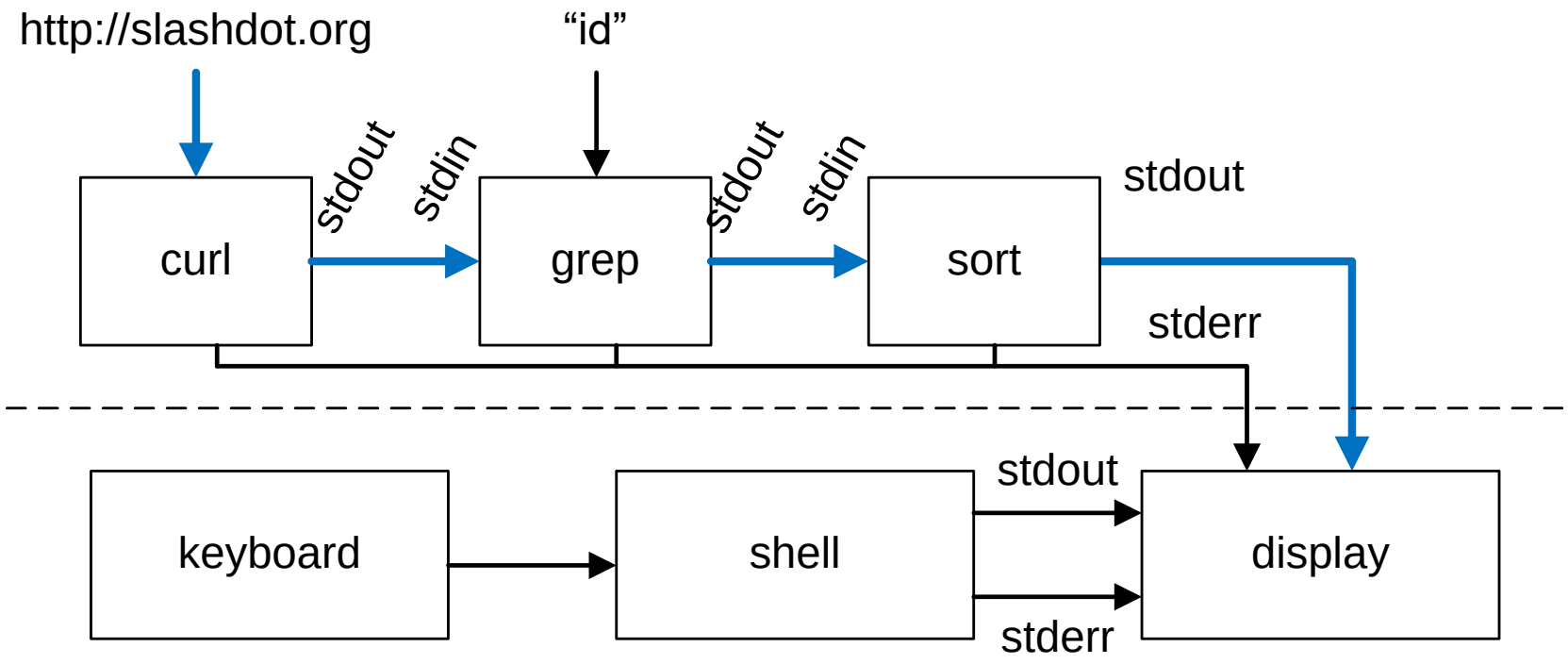

```
cat index.html | grep html
```

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?

- Can merge or split data streams?
  - ***Yes***; works very well.
- Can create cyclic graphs (loops)?
  - ***No***; need to worry about loop carried dependencies
  - Can sometimes get round it in hardware with C-Slow
- 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

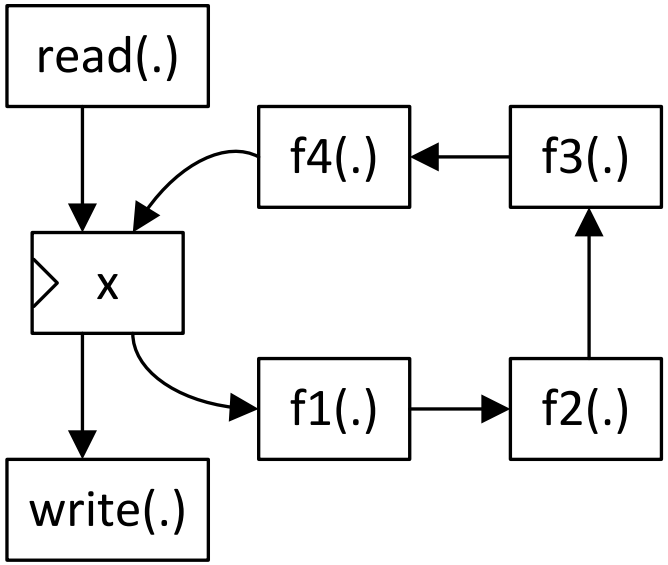
- Supported in shell via ‘tee’ and process substitution
  - e.g. Compress file to multiple types of archive

```
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);
```



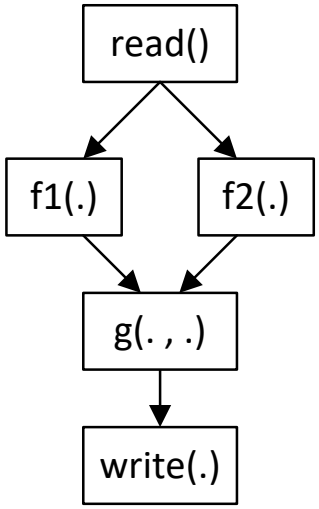
Reconvergent graphs

- 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);

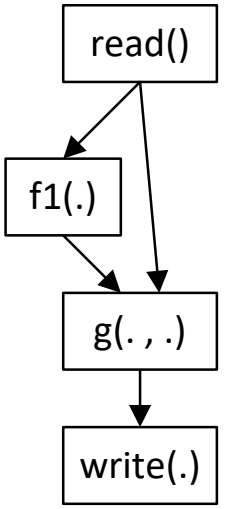
  write(z);
}
```




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

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

  write(z);
}
```



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

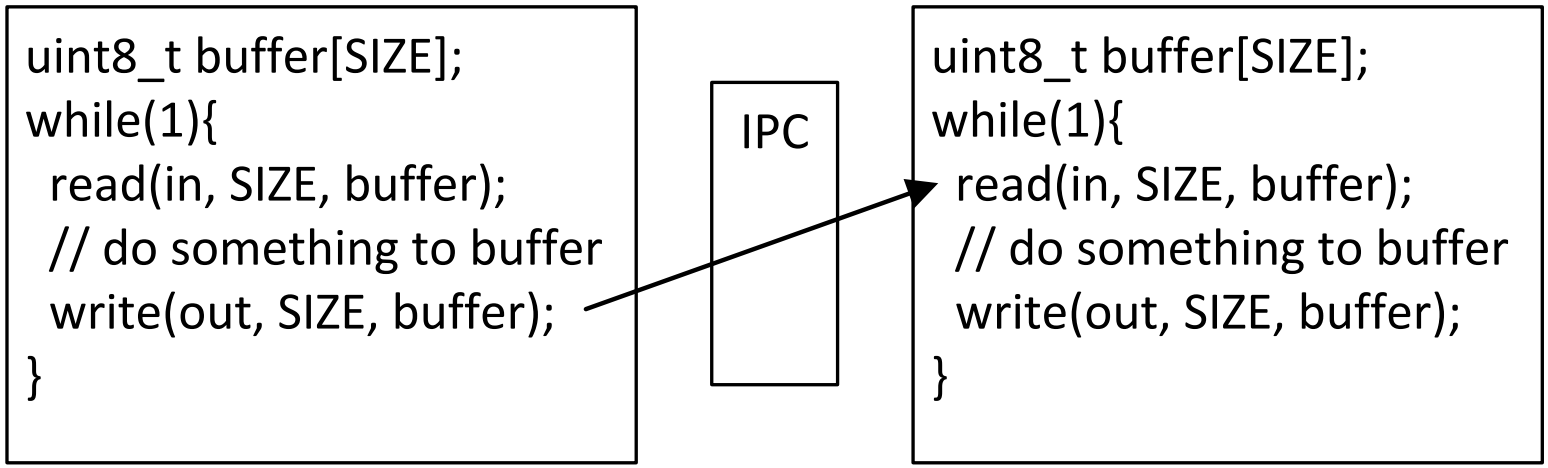
Don’t tell people I said pipes were the solution to big data

Problems with pipes

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

Communications overhead

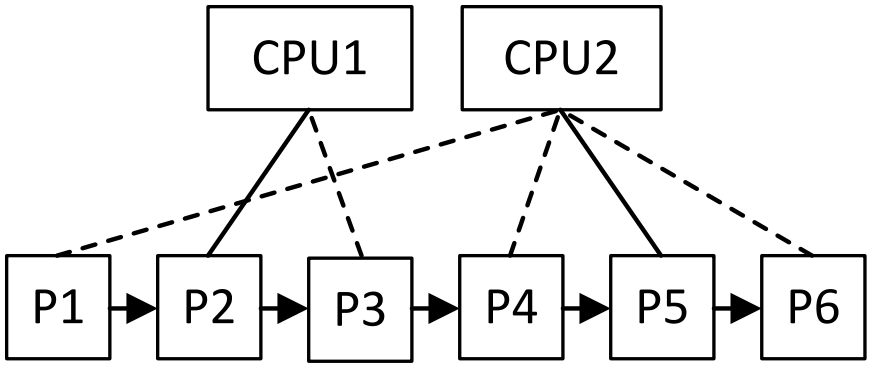
- Every transfer potentially involves some IPC
  - Data has to move from one address space to the other
  - *An advantage when you want to move data between machines*
- Need frequent calls to the Operating System



