Concurrent Programming III – Differing Tasks

HPPS

David Marchant

Approaches to concurrency

- So far we've looked at OpenMP and threading
- These are what we use most often, especially where we simply want to go faster

Reminder of problems so far. . .

- **Classical problem classes of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
- **As well as these, some problems just aren't suited to identical threads, we want different processing doing different things.**

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
      - *Much* larger difference on non-Unices.

Threading example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int THREADS = 4;

struct thread_args {
    pthread_t thread_id;
    int thread_num;
};

void *my_thread(void *arg) {
    struct thread_args *thread_info = arg;
    printf("Hello from thread %d\n", thread_info->thread_num);
    return NULL;
}

int main() {
    struct thread_args *all_thread_info = calloc(THREADS, sizeof(*all_thread_info));
    pthread_t* thread_nums[THREADS];
    for (int i=0; i<THREADS; i++) {
        all_thread_info[i].thread_num = i + 1;
        pthread_create(&all_thread_info[i].thread_id, NULL, my_thread, &all_thread_info[i]);
    }

    for (int i=0; i<THREADS; i++) {
        pthread_join(all_thread_info[i].thread_id, NULL);
    }

    free(all_thread_info);
    exit(0);
}
```

```
user@system:~ gcc -o example example.c -lpthread
user@system:~ ./example
Hello from thread 2
Hello from thread 1
Hello from thread 3
Hello from thread 4
user@system:~
```

example.c

pthreads vs. OpenMP

- Generally OpenMP is:
  - Quicker and easier to implement
  - Good enough
  - Handles a lot of races and deadlocks for you
- Manual pthreads is:
  - Not actually that hard
  - A lot more verbose
  - Lets you make all manner of errors
  - More scope for fine tuning
  - You aren’t just limited to parallelising loops

but OpenMP isn’t as limited as I first suggested

What we’ve seen so far

```
void vector_add(int n, const int *a, const int *b, int *c) {
	#pragma omp parallel for
	for (int i = 0; i < n; i++) {
		c[i] = a[i] + b[i];
	}
}
```

- This function adds two vectors A and B into C
- For N items we start N parallel regions
- Possibly each region is one thread, but this is hardware/scheduler dependent

Dividing Into Chunks

```
void vector_add_quartered(int n, const int *a, const int *b, int *c) {
    int chunks = 4;
    int chunk_size = n / chunks;
    #pragma omp parallel for
    for (int chunk=0; chunk<chunks; chunk++) {
        int start = chunk * chunk_size;
        int end = start + chunk_size;
        for (int i = start; i < end; i++) {
            c[i] = a[i] + b[i];
        }}}
```

- Here we divide the problem N into 4 chunks
- Assuming we've 4 cores, this could neatly map with the minimum threading overhead

Programmatic Chunks

```
void vector_add_dynamic(int n, const int *a, const int *b, int *c) {
	#pragma omp parallel {
		int t = omp_get_thread_num();
		int P = omp_get_num_threads();
		int chunk_size = n / P;
		int start = t * chunk_size;
		int end = start + chunk_size;
		if (t == omp_get_num_threads()-1) {
			end = n;
		}
		for (int i = start; i < end; i++) {
			c[i] = a[i] + b[i];
		}}}
```

- Better still would be to dynamically detect how many chunks to split into
- Note we need to account for an uneven divide in our last chunk
- Also no longer using **for** clause

Why bother?

- **Avoids the overhead of starting and stopping loads of unnecessary threads**

- **Can be used to gives different problems to different threads**

- **Usually we want the same *task*, but on different *data***

- **This example could be achieved with a reduction clause, but we could use the same logic to manage problems that reductions can't**

Key functions

```
int omp_get_thread_num(void);
```

- Gets a unique id number for the calling openmp thread
- Must be called within openmp scope

```
int omp_get_num_threads(void);
```

- Gets count of openmp threads in current openmp scope
- Must be called within openmp scope

```
int omp_get_max_threads(void);
```

- Gets count of theoretical max scheduled by openmp in the current program
- Can be called anywhere

Note that all of these require:

```
#include <omp.h>
```

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
      - *Much* larger difference on non-Unices.

Processes in C

- **Relatively few important functions**
  - fork() - creates a duplicate child process
  - execve() - replace current process with specified program
  - getpid() - get current process id
  - wait() - Wait for any child processes to complete
  - waitpid() - Wait for any or specific child process to complete
- **Note that these won't work on windows, so we won't really focus on them**
- **Even on non-windows, you're probably better using threads as they are much faster**
- **Processes tend to be easier to program though (ish)**

Processes in C

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    fork();
    fork();
    printf("Hello from process %d\n", getpid());
    exit(0);
}
```

