FIT3143 Week 3

Flynn's Taxonomy, Processes, Threads, IPC, Shared Memory Communication, Pipelining, Superscalar Processing

Prepared by Marcus, Kevin, Alexis, and Ben

Question 1A:

Discuss the types of parallel systems based on the number of independent instructions and data streams.

Question 1A (Marcus)

- “instruction”=function, “data”=object

- SISD
  - One instruction at a time
  - One data point at a time
- SIMD
  - One instruction controls all cores
  - Each core processes **different data**
- MISD
  - Multiple instructions applied to the **same** data
  - Rare, mostly theoretical or used in **fault-tolerant systems**
- MIMD
  - Each core executes **its own instruction stream**
  - Each core works on **its own data**
  - Most **general-purpose parallel computing** fits here

Question 1B:

For each type of parallel system, name a few examples found in human history and describe a real-life application that can be solved by these machines.

Question 1B (Marcus)

- SISD
  - Older computers only have one core and perform instructions sequentially.
- SIMD
  - GPUs processing pixels
- MISD
  - Cryptography algorithms attempting to crack a single coded message.
  - Independent frequency filters operating on a single signal stream.
- MIMD
  - Modern day computers have multiple cores that perform different jobs over different sets of data at the same time.

Question 2A:

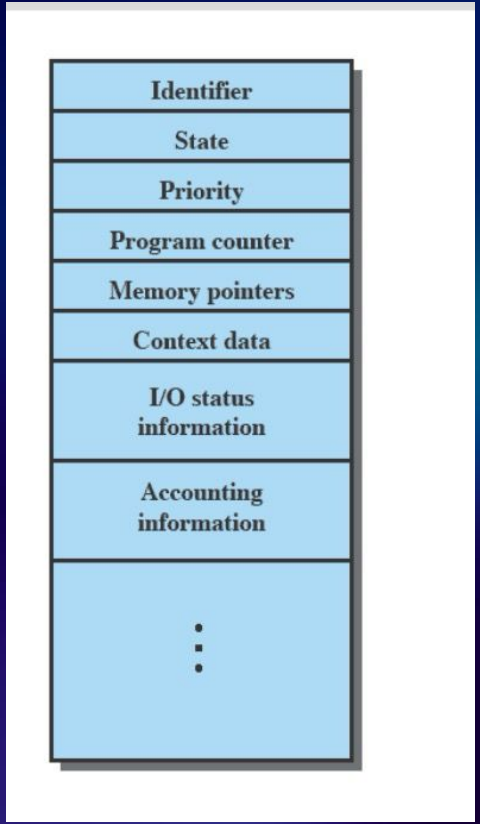
Explain the composition of a process. Discuss at least 6 attributes that must exist for a process so that it can be identified with its state known.

2a - Kevin

A process is a program in execution, containing the program code, a set of data, and all the resources it needs to run. The operating system maintains various attributes so it can identify the process and know its current state.

Six key attributes of the process are:
1. Identifier - unique id for the process

2. State - current status (e.g. new, ready, running, waiting, terminated)
3. Program counter - points to next instruction in a program
4. Memory pointers - addresses of key data in memory required for the program
5. Priority - how prioritised is this process and when should it run (in comparison to other processes)
6. I/O status information - Information about open files, allocated I/O devices, and pending I/O requests.



Question 2B:

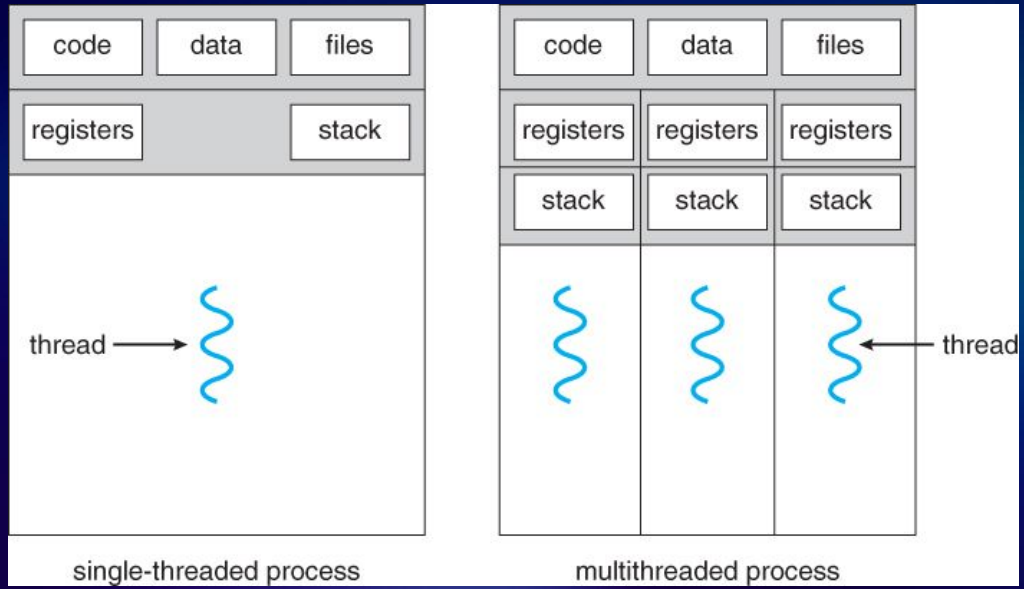
Discuss two resources that are shared between threads within a process, but not shared between processes.