IPC = Interprocedural Communication

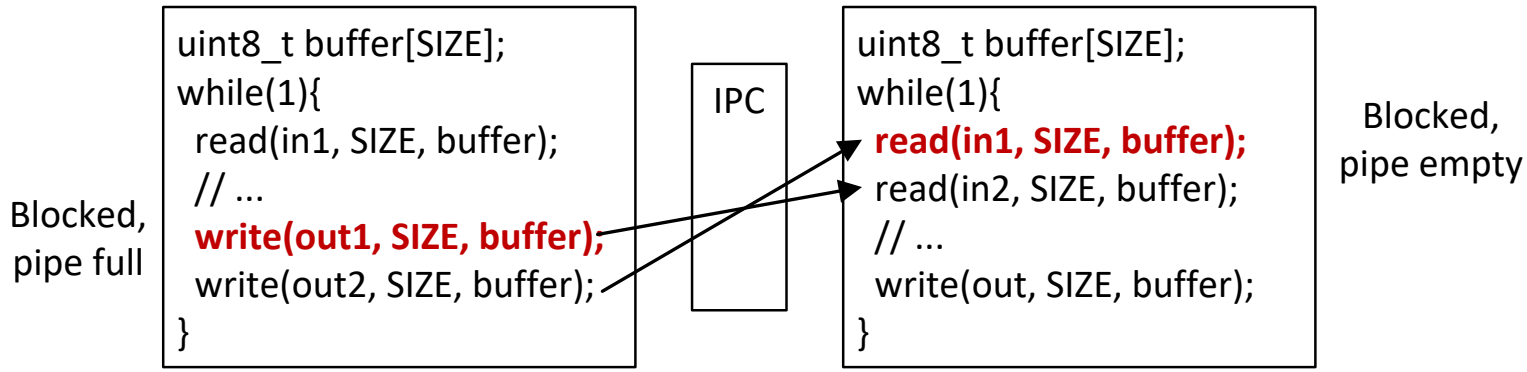
Scheduling overhead

- 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);

  // ...
}
```

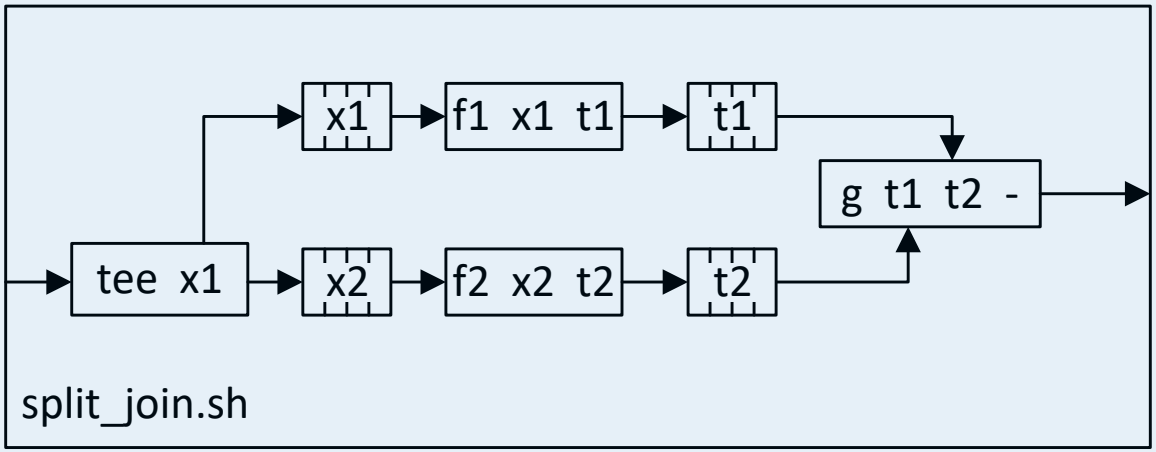
Non-Unix Pipelines

- TBB : using the `tbb::pipeline` library
  - All intermediate results stay in memory (and cache?)
  - Very good load-balancing
    - Limits the number of active stages to the number of available CPUs
    - Increases the number of samples in flight if there are lots of CPUs
  - Support for sequential stages in the pipeline
    - Some stages cannot be run in parallel (e.g. writing to a file)
- OpenCL : can create chains of kernel invocations
  - Create dependencies between kernel invocations using events
  - Data can remain in global memory between invocations
  - Can include copies to and from global memory at start and end

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);
}
```



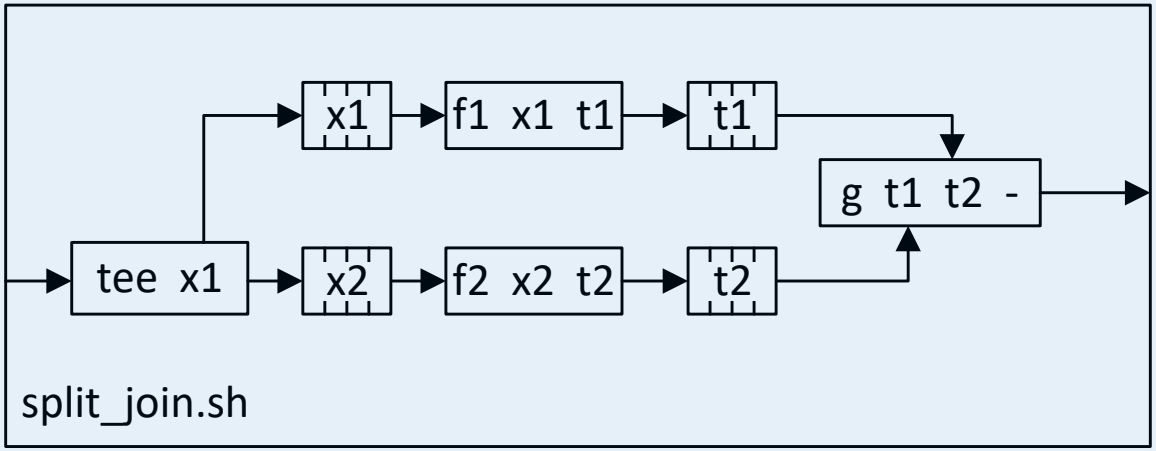
Reconvergent graphs using pipes

- 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 &
```



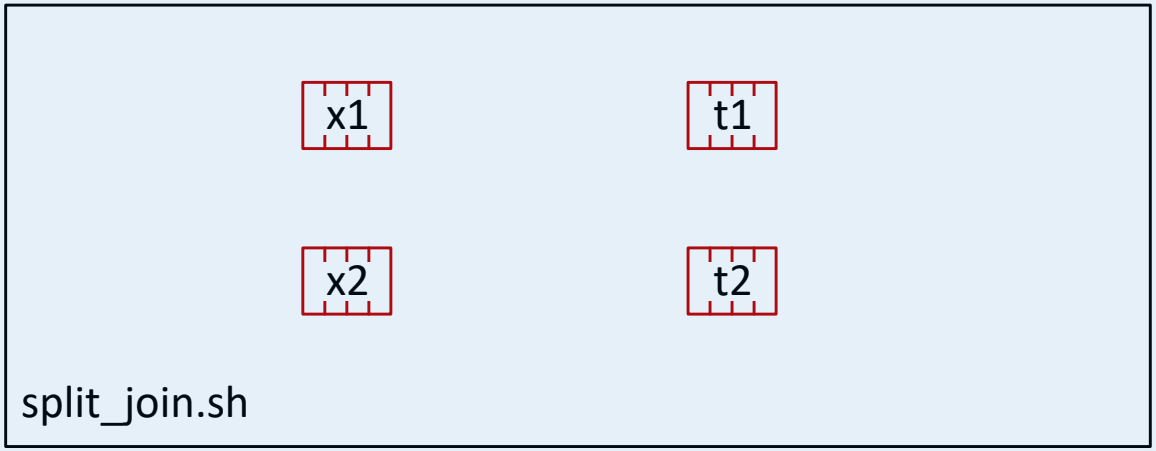
1 – Create FIFOs

- 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 &
```