fork.c

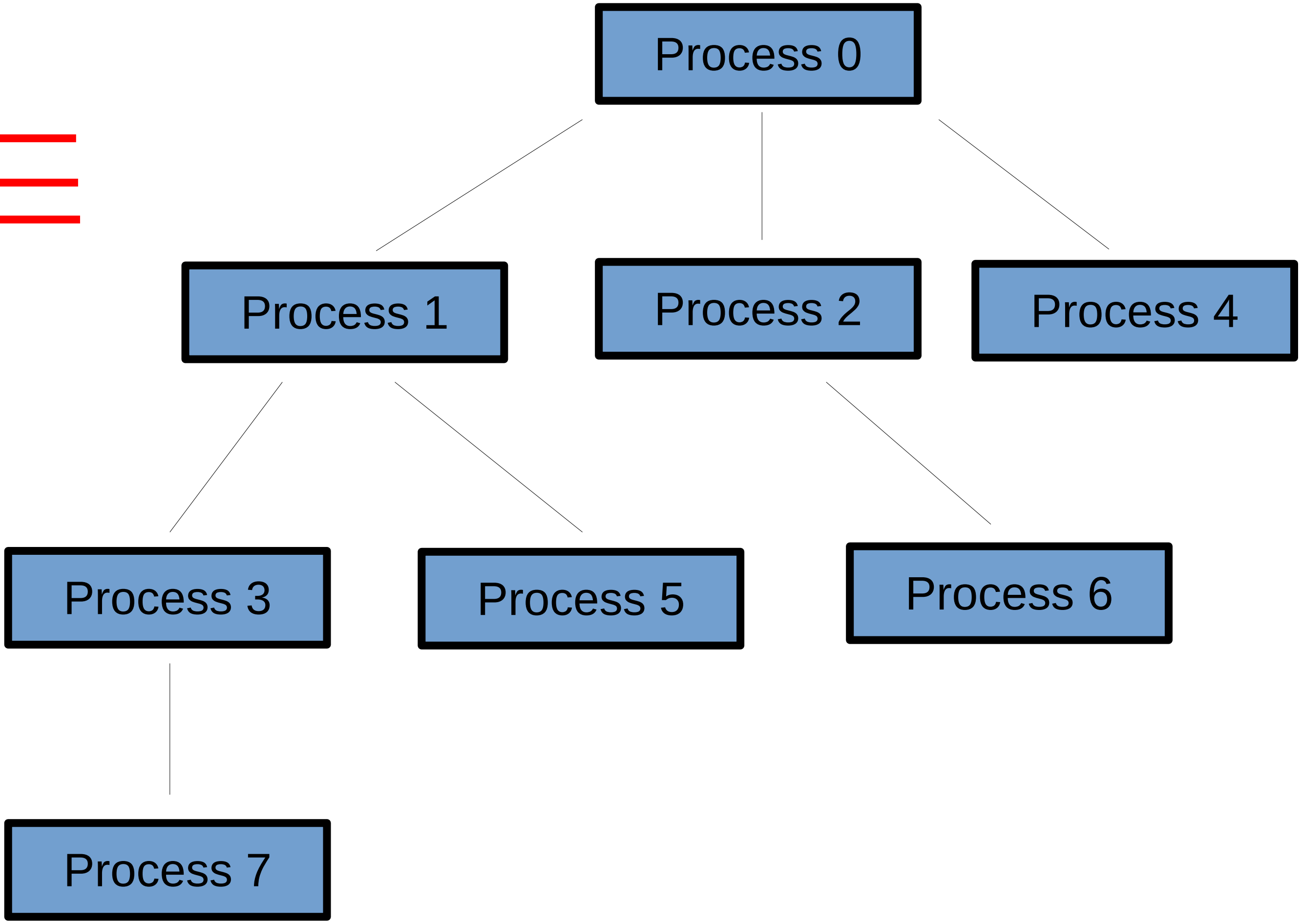
```
user@system:~ gcc -o fork fork.c
user@system:~ ./fork
Hello from process 6196
Hello from process 6198
Hello from process 6197
Hello from process 6199
user@system:~
```

Processes in C

- Very easy to get many different processes running concurrently (and in parallel if you've got the hardware)
- Like using OpenMP for threading, this can mean very few new lines of code
- Can be relatively easy to lose track of how many processes you've created
- Every fork() will create a child process

Processes in C

```c
int main()
{
     fork();
     fork();
     fork();
}
```



Processes in C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    if (fork() == 0)
    {
        printf(“This is the parent process”);
    }
    fork();
    printf("Hello from process %d\n", getpid());
    exit(0);
}
```

parent.c

```
user@system:~ gcc -o parent parent.c
user@system:~ ./parent
Hello from process 13227
This is the parent process
Hello from process 13229
Hello from process 13228
Hello from process 13230
user@system:~
```

We don’t want just copies

- **These models are good and all, but aren’t always suitable**
  - Clients and Servers
  - Hardware simulations
  - Pipelining
- **We need a way of designing completely different processes from the ground up**
- **We could also run very different threads, but typically processes are used for this due to their differing nature**
- **C is not great at this so lets switch to Python (but it is perfectly possible in C!)**

Multiprocessing

- A Python library for creating multiple processes.
- Can be used to create pools of worker processes

```python
import multiprocessing
import time

data = (
    ['A', '2'], ['B', '1'],
    ['C', '3'], ['D', '2']
)

def mp_worker(args):
    print(f"{args[0]} Waiting for {args[1]}s")
    time.sleep(int(args[1]))
    print(f"{args[0]} DONE")