2b - Kevin

Within a process, all threads share certain resources:

1. Address Space (Code and Data Segments) – All threads can access the same program instructions and global variables.
2. Open File Descriptors / Handles – Threads in the same process share open file handles, meaning they can read from or write to the same files without re-opening them.

Processes, on the other hand, do not share memory or file descriptors unless explicitly set up (e.g., shared memory mapping).



Question 2C:

Explain three IPC mechanisms applicable to the same host operating system.

2c - Kevin

Three commonly used Inter-Process Communication (IPC) methods for processes on the same system are:

1. Shared Memory – A block of memory accessible by multiple processes. Fastest IPC method because processes directly read/write to a shared region in RAM. Requires synchronization (e.g., semaphores) to avoid race conditions. Can allow any message structure.
2. Message Passing – OS-managed linked lists of messages. Processes send and receive messages through the queue, with built-in asynchronization. Typically enforces message structure.
3. Unix Signals and analogues – asynchronous signals that are simple and lightweight, but carry very limited instructions. Commonly used for control purposes, e.g. terminating a process. Imposes structure.

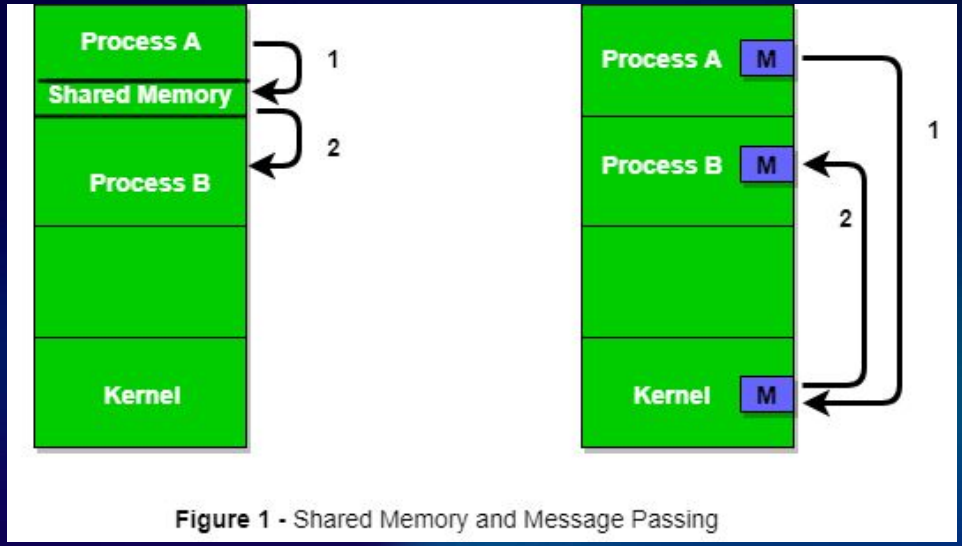
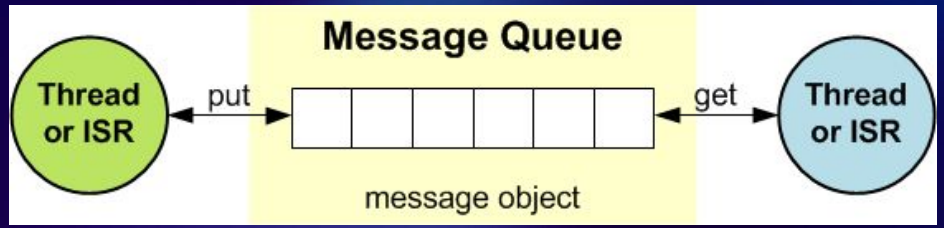


Figure 1 - Shared Memory and Message Passing



Question 2D:

Name one IPC mechanism that is applicable for a network of computers (potentially with different operating systems).