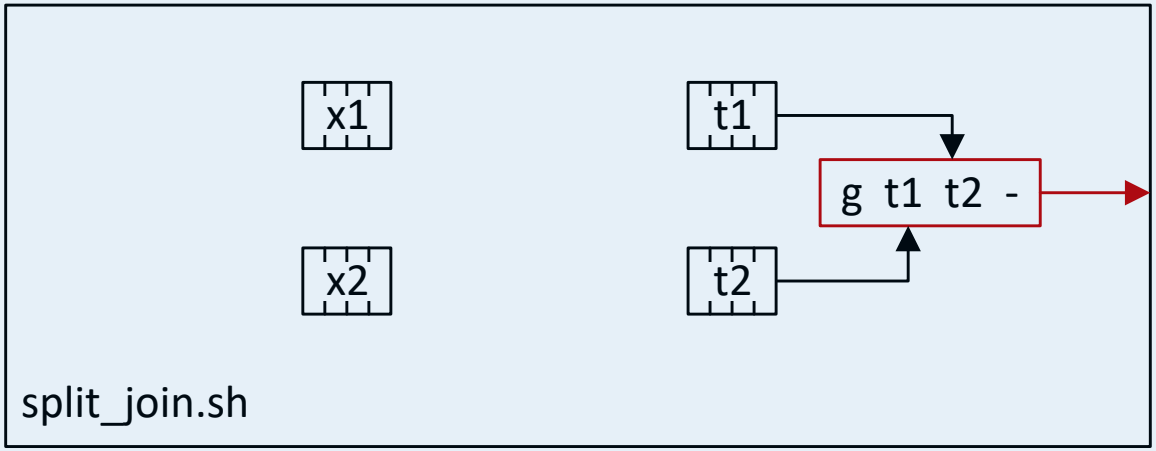
2 – Build backwards from the output

- 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 &
```



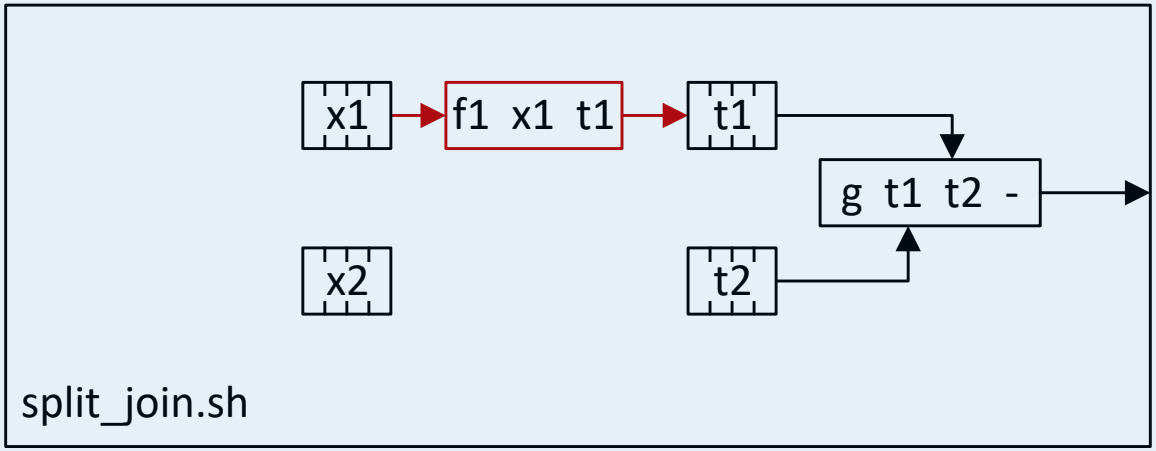
3 – Connect filters

- 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 &
```



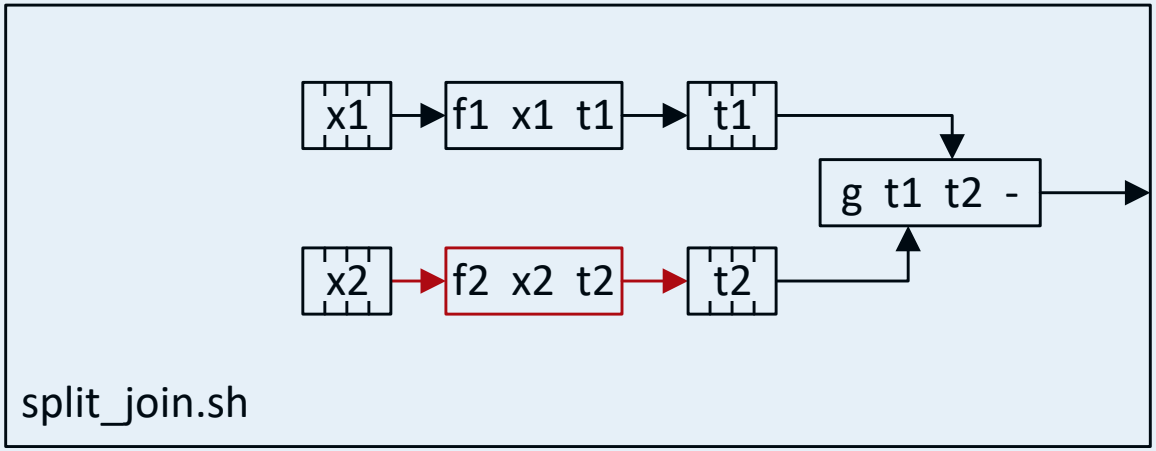
3 – Connect filters

- 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 &
```



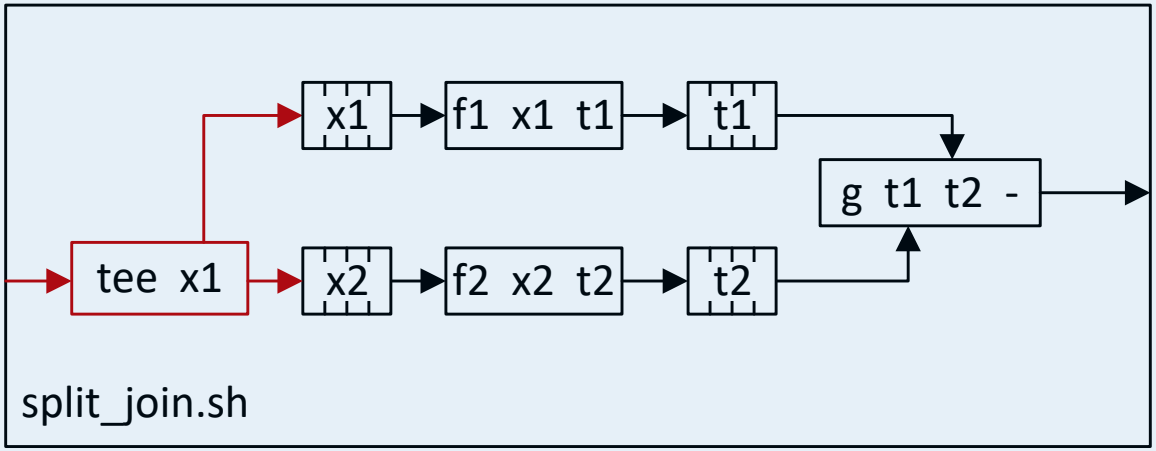
4 – Split input to each branch

- `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 &
```