p = multiprocessing.Pool(2)
p.map(mp_worker, data)
```

pool.py

```
user@system:~ python3 pool.py
A Waiting for 2s
B Waiting for 1s
B DONE
C Waiting for 3s
A DONE
D Waiting for 2s
D DONE
C DONE
user@system:~
```

- But also can be used to define many different processes…

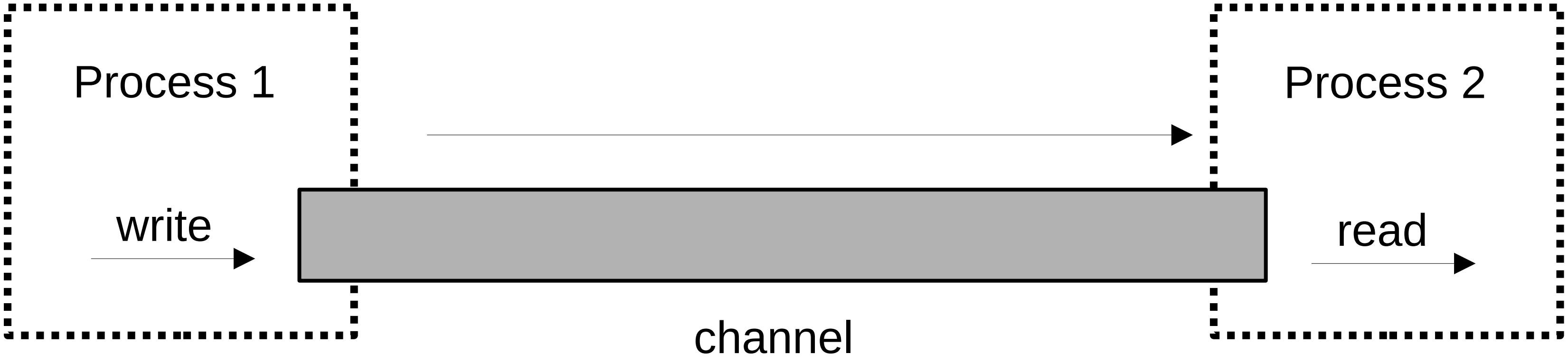
But we have a problem . . .

- **Unlike threads, processes do not (in theory) share any data**
- **This means race conditions aren't a problem** !
- **But this does mean that sharing our data is going to be difficult**
- **2 Strategies to deal with this:**
  - For child processes we can share channels / queues / pipes
  - For independent processes we can use sockets

Channels

- **Channels are the generic term we will use to describe any means for two processes to communicate**

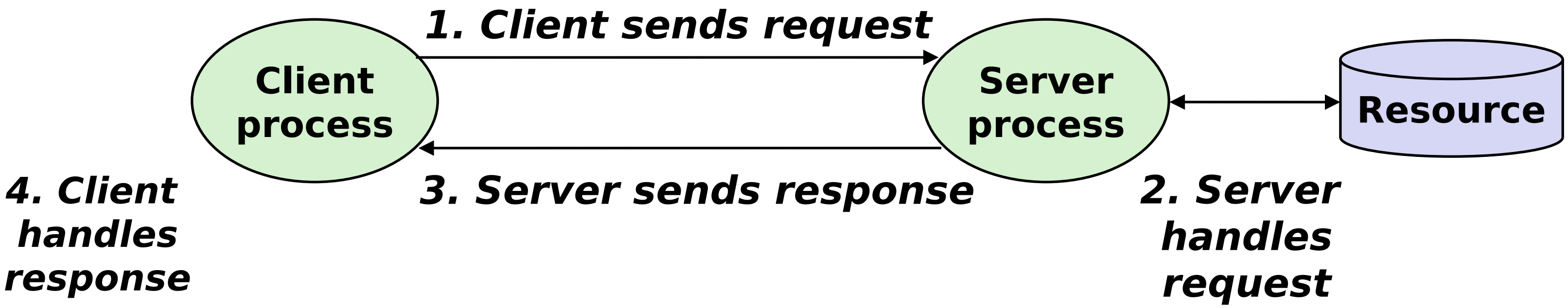
- **Channels are (generally) one way communication tools**



- **Note these can also be used to eliminate race conditions if used instead of shared data (but with a lot of overhead)**

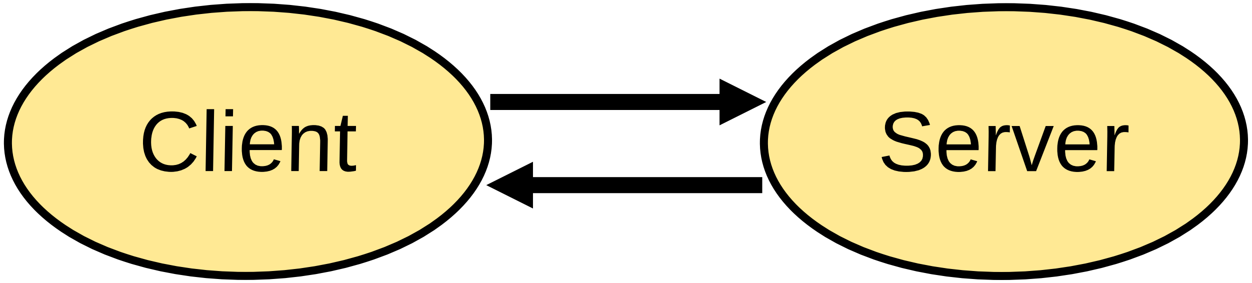
A Client-Server Transaction

- **Most network applications are based on the client-server model:**
  - A ***server*** process and one or more ***client*** processes
  - Server manages some ***resource***
  - Server provides ***service*** by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)



Note: clients and servers are processes running on hosts (can be the same or different hosts)

A Client-Server Transaction



- Here, we are still going to use the client/server model
- Clients send messages, Servers receive them and reply
- Setting different processes up as seperate programs can be handy, but sometimes we might wish to use several in one

A Multiprocessing Client-Server

```
import multiprocessing

PRODUCTION_COUNT = 4

def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {i}")

def consumer(from_producer):
    while True:
        message = from_producer.get()
        print(message)

q = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=(q,)),
    multiprocessing.Process(target=consumer, args=(q,))
]

for p in process_list:
    p.start()
```

```
user@system:~ python3 prod-cons.py
Message 0
Message 1
Message 2
Message 3
```

prod-cons.py

Pipes and Queues

- Two semantically similar constructions for sending messages
- Both are essentially lists that can be added to and removed from
- Queues are much more lightweight but uni-directional
- Pipes are more computationally heavy, but bi-directional
- As we will see in a minute, bi-directionality is actually a problem if used carelessly

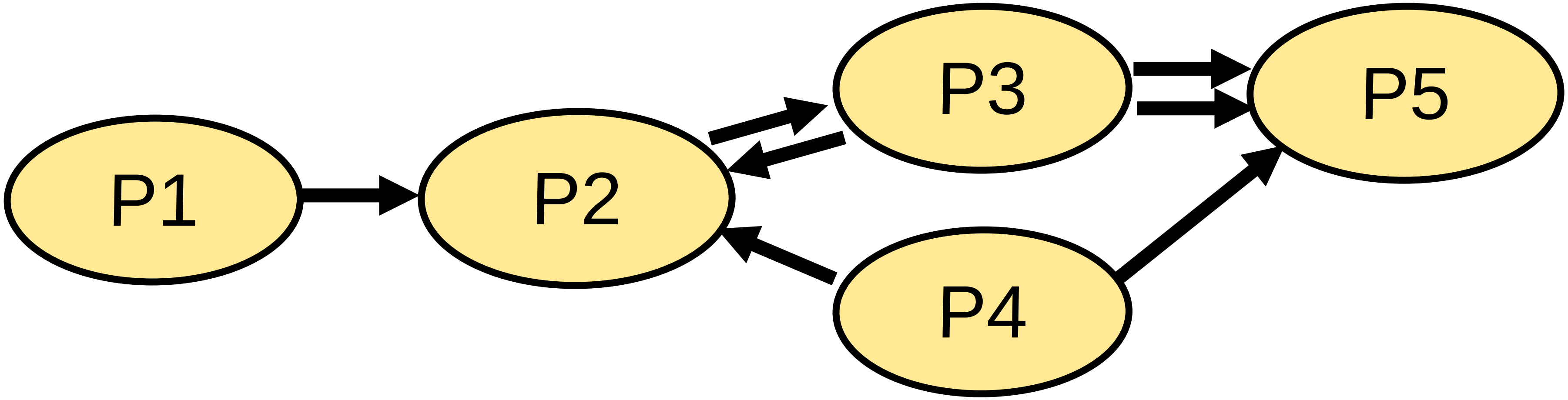
Ending the Client-Server