2d - Kevin

Stream-Oriented Inter-Process Communication (IPC)

- Stream-oriented IPC is a common method for communication within operating systems and between networked hosts.
- It sends data as a continuous flow of bytes with no built-in structure, typically received in the order sent (FIFO).
- The two main mechanisms are BSD Sockets and SVR4 STREAMS, both using standard APIs.

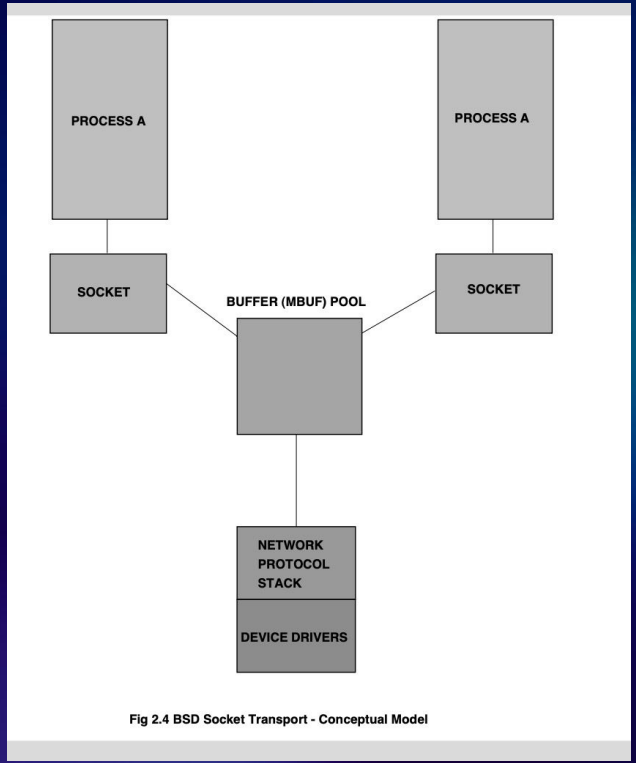


Fig 2.4 BSD Socket Transport - Conceptual Model

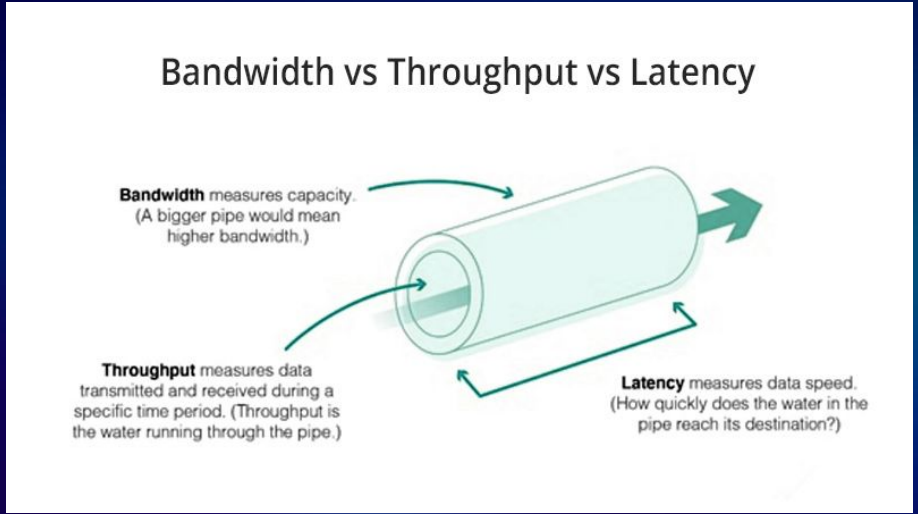
Question 2E:

Explain the two aspects of performance for such a mechanism.

2e - Kevin

1. Latency – The time taken for a message to travel from the sender to the receiver. Lower latency means faster responsiveness. Where data dependencies exist, latency becomes a major problem.