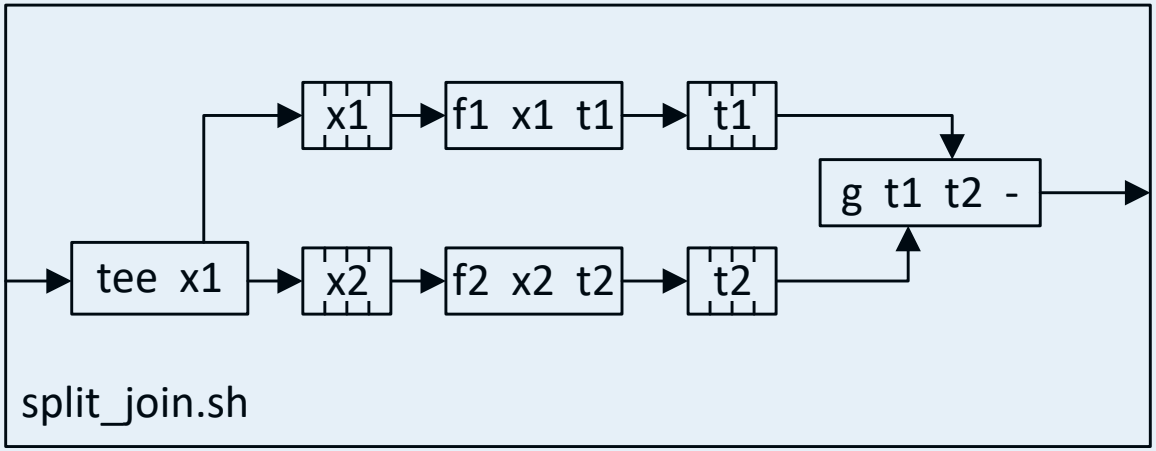
5 – Connect the inputs & outputs

- 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

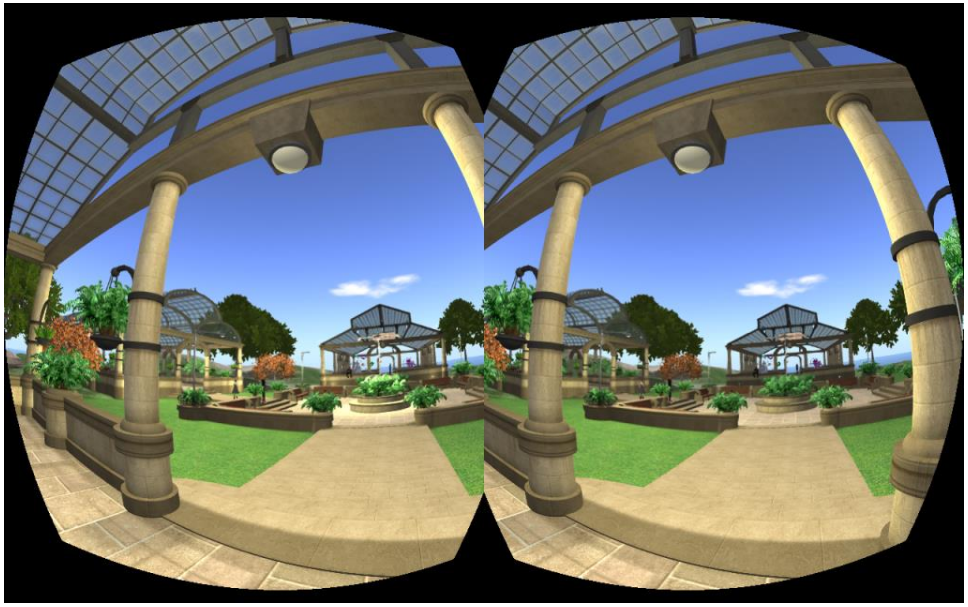
g t1 t2  &
f1 x1 t1  &
f2 x2 t2  &
tee x1 > x2 &
```



Performance: Metrics and Scales

- **Metric**: a way of measuring or characterising a system
  - May use “metric”, “measure”, or “property” depending on context
- **Scale**: the unit of measurement for a given metric
  - If you apply the metric, what sort of result does it give?
- Need consistent metrics within the design process
  - Requirements: *what properties should the solution have?*
  - Analysis: *what space of solutions meet the requirements?*
  - Design: *how best to implement the identified solution?*
  - Evaluation: *does the solution meet requirements?*
  - Optimisation: *can the solution be made any better?*
- Need metrics to compare multiple solutions
  - *Is system A better than system B?*

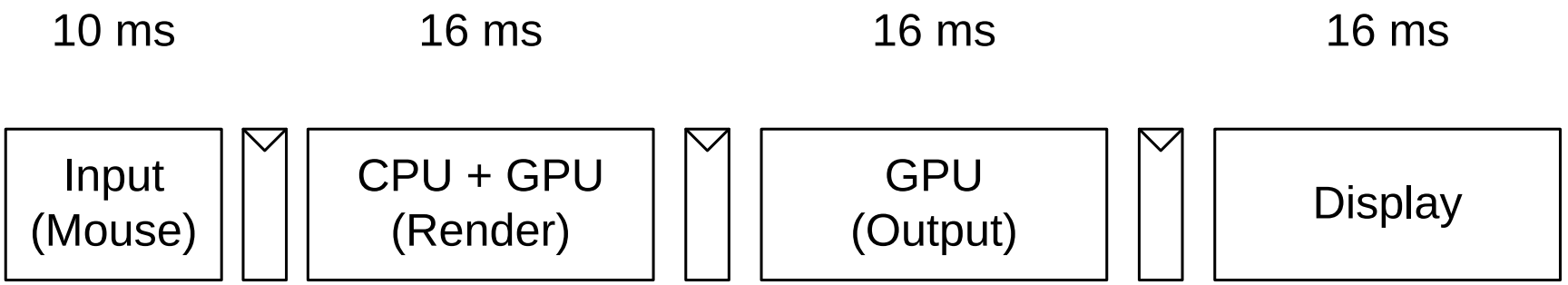
Example: Oculus Rift



All images owned by Oculus VR, as best I can tell

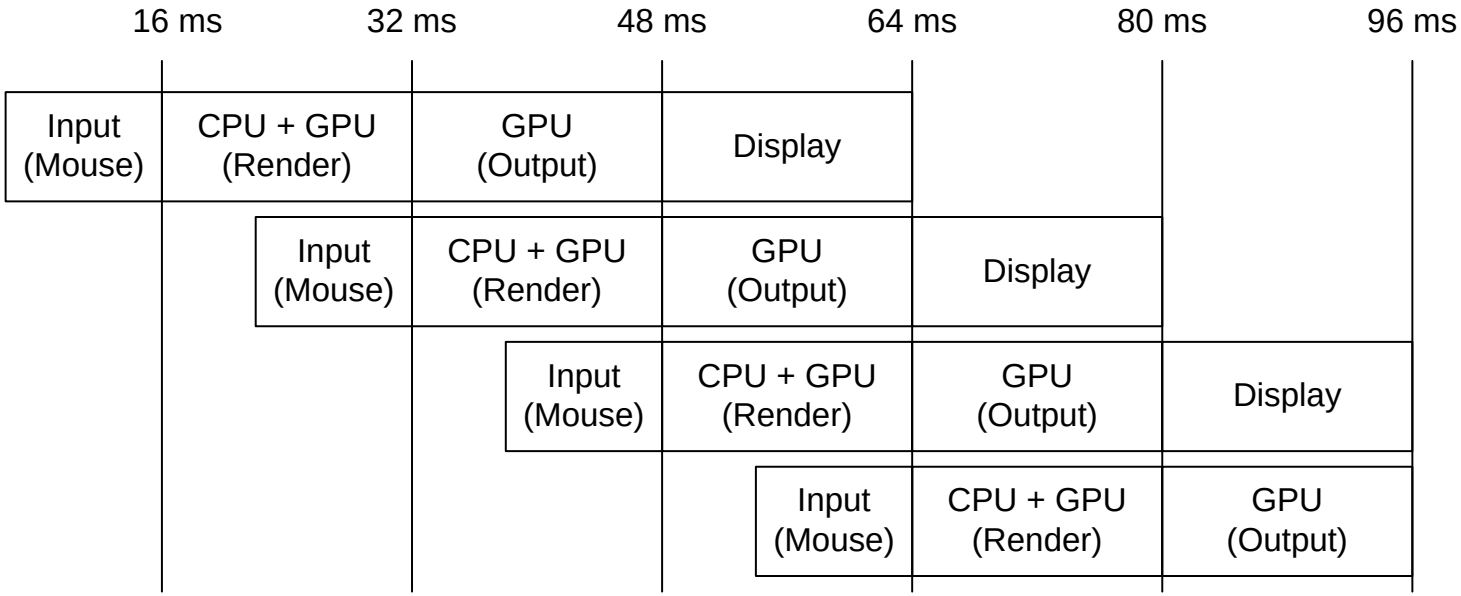
Typical graphics pipeline

- Most games can get 60Hz to 120Hz
  - 60Hz is usually enough for perceived smooth motion
- Results are buffered between each stage
  - Each stage has a throughput of 60Hz+
  - Takes some time to scan data from GPU RAM to display
  - Displays have persistence; brightness change not instantaneous



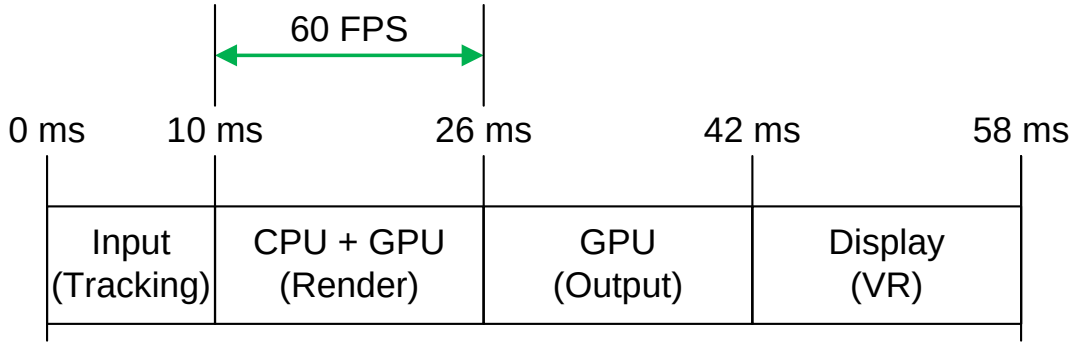
Typical graphics pipeline

- Classic pipeline: *performance is limited by the slowest stage*
- Once the pipeline is primed, we can get one frame per cycle



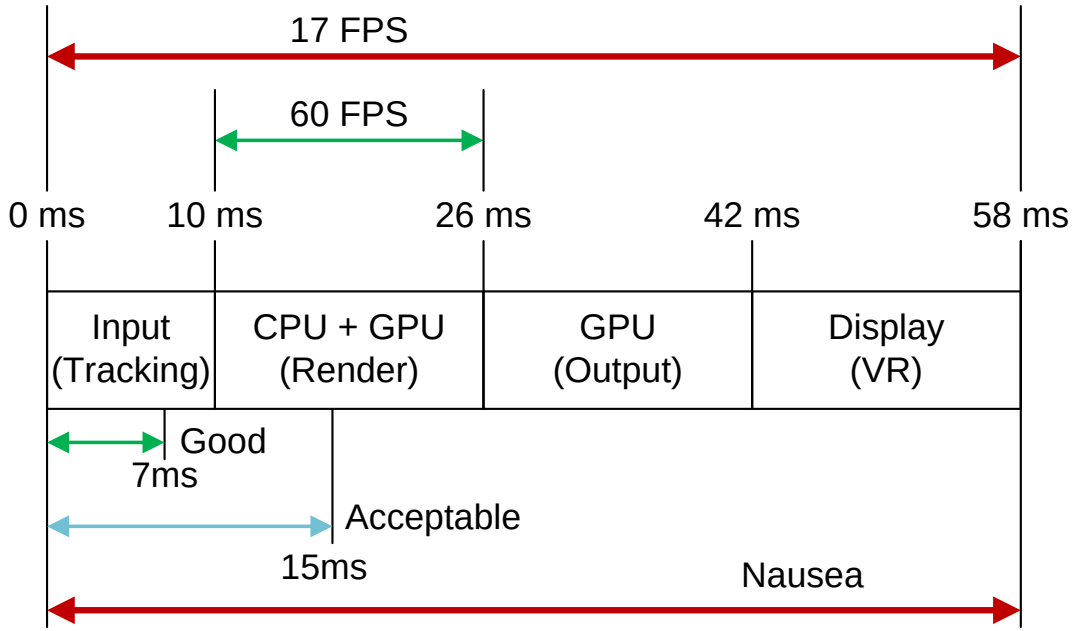
VR puts a constraint on latency

- Feedback loop: body->tracking->GPU->display->eye



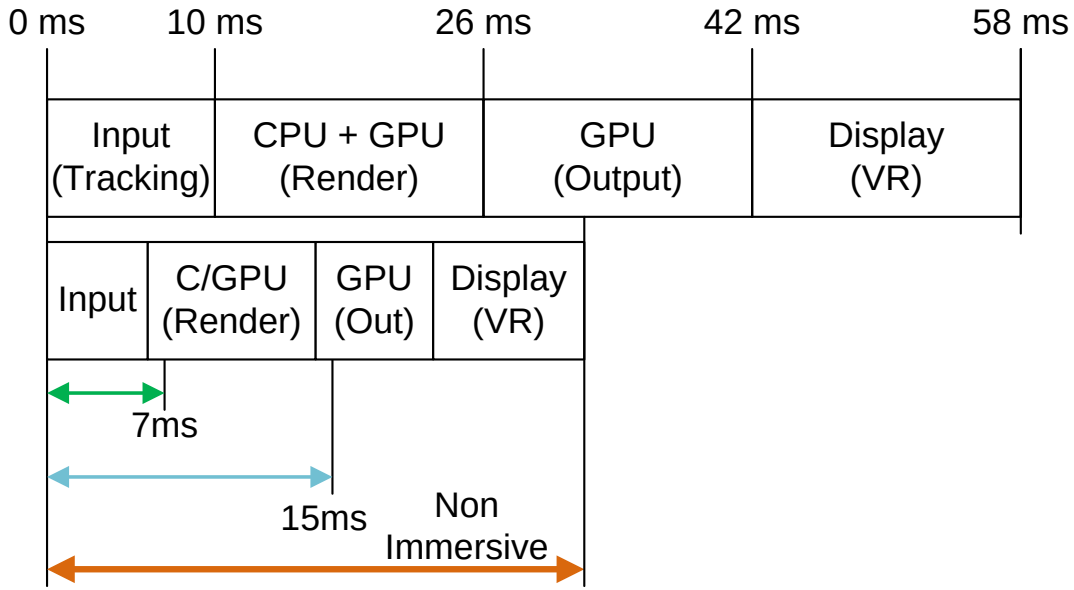
VR puts a constraint on latency

- Feedback loop: body->tracking->GPU->display->eye
- Humans are very sensitive to latency discrepancies
  - Mismatch between movement and vision: "*I'm poisoned*"!
  - Need 7-15ms for good user experience



Some solutions in hardware and software

- A number of things can reduce overall complexity
  - Reduce scene complexity and pixel count (late 90s quality)
  - Better input hardware with low latency
  - Different output technology: e.g. laser scanning displays
- Still difficult to get below 25ms (*even though 40FPS is trivial*!)



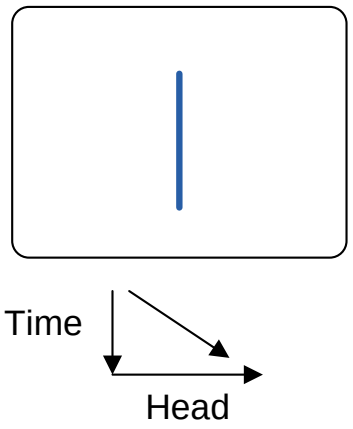
Remaining problem: distortion

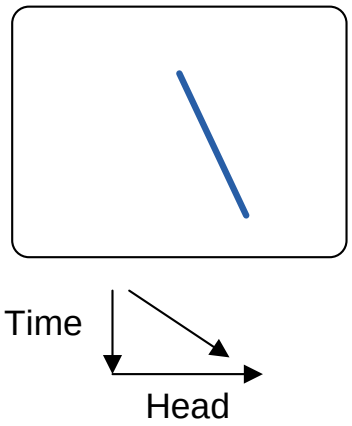
- Images are displayed in raster scan order
- Both eyes and the head are moving independently

Michael Abrash:
http://blogs.valvesoftware.com/abrash/raster-scan-displays-more-than-meets-the-eye/

Remaining problem: distortion

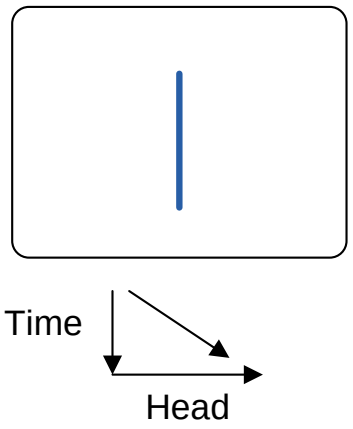
- Images are displayed in raster scan order
- Both eyes and the head are moving independently
  - Perceived object is different from displayed object