```
import multiprocessing

PRODUCTION_COUNT = 4
KILL = “kill”

def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {I}")
    to_consumer.put(KILL)

def consumer(from_producer):
    while True:
        message = from_producer.get()
        if message == KILL:
            return
        print(message)

q = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=(q,)),
    multiprocessing.Process(target=consumer, args=(q,))
]

for p in process_list:
    p.start()
```

```
user@system:~ python3 prod-cons.py
Message 0
Message 1
Message 2
Message 3
user@system:~
```

prod-cons.py

A Process Communciations



- **Our example only used 2 processes, but we can use as many as we like and connect them however we like**
- **Of course nothing is ever that easy, we might run into problems if we do this without care**

Blocking points

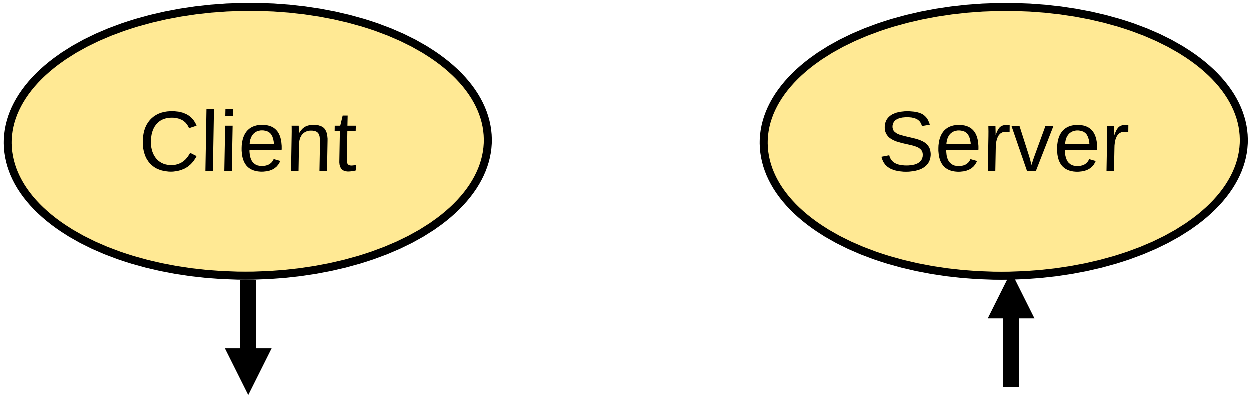
- **Process communication is blocking, that is sequential code will wait until both the client and the server are ready to proceed before it does so.**

Block points

```
...

def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {I}")
    to_consumer.put(KILL)

def consumer(from_producer):
    while True:
        message = from_producer.get()
        if message == KILL:
            return
        print(message)

q = multiprocessing.Queue(1)

...
```

Blocking points

- When drawing processes each blocking communication is marked as a channel either leaving or entering the processes
- Outbound communications leave the process
- Inbound communications enter the process
- Note that commmunications with replies are *usually* not shown separately



Blocking points

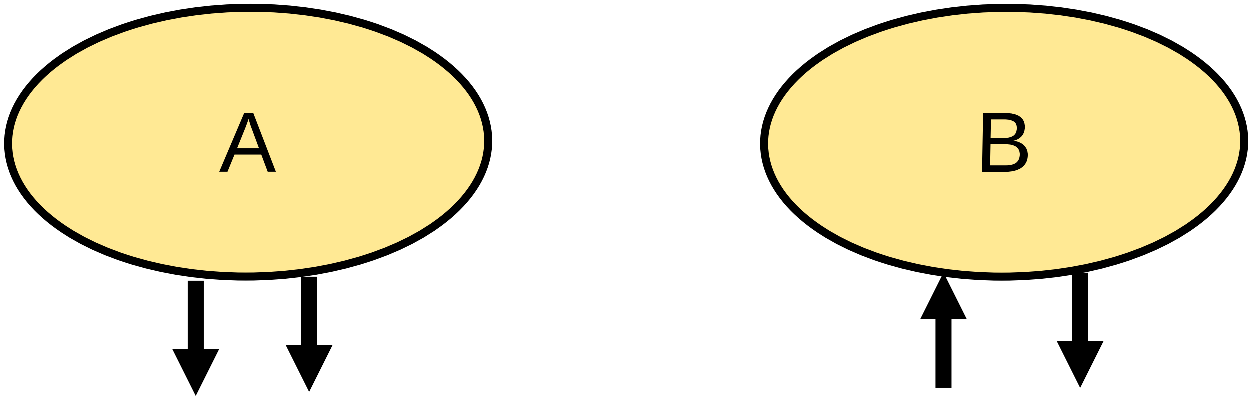
- A single process may have multiple blocking points
- These may each be to/from the same address, or to separate addresses
- They may be any combination of client / server communications

Block points

```
...
def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {I}")
    to_consumer.put(KILL)