2. Throughput (Bandwidth/Capacity) – The amount of data transmitted successfully over the network per unit time. Higher throughput means more data can be sent efficiently. In many applications even high bandwidth networks may have large latency



Question 3A (Alexis):

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

void *print_message_function( void *ptr )
{
      char *message;
      message = (char *) ptr;
      printf("%s \n", message);
}

int main()
{
      pthread_t thread1, thread2;
      char *message1 = "Thread 1";
      char *message2 = "Thread 2";
      int  iret1, iret2;

     /* Create independent threads each of which will execute function */

      iret1 = pthread_create( &thread1, NULL, print_message_function,
                              (void*) message1);
      iret2 = pthread_create( &thread2, NULL, print_message_function,
                              (void*) message2);

      printf("Thread 1 returns: %d\n",iret1);
      printf("Thread 2 returns: %d\n",iret2);
      return 0;
}
```

Discuss the possible outcomes of the code above.

Question 3A

- There are several possible outcomes, depending on whether one, both or none of the child threads finish executing before the main function returns. In all cases, we can see that both threads have been created successfully
- Case 1: Neither of the child threads finish executing:

```
Thread 1 returns: 0
Thread 2 returns: 0
```

- Case 2: One child thread finishes executing -> the result of print_message_function is printed to the console

```
Thread 1
Thread 1 returns: 0
Thread 2 returns: 0
```

```
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

- Case 3: Both threads finish executing

```
Thread 1
Thread 1 returns: 0
Thread 2
Thread 2 returns: 0
```

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

As we can see, the output may be in any order depending on which process completes first.

Question 3A

- Case 4: Partial execution

```
Thread 1 returns: 0
Thread 2 returns: 0

read 1
Thread 2
```

```
Thread 1 returns: 0
Thread 2 returns: 0

read 1
```

There may be garbled or partial output if the threads start printing but the process exits midway or while flushing output buffers.

Question 3B (Alexis):

Explain the reasoning behind your outcomes. Why?

Question 3B

In the provided code, the main thread does not wait for the child threads to execute completely before exiting (no JOIN statement), resulting in unpredictable behaviour.

Case 1 & 2: When threads are killed before finishing (parent thread finishes executing before the children) we can observe incomplete output.

Case 3: This is our expected output, with both child threads successfully printing to the console before the main function returns,

Case 4: Rarely, we can observe partial output as a result of one or more processes ending before the full output is flushed.

In all cases, we can expect both threads to be created successfully (using pthread_create()), hence we can observe that both threads return 0. We can also see that the output can be in any order depending on which thread is first to finish executing.

Question 4A:

Assume a time phase duration of 1 nanosecond and 4 stages to execute an instruction. Draw and present two diagrams, one without pipelining and one with pipelining, to explain why pipelining would increase the throughput.

Question 4A (Marcus)

After fifteen nanoseconds (T15) we would have only completely executed 4 instructions without pipelining, whereas, with pipelining, we would have completely executed 13 instructions.

Based on the diagram we can see that at all stages of the CPU work is being done, whereas, we are only working at one stage at a time without pipelining.

So generally, we can say pipelining multiplies the throughput by whatever amount of stages we have.

	Stages			
T0	Inst #1			
T1		Inst #1		
T2			Inst #1	
T3				Inst #1
T4	Inst #2			
T5		Inst #2		
T6			Inst #2	
T7				Inst #2
T8	Inst #3			
T9		Inst #3		
T10			Inst #3	
T11				Inst #3
T12	Inst #4			
T13		Inst #4		
T14			Inst #4	
T15				Inst #4

Without pipelining

	Stages			
T0	Inst #1			
T1	Inst #2	Inst #1		
T2	Inst #3	Inst #2	Inst #1	
T3	Inst #4	Inst #3	Inst #2	Inst #1
T4	Inst #5	Inst #4	Inst #3	Inst #2
T5	Inst #6	Inst #5	Inst #4	Inst #3
T6	Inst #7	Inst #6	Inst #5	Inst #4
T7	Inst #8	Inst #7	Inst #6	Inst #5
T8	Inst #9	Inst #8	Inst #7	Inst #6
T9	Inst #10	Inst #9	Inst #8	Inst #7
T10	Inst #11	Inst #10	Inst #9	Inst #8
T11	Inst #12	Inst #11	Inst #10	Inst #9
T12	Inst #13	Inst #12	Inst #11	Inst #10
T13		Inst #13	Inst #12	Inst #11
T14			Inst #13	Inst #12
T15				Inst #13

With pipelining

Question 4B:

What is the throughput speedup and the pipeline latency for such a pipeline system?

Question 4B (Marcus)

The throughput speedup in this case is 4x.