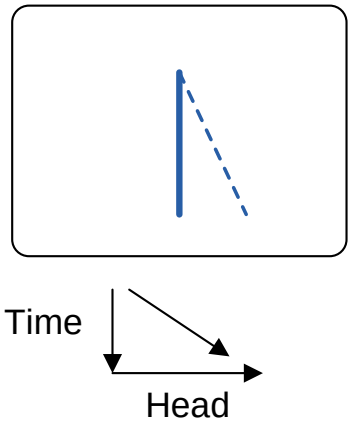




Remaining problem: distortion

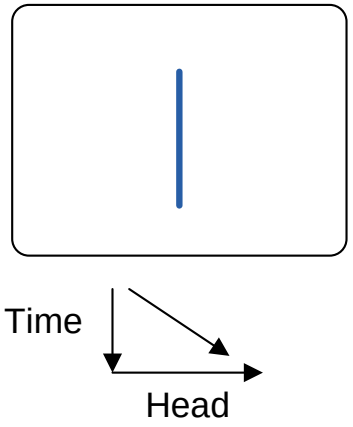
- Images are displayed in raster scan order
- Both eyes and the head are moving independently
  - Perceived object is different from displayed object
- Can try to pre-warp the geometry according to motion

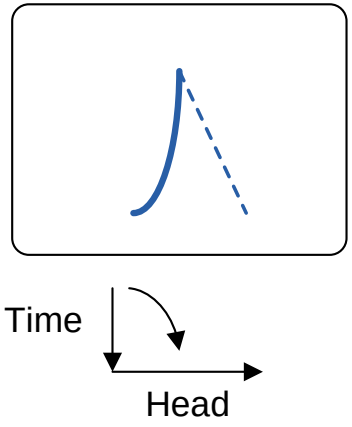




Remaining problem: distortion

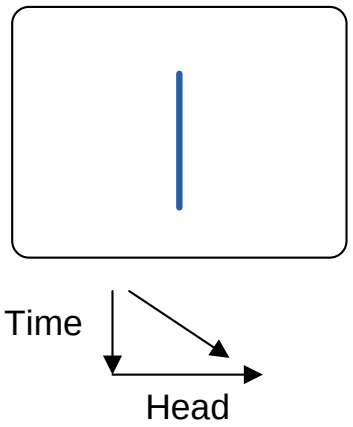
- Images are displayed in raster scan order
- Both eyes and the head are moving independently
  - Perceived object is different from displayed object
- Can try to pre-warp the geometry according to motion
  - But we only update at 17 FPS: motion can change

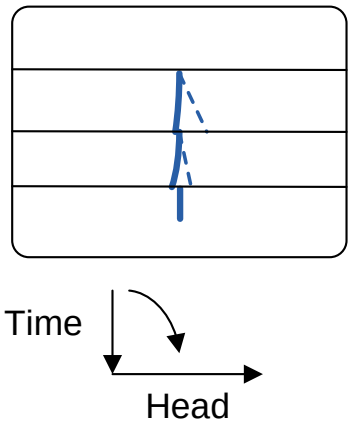




Partial solution: *race the beam*

- Render image in horizontal chunks
  - Each chunk uses the very latest tracking information
  - Chunks are sent to display as soon as they are ready
- The smaller the chunks, the less perceived distortion





“Performance” is application dependent

- Throughput and latency are related, but very different
  - Throughput of X -> latency >= 1/X
  - Latency of L -> throughput = 1/L (usually)
- Usually quite easy to get high throughput
  - Throughput of 10 GFlops vs latency of 100 ps
  - Throughput of 1000 FPS vs latency of 1ms per image

- People try to optimise one to improve the other
  - Increase mouse update speed
  - Tweak GPU settings to get 120Hz
  - (Do they have any effect?)

Final Year Projects…

The Interim Report should contain at least the following sections:

- Project specification
- Background
- Implementation Plan
- Evaluation Plan
- Ethical, Legal, and Safety Plan

The project specification should state clearly what the project is intended to deliver, including all hardware, software, simulation, and analytical work, and provide some motivation.

Results

This covers an area different from the 'Testing' chapter, and relates to the understanding and analysis of the algorithm, program, or hardware behaviour. Where the deliverable is a product with easily quantified performance then the previous Chapter shows how functional correctness is established, while this Chapter shows qualitatively or quantitatively how well the product works. The list below shows typical content, not all of this will be applicable to all projects.

- An empirical relationship between parameters and results may be investigated, typically through the use of appropriate graphs.
- Where theory predicts aspects of performance the results can be compared with expectation and differences noted and (if possible) explained.
- Semi-empirical models of behaviour may be developed, justified, and tested.

Qualitative vs Quantitative metrics

- **Quantitative**: hard numbers, measurable, objective
  - Time, Energy, Space
  - Signal-to-Noise, Frames-per-second, Memory Usage
  - Money (?)