def consumer(from_producer):
    while True:
        message = from_producer.get()
        if message == KILL:
            return
        print(message)
...
```

Blocking points

- A single process may have multiple blocking points
- These may each be to/from the same address, or to separate addresses
- They may be any combination of client / server communications

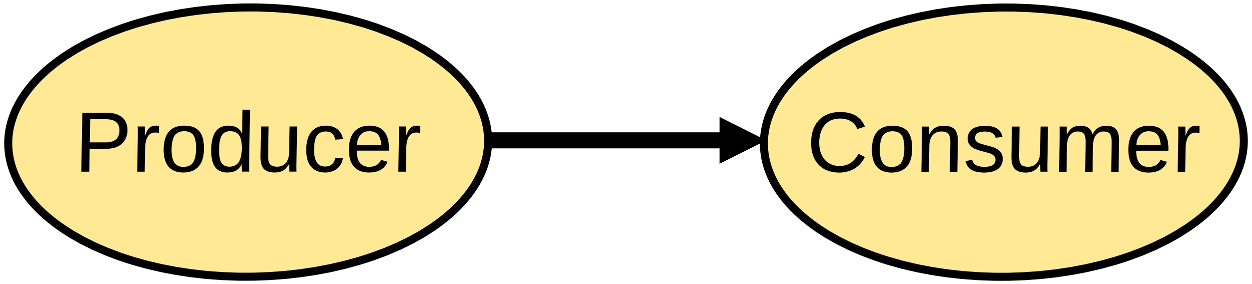


CSP

- Communicating Sequential Processes
- First proposed by Tony Hoare in 1978
- Used as foundation for concurrency in many high level langauges, such as Go
- Implementations in a variety of languages, but mostly outdated, so we won't use it directly

- No shared data
- Processes communicate with each other via channels
- If we map processes communications we can guarantee deadlock free

CSP

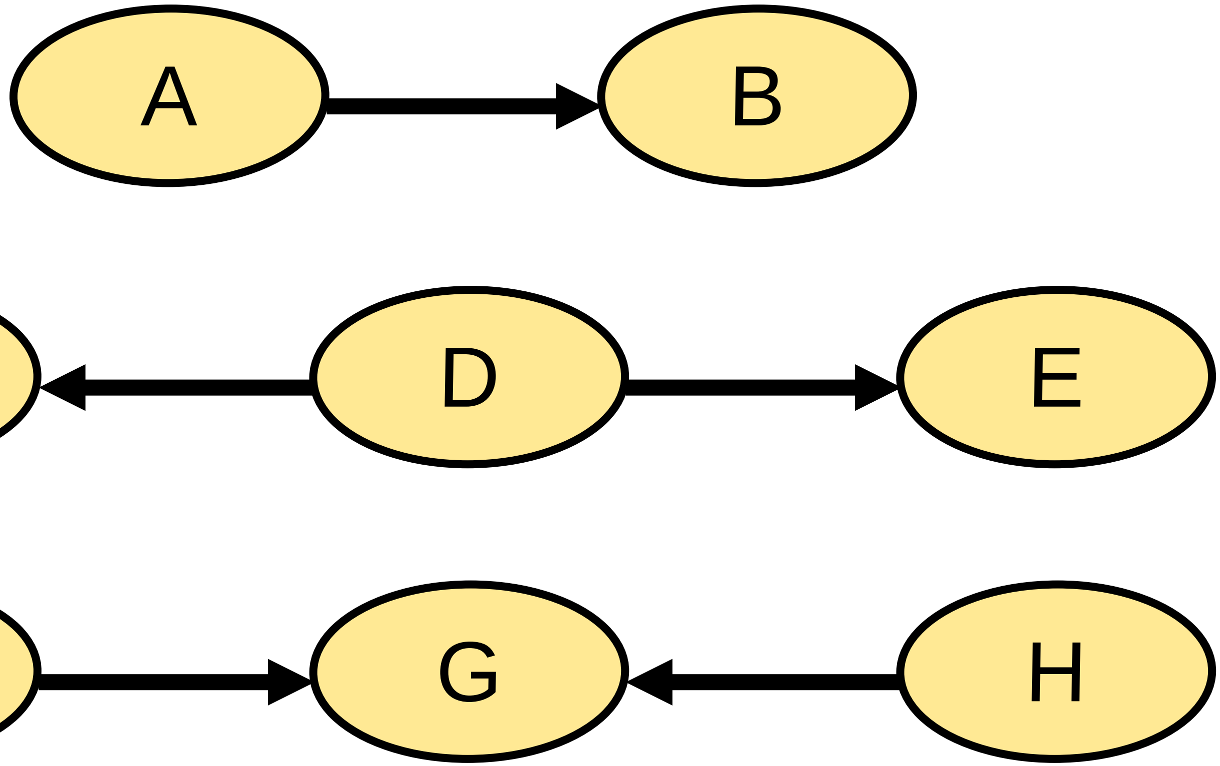


CSP

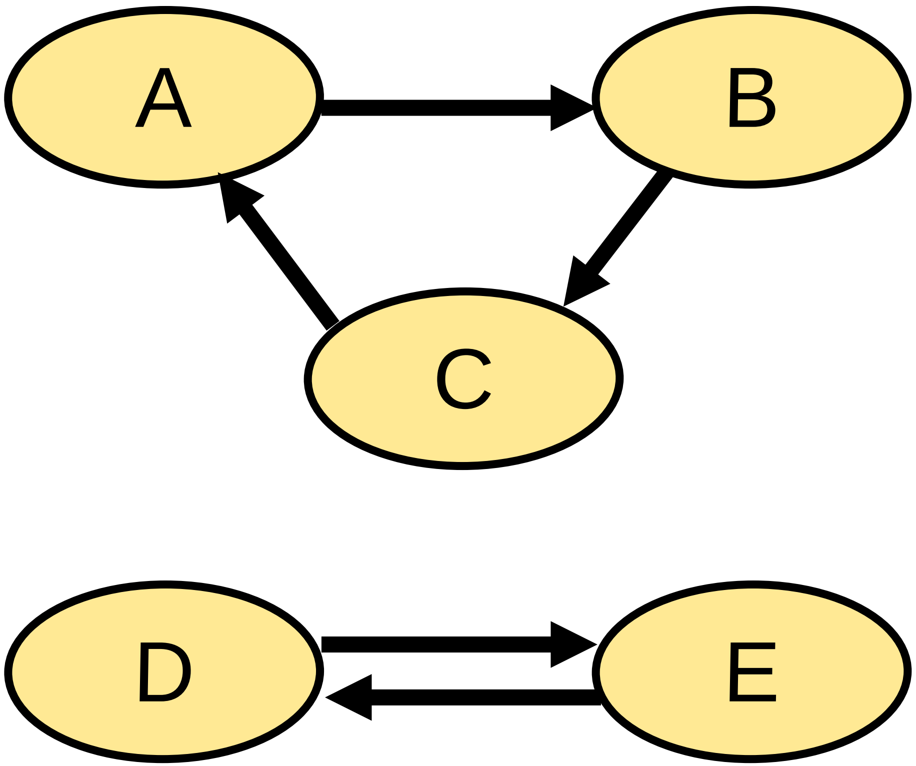
- **If we can map all communications, then we can ensure our design is sound**
- **If we never have a loop of Clients/Servers, we cannot deadlock**
- **This is mathmatically certain according to CSP**
- **Remember, we must assume that if deadlock might occur, it will**

CSP

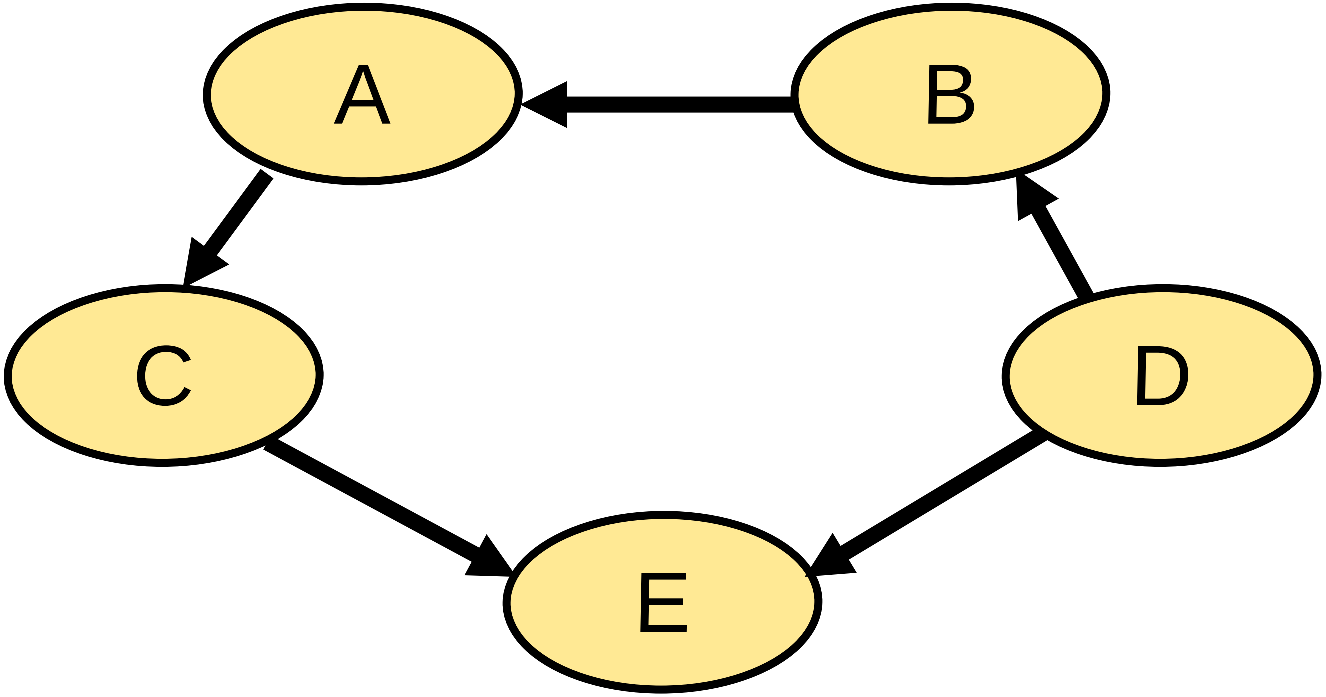
No Deadlock



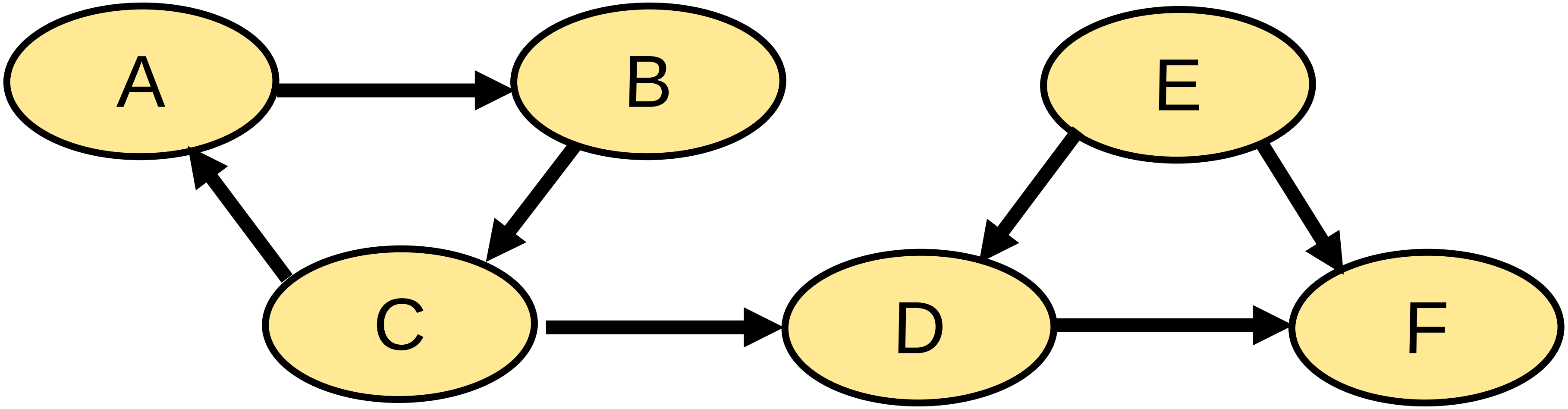
Deadlock



No Deadlock

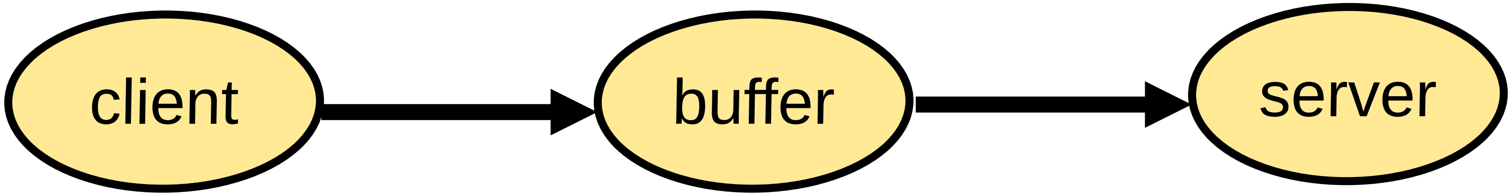


Deadlock



Avoiding Deadlock

- This harsh interpretation is not strictly true (but it is)
- For instance, networking has buffers and additional (semi-hidden) processes
- These are more visible in our multiprocessing example from earlier
- Queues, pipes, and networks almost always act as buffers allowing the client to progress



- This does not alter the conclusion though, its just slower to reach it

Avoiding Deadlock

- Also, these diagrams do not actually mean we will deadlock, just that we might
- If we can avoid deadlock according to the diagram, we know we are deadlock free, guaranteed
- If not deadlock free according to the diagram, we just need to justfiy how we have avoided
- But the road to deadlock is paved with good intentions

Ending the Client-Server