- **Qualitative**: feelings, opinions, subjective
  - Complexity: “Simple”, “Tricky”
  - Design Effort: “Easy”, “Hard”
  - User Experience: “snappy”, “intuitive”, “pretty”

- Try to use quantitative measurements when possible
  - Repeatable: all measurements yield the same result
  - Verifiable: third-parties should get the same result

Types of scale

- **Nominal**: categories with no intrinsic order
  - Apples, Oranges, Pears
- **Ordinal**: categories which can be compared
  - Bad, Ok, Good
- **Interval**: numbers with meaningful differences
  - Year
- **Ratio**: numbers with differences and a meaningful zero
  - Frames per second

Choosing scales

- A metric can be defined in terms of many scales
- Temperature
  - *Nominal*: “Balmy”, “Temperate”
  - *Ordinal*: Cold, Chilly, Warm, Hot
  - *Interval*: Degrees Fahrenheit
  - *Ratio*: Degrees Kelvin
- Can create mapping from ratio to other scales
  - Fahrenheit = 9 * K / 5 − 459.67
  - Cold = {K<280}, Chilly = {K>=280 && K<=288}, ...
  - Balmy = {K>293 && K<300}; Temperate = {K>288 && K<=293}
- Can’t go from nominal to ratio without inventing information
  - “Cold” represents a range, but Kelvin is a point

The joys of ratio scales

- Easy to see which system is better
  - If metric(A) < metric(B) then B is better than A
- Easy to see *by how much* the system is better
  - A is better than B by a factor metric(A)/metric(B)
- We can define meaningful combination metrics
  - metric(X) = metric(A) + metric(B) d=1
  - metric(Y) = sqrt[metric(A)^2 + metric(B)^2] d=2
  - metric(Z) = max[metric(A),metric(B)] d=∞
  - metric(G) = [metric(A)^d + metric(B)^d]$^{1/d}$

Averaging of ratio metrics

- Many metrics are formed from a number of measurements
  - Benchmarks: perform the same task on many different inputs
  - Smoothing: remove statistical noise from different runs
- Ratio metrics can be meaningfully averaged
  - But usually ***not*** using the arithmetic mean
- Geometric mean g(x)
  - Good for combining values on different scales
  - Useful for summarising improvements
- Harmonic mean h(x)
  - Sometimes useful for averaging rates
  - *Usually better to stick to geometric*

$$a(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$g(\mathbf{x}) = \sqrt[n]{\prod_{i=1}^{n} x_i}$$

$$h(\mathbf{x}) = \frac{n}{\sum_{i=1}^{n} 1/x_i}$$

$$h(\mathbf{x}) \leq g(\mathbf{x}) \leq a(\mathbf{x})$$

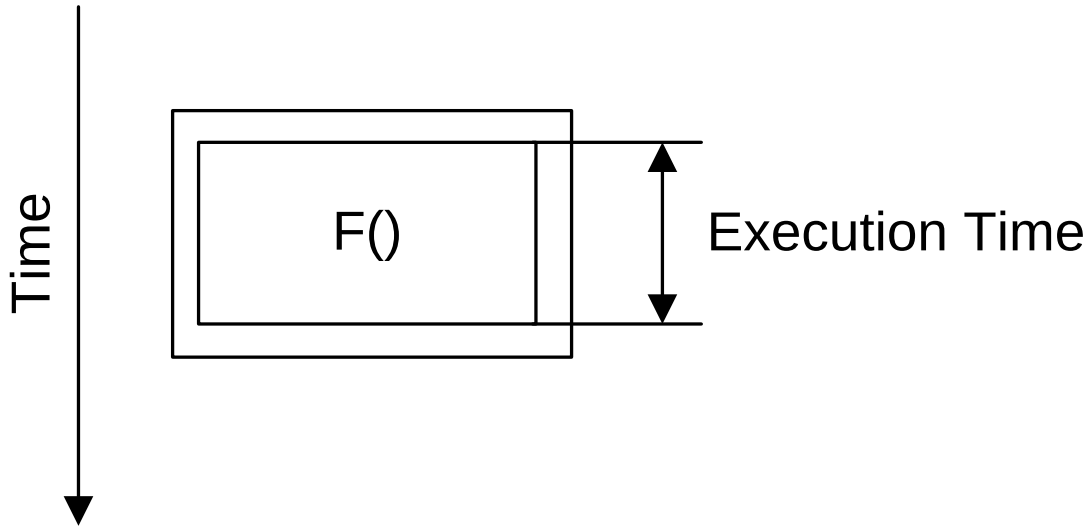
"How not to lie with statistics: the correct way to summarize benchmark results", P. Fleming, J. Wallace, 1986, http://dx.doi.org/10.1145%2F5666.5673

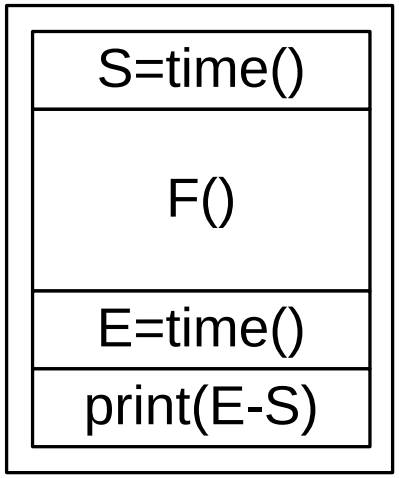
Execution Time

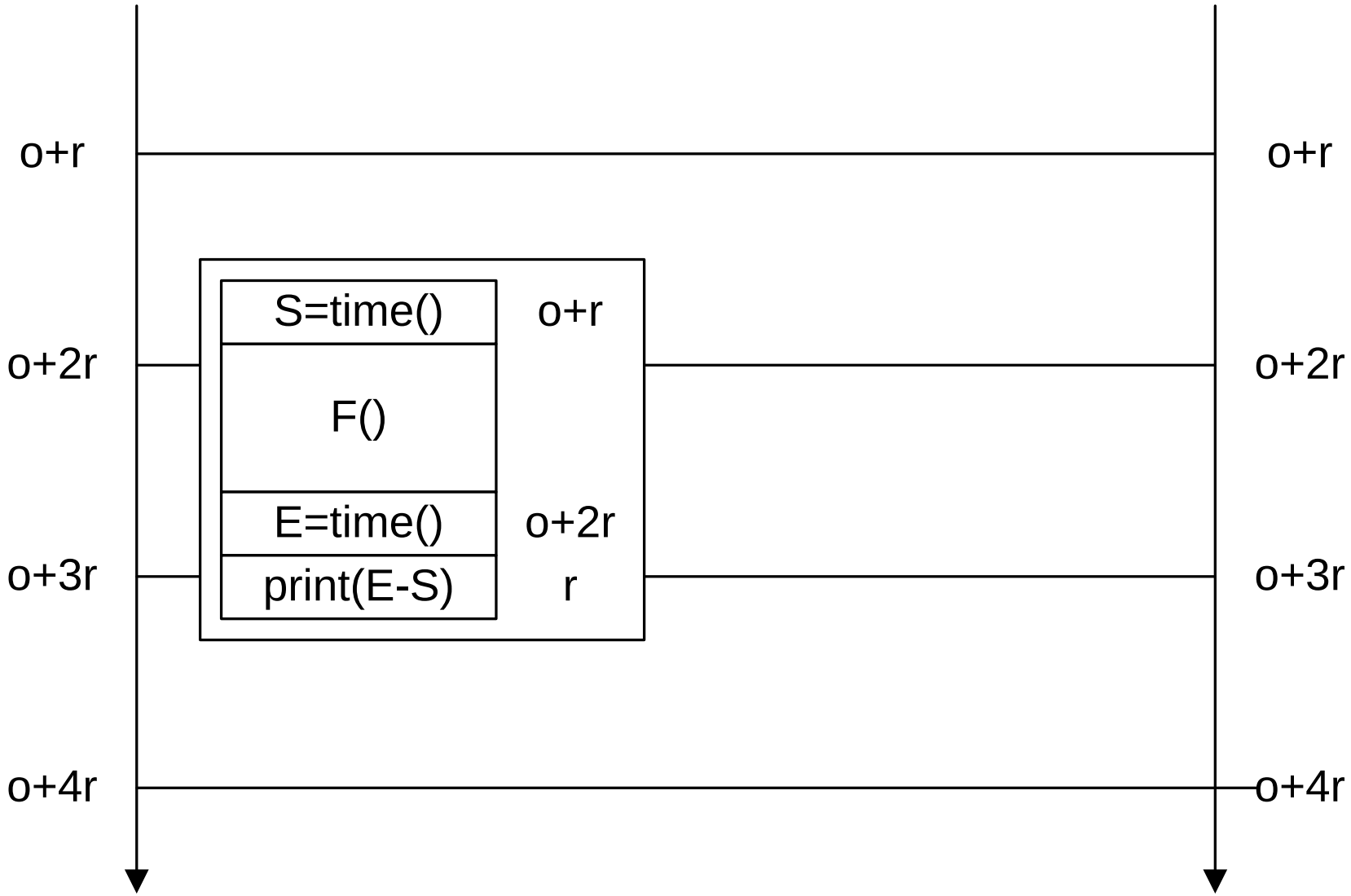
- How long does it take to complete some task X?
  - Fundamental property of a compute system
- Scale is elapsed time: seconds, hours, years(!)
  - Ratio scale: *can't take less than zero seconds*
  - Ratio scale: *one second is 2x better than two seconds*