```
import multiprocessing

PRODUCTION_COUNT = 4
KILL = "kill"

def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {I}")
    to_consumer.put(KILL)

def consumer(from_producer):
    while True:
        message = from_producer.get()
        if message == KILL:
            return
        print(message)

q = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=(q,)),
    multiprocessing.Process(target=consumer, args=(q,))
]

for p in process_list:
    p.start()
```

```
user@system:~ python3 prod-cons.py
Message 0
Message 1
Message 2
Message 3
user@system:~
```

This means we don't really 'block'

prod-cons.py

Broadcasting

- Most communication channels are point to point
- They can be shared by many processes (as in mutliple servers may share the same queue)
- But only a single processes will pull a single message.
- If you want a message to be received by multiple processes, it must be sent multiple times.
- Some libraries or channel types will allow a broadcast (one message to multiple servers), but this is not always possible

Broadcasting

```
import multiprocessing
PRODUCTION_COUNT = 4
KILL = “kill”

def producer(to_consumer):
    for i in range(PRODUCTION_COUNT):
        to_consumer.put(f"Message {I}")
    to_consumer.put(KILL)
    to_consumer.put(KILL)

def consumer(from_producer):
    while True:
        message = from_producer.get()
        if message == KILL:
            print(f”{name} killed”)
            return
        print(f”{name}: {message}”)

q = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=(q,)),
    multiprocessing.Process(target=consumer, args=(“A”, q)),
    multiprocessing.Process(target=consumer, args=(“B”, q))
]

for p in process_list:
    p.start()
```

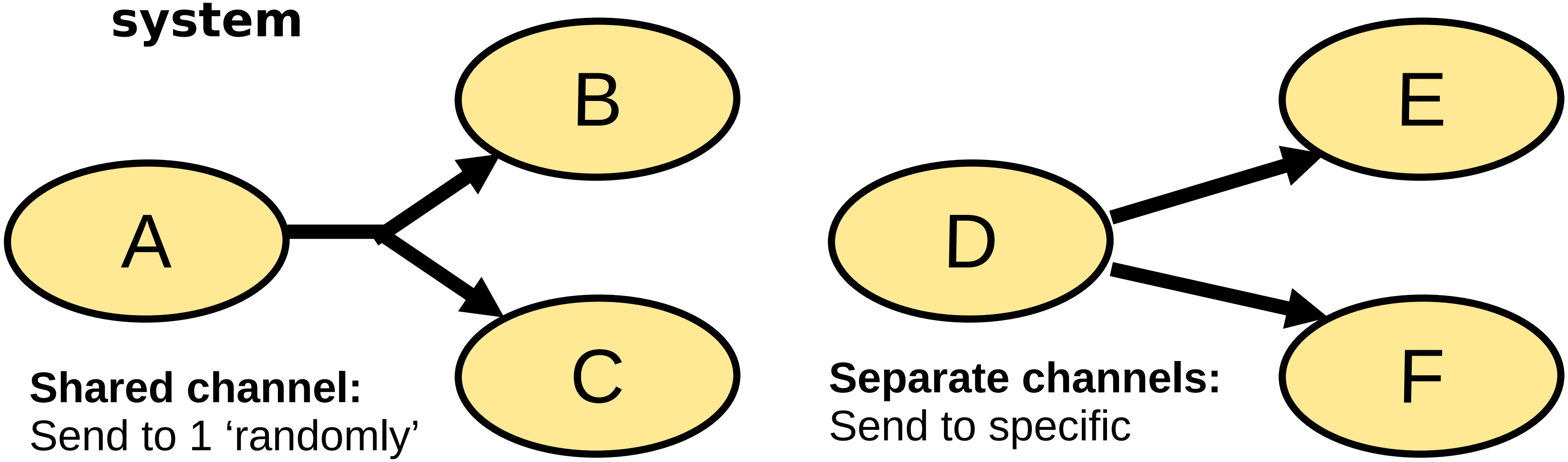
```
user@system:~ python3 mult-cons.py
A Message 0
A Message 1
B Message 2
A Message 3
B killed
A killed
user@system:~
```

Note that on a shared channel, there is no way to specifcally address either consumer

mult-cons.py

Broadcasting, or not

- **However the lack of broadcasting can be a useful feature**
- **Only non-blocking (e.g. free processes) will pick up the message**
- **A useful property for if you want a thread-pool-like system**



Shared channel:
Send to 1 ‘randomly’

Separate channels:
Send to specific

Choice

- **As well as mutliple servers for a single client, we also have the case of multiple clients for one server**

```
message_1 = from_input_one.get()
message_2 = from_input_two.get()
message_3 = from_input_three.get()
```

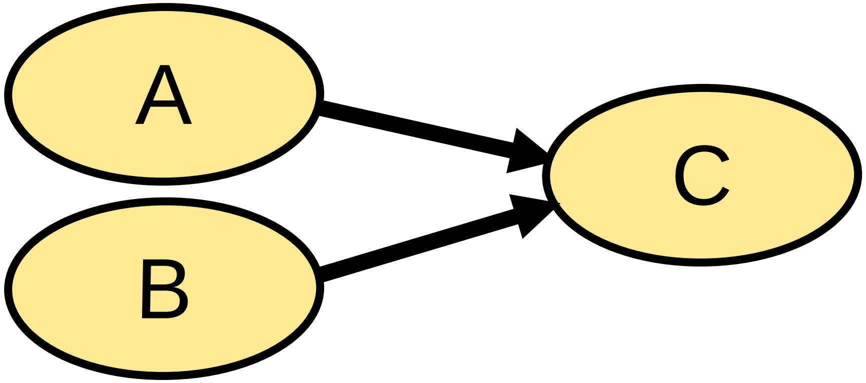
- **We cannot try to read from each in turn, as this is a blocking operation**

- **But we will sometimes need to decide between multiple input channels**

Choice

- We can use Select statements to block until one or more channels are ready to read
- In Python this is some quite dense code

```
import select

read_list = [ … ]
write_list = [ … ]
exception_list = [ … ]

read_ready, write_ready, exception_ready = select.select(
    read_list, write_list, exception_list)
```



Choice

```
import multiprocessing
import select

def producer(name, to_consumer):
    for i in range(3):
        to_consumer.put(f"{name} {I}")

def consumer(in_1, in_2):
    while True:
        (inputs, _, _) = select.select([in_1._reader, in_2._reader], [], [])
        if in_1._reader in inputs:
            message = in_1.get()
        elif in_2._reader in inputs:
            message = in_2.get()
        print(f”{message}”)

q1 = multiprocessing.Queue()
q2 = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=(“A”, q1)),
    multiprocessing.Process(target=producer, args=(“B”, q2)),
    multiprocessing.Process(target=consumer, args=(q1, q2))
]

for p in process_list:
    p.start()
```

```
user@system:~ python3 select.py
A 0
A 1
B 0
A 2
B 1
B 2
```

select.py

Barriers

- As well as just receiving isolated messages from multiple sources, it is often that we want to synchronise on multiple inputs

- For instance, how do we check that several processes are done

- We build a barrier and synchronise on that

Barriers

```
import multiprocessing
import select
import time

def producer(name, sleepy_time, to_consumer):
    print(f"{name} sleep for {sleepy_time}")
    time.sleep(sleepy_time)
    print(f"{name} awoken")
    to_consumer.put(1)

def consumer(in_1, in_2):
    barrier[False, False]
    while True:
        (inputs, _, _) = select.select([in_1._reader, in_2._reader], [], [])
        if in_1._reader in inputs:
            _ = in_1.get()
            barrier[0] = True
        elif in_2._reader in inputs:
            _ = in_2.get()
            barrier[1] = True
        if all(i for i in barrier):
            print("Barrier passed")
            return

q1 = multiprocessing.Queue()
q2 = multiprocessing.Queue()
process_list = [
    multiprocessing.Process(target=producer, args=("A", 1, q1)),
    multiprocessing.Process(target=producer, args=("B", 2, q2)),
    multiprocessing.Process(target=consumer, args=(q1, q2))
]

for p in process_list:
    p.start()
```

```
user@system:~ python3 barrier.py
A sleep for 1
B sleep for 2
A Awoken
B Awoken
Barrier passed
user@system:~
```

barrier.py

Where would we use CSP-like systems?

Networking

- Hopefully we’ve been through this enough already . . .

- All networked applications use this principle already

- Often time higher level network communications such as we used in A4 adds layers of complexity that hide these underlying principles but they’re still there

- Small scale IOT devices and the like won’t have enough resources to abstract them away though, and so they will be central

Simulating Hardware

- **Hardware does not context switch, each component runs both concurrently and in parallel**
- **Therefore a good simulation of this would do the same**
- **These is a very common bachelors/masters projects**
  - Detector simulators (X-Rays, microscopes etc)
  - FPGA systems, processors, GPUs
  - Simulations of pre-production machines/experiments/products

Pipelined Workflows

- **Scientific analysis can often be computed in a isolated, but dependent manner**

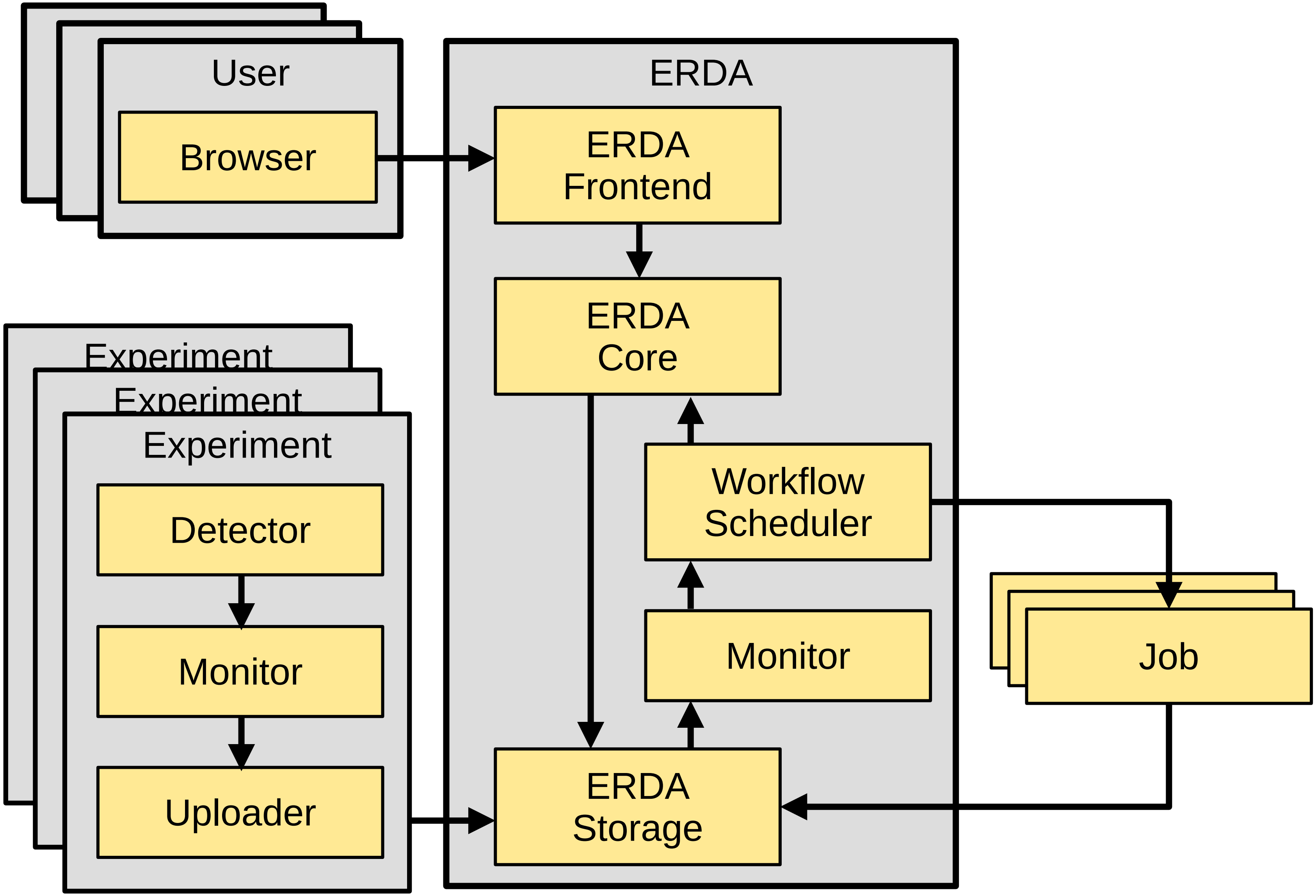
- **These is also a very common use case across all of science**
  - Simulation of physical systems (weather, astronomy)
  - Ongoing analysis

- **Also common in commercial space**
  - Big Data Analysis
  - Social Media
  - Stock Market Analysis

An Example Design

- We won’t look at the code here, it is long, complex, and dull
- This system encompasses many different machines and processes, communicating in a variety of ways
- Used to gather data, analyse the results dynamically and on an ongoing basis
- This system exists in its entirety, but as isolated parts and is being brought together to demonstrate the capability of this design methodology

An Example Design



Conclusions

- **Going beyond simple loops introduces complexity**
- **OpenMP supports variation in looping**
  - Thread IDs can be used to identify individual jobs, elements, tasks, indexes etc etc etc
  - Chunking work is often a very good approach
  - Data parallel!
- **Inter process/thread communication is an alternative**
  - Much more complexity
  - Risks deadlock
  - But can be spread over different nodes/network
  - Task parallel!