- Need to carefully define what the components mean
  - What precisely is task X?
  - When does timing start, when does it stop?
  - What sort of time is being measured
    - Wall-clock time: e.g. human with a stop-watch
    - CPU time: only measure time when the task was executing

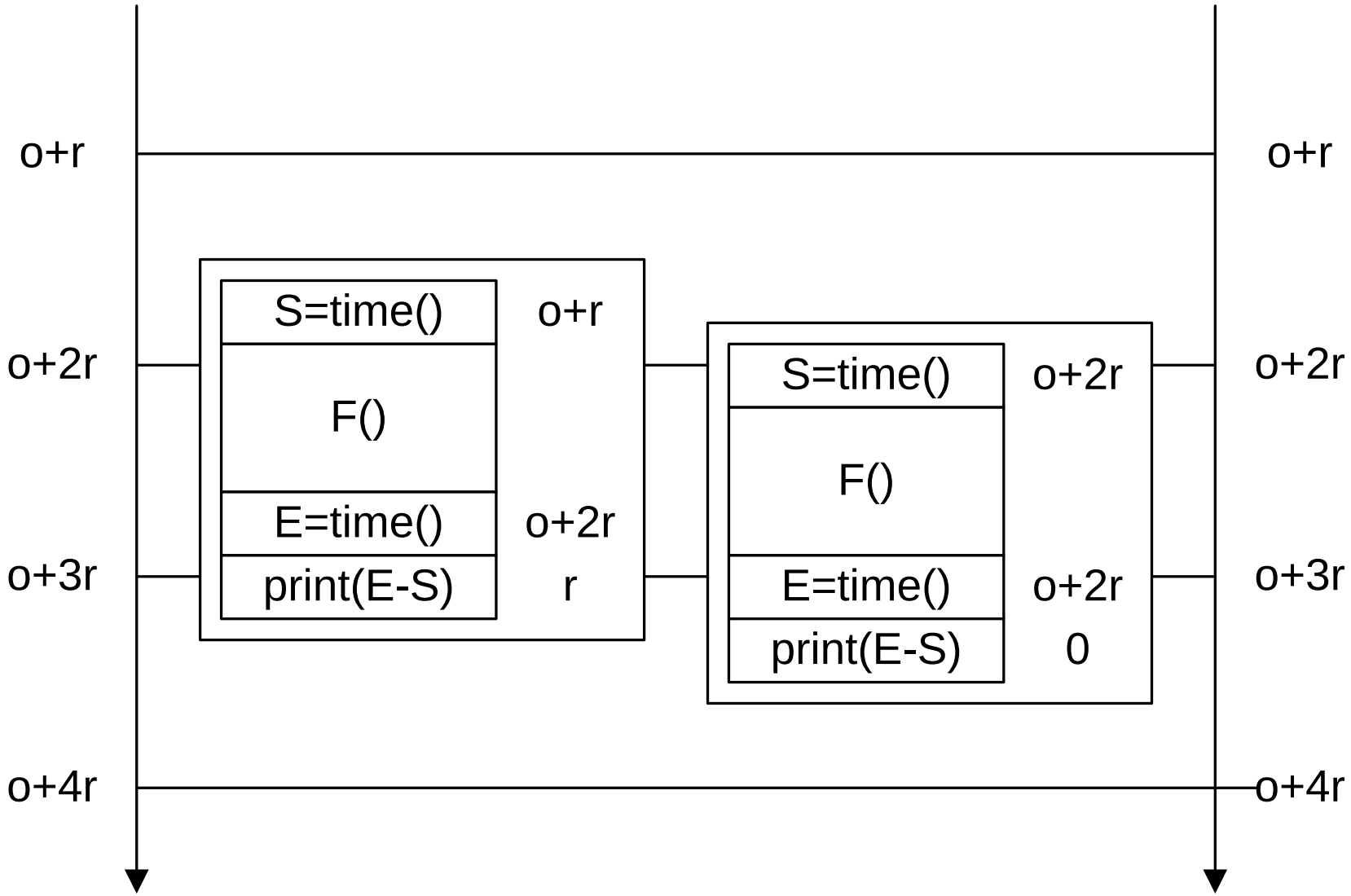
Measuring elapsed time

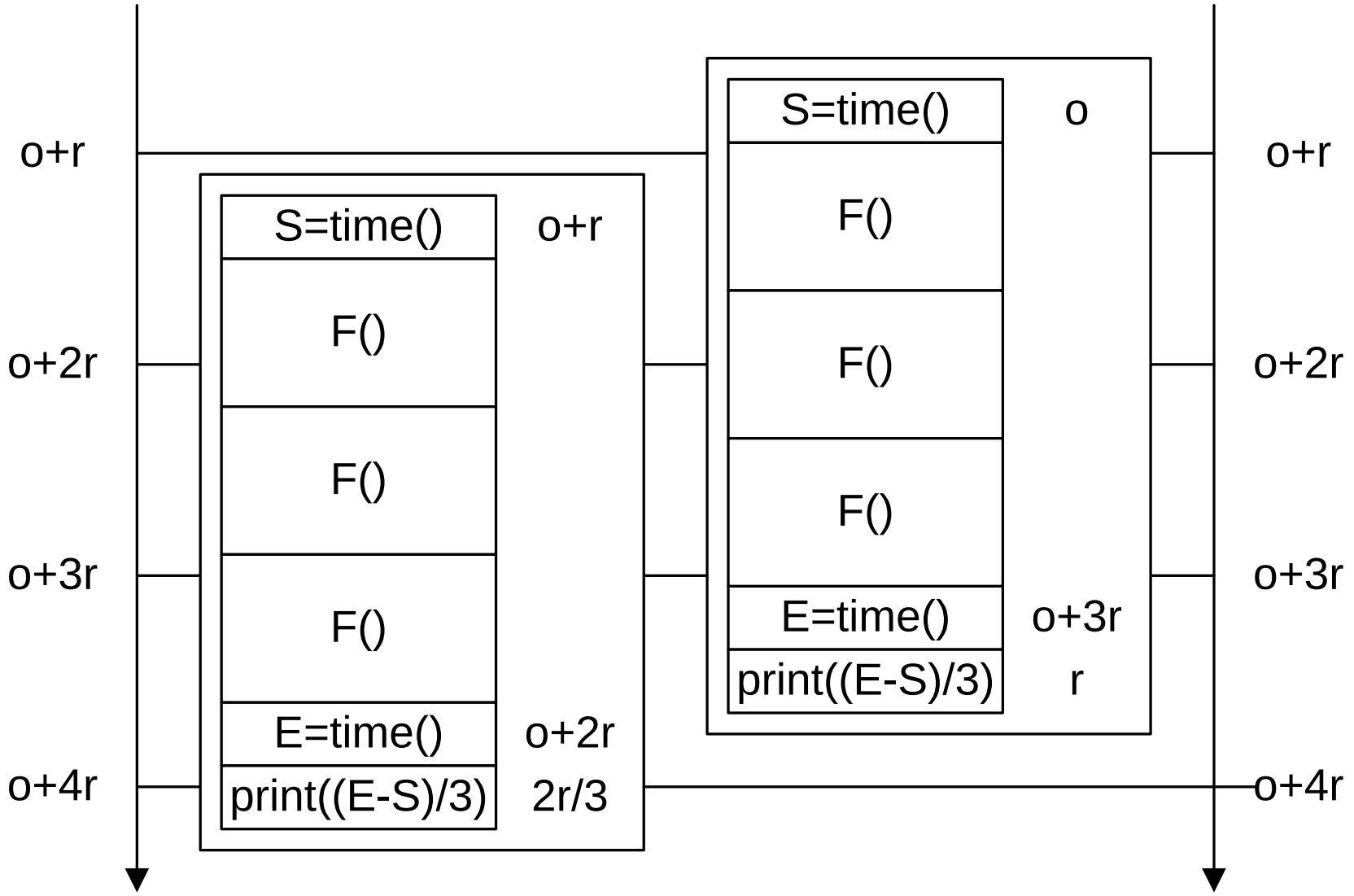
- This is often surprisingly difficult!
- We need some sort of timer as a base
  - **Precision**: Smallest representable time difference
  - **Resolution**: Smallest time difference you can measure
  - **Accuracy**: How reliable is the timer
- Timer can be part of the system, or external
  - *External*: you with a stop-watch, or pulse timer
  - *Internal*: the system itself records time during execution
- Internal timers are convenient, but tricky
  - Performing timing may change the systems performance!
  - Timer may not be consistent across the system

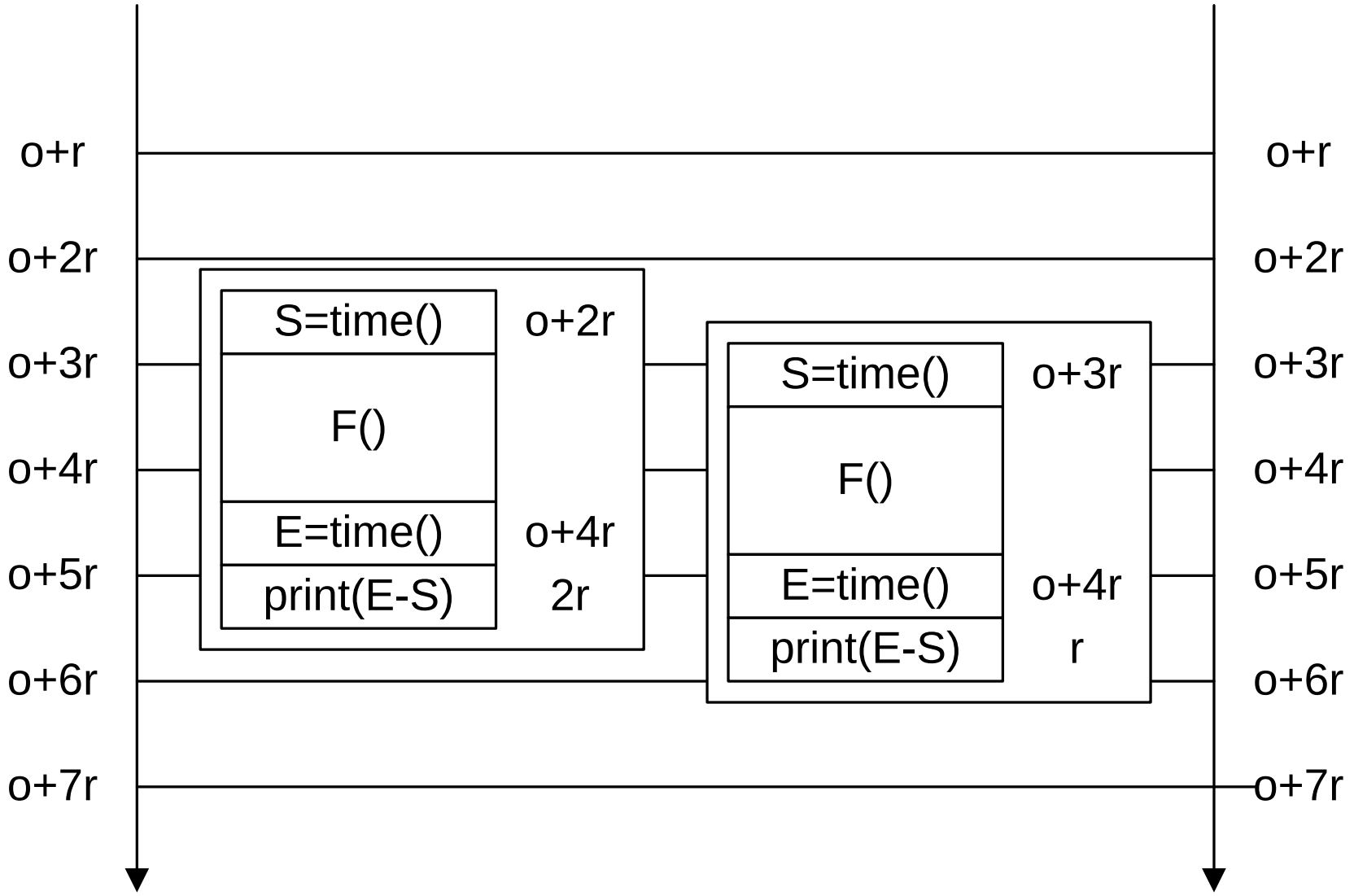

Time
F()
Execution Time


S=time()
F()
E=time()
print(E-S)


o+r
o+r
S=time()
o+r
o+2r
F()
o+2r
E=time()
o+2r
o+3r
print(E-S)
r
o+3r
o+4r
o+4r


o+r
o+2r
o+3r
o+4r
S=time()
F()
E=time()
print(E-S)
o+r
o+2r
r
S=time()
F()
E=time()
print(E-S)
o+2r
o+2r
0
o+r
o+2r
o+3r
o+4r


o+r
o+2r
o+3r
o+4r
S=time()
o+r
F()
F()
F()
E=time()
o+2r
print((E-S)/3)
2r/3
S=time()
o
F()
F()
F()
E=time()
o+3r
print((E-S)/3)
r
o+r
o+2r
o+3r
o+4r


o+r
o+2r
o+3r
o+4r
o+5r
o+6r
o+7r
S=time()
o+2r
F()
E=time()
o+4r
print(E-S)
2r
S=time()
o+3r
F()
E=time()
o+4r
print(E-S)
r
o+r
o+2r
o+3r
o+4r
o+5r
o+6r
o+7r

Internal timers

- Most platforms provide calls which will return elapsed time

	Windows	**Linux**	**x86 CPU**
Timer	QueryPerformanceCounter()	clock()	RDTSC
Resolution	micro-second	milli-second	nano-second
Measures	Wall-Clock	CPU-Time	Clock-Cycles
Consistency	Entire System	Current Thread	Current CPU

Problems when measuring time

- Start-up overhead: the first execution usually takes longer
  - Filling caches; compiling code; performing initialisation
  - **Cold-start**: restart system; time first execution of task
  - **Warm-start**: execute task once; measure the second execution
- Interference: other tasks are running on the system
  - Multi-user system: other users may execute system
  - Operating-system tasks: e.g. disk utilities affect performance
- Intrinsic variability: computers are not very deterministic
  - Cache contents differ significantly on each run
  - Unpredictable scheduling of threads by system
  - Interaction between resolution of timer and length of task
- Try to measure over multiple executions and average

Throughput

- At what ***rate*** can task X be executed
  - Unit is executions per time
- Throughput is ***not*** just the inverse of elapsed time
  - Elapsed time is latency – time for a **single** task
  - Throughput is the execution rate for **many** tasks
- There is often a trade-off between latency and throughput
  - Parallel systems can do lots of tasks in parallel: good throughput
  - Difficult to split a single task into many pieces: poor latency

- Frames-per-second our obvious throughput example
  - Gaming: need high frames/second ***and*** low latency
  - Video: need high frames/second, but latency is not important

Energy

- Definitions of energy consumption are application specific
  - How much energy is used while executing task X?
  - What is the average power of a system capable of task X?
  - What is the peak power of a system executing task X?
- Closely related to temperature metrics
  - How hot does the device get while doing X?
  - What are the cooling requirements for continually doing X?
- Computers ***must*** consume energy: Landauer's principle
  - Any bit-wise operation (e.g. `and`, `or`) requires k*T*log(2) joules
    - k: Boltzmann constant; T: circuit temperature
  - e.g. a task requires 2^{64} bit-wise operations at 60 degrees C
    - $3.2*10^{-21}$ joules/bit -> ~0.05 joules
  - We're not that close to the limit yet...

Memory constraints

- All levels of storage: memory, cache, disk
  - All the working memory needed during calculation
  - May include input and output storage: are they the same place?
  - Memory constraints are balanced against communication costs
- There is an explicit size/bandwidth constraint

Level	Capacity	Bandwidth
Remote “Disk” (SAN)	10 EB	100 MB/sec
Local “Disk” (SSD)	10 TB	500 MB/sec
Memory (GDDR)	8 GB	300 GB/sec
Cache (local mem)	128 KB	2 TB/sec

- Often we will trade-off computation versus storage
  - Or programmer effort versus IO

Quality

- There is often a space of answers: how good is an answer?
  - Compression quality
  - Signal to noise ratio
- Qualitative quality metrics can often be made quantitative
  - User-interface responsiveness -> Input to display latency
  - Image quality -> root-mean-square-error against reference model
- If quality is a ratio metric, there might be a “perfect” answer
  - But: realities of floating-point maths means this may not happen
- Understanding quality is critical for high performance
  - Achieving the “perfect” answer is often computationally intractable
  - Usually a trade-off between quality and speed
  - Need to choose the best tradeoff between the two

Making sure metrics are meaningful

- Some things are quantifiable, but not very useful
  - CPU performance: MHz is not the same as performance
  - Cameras: Mega-Pixels is not the same as quality
- Consistent and quantifiable metrics provide open competition
  - Suppliers of systems always want to use the “best” metrics
  - Metrics should be defined by users or communities, not suppliers
- People will optimise for metrics (it’s what they are for!)
  - Poor metrics lead to poor design and optimisation
  - Part of the specification problem