The pipeline latency is 4ns as it takes 4ns to complete inst #1.

	Stages			
T0	Inst #1			
T1		Inst #1		
T2			Inst #1	
T3				Inst #1
T4	Inst #2			
T5		Inst #2		
T6			Inst #2	
T7				Inst #2
T8	Inst #3			
T9		Inst #3		
T10			Inst #3	
T11				Inst #3
T12	Inst #4			
T13		Inst #4		
T14			Inst #4	
T15				Inst #4

Without pipelining

	Stages			
T0	Inst #1			
T1	Inst #2	Inst #1		
T2	Inst #3	Inst #2	Inst #1	
T3	Inst #4	Inst #3	Inst #2	Inst #1
T4	Inst #5	Inst #4	Inst #3	Inst #2
T5	Inst #6	Inst #5	Inst #4	Inst #3
T6	Inst #7	Inst #6	Inst #5	Inst #4
T7	Inst #8	Inst #7	Inst #6	Inst #5
T8	Inst #9	Inst #8	Inst #7	Inst #6
T9	Inst #10	Inst #9	Inst #8	Inst #7
T10	Inst #11	Inst #10	Inst #9	Inst #8
T11	Inst #12	Inst #11	Inst #10	Inst #9
T12	Inst #13	Inst #12	Inst #11	Inst #10
T13		Inst #13	Inst #12	Inst #11
T14			Inst #13	Inst #12
T15				Inst #13

With pipelining

Question 4C:

Discuss two reasons for pipeline stalls.

Question 4C (Marcus)

- A **cache miss**, which is when the CPU fails to find a requested piece of data in the CPU cache, requiring the data to be fetched from a slower storage location like main memory.
- A **data hazard** is when some instruction N+1 is dependent on the result of instruction N, but it may not be the case that N's result is ready yet. One way to handle this is to stall the pipeline until the result is ready for use. Data forwarding is may be a better approach, as stalling impacts performance.
- An additional reason is **conditional branches**, because they always have to wait (stall) until an earlier instruction completes to determine the direction of the following branch.

Question 5A:

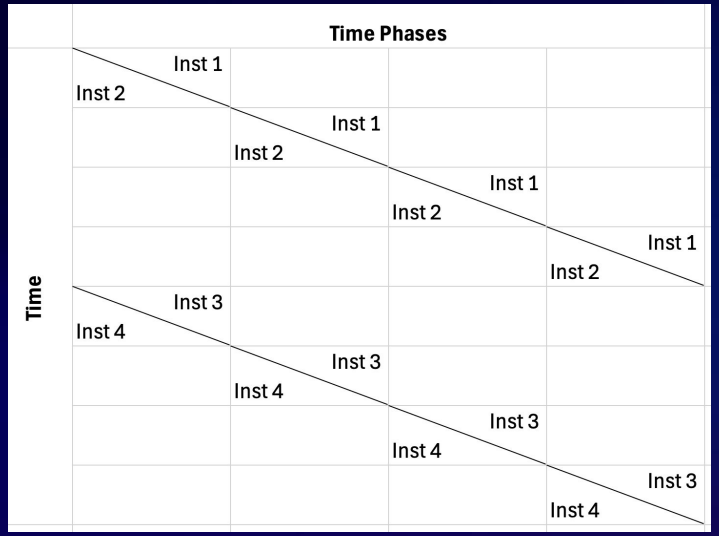
Assume a time phase duration of 1 nanosecond and 4 stages to execute an integer instruction. Assume a 8-way superscalar CPU core has 2 integer Execution Units (EU), 4 floating point (FP) EUs, and 2 Address Arithmetic EUs. Given 4 integer instructions, draw and present a time-phase diagram to explain how the superscalar CPU can speed-up the throughput.

Question 5A - Ben

Question details:
Time phase: 1 ns
Latency: 1*4 = 4ns
EUs: 2 relevant

Superscaling introduction:
- Multiple EUs in a core
- Can increase throughput, if there’s ILP

	Time Phases			
Time	Inst 1			
		Inst 1		
			Inst 1	
				Inst 1
	Inst 2			
		Inst 2		
			Inst 2	
				Inst 2
	Inst 3			
		Inst 3		
			Inst 3	
				Inst 3
	Inst 4			
		Inst 4		
			Inst 4	
				Inst 4



Question 5A - Answer - Ben

Question 5B:

Explain how anti-dependency can slow down superscalar CPU, and give a brief (max 7 coding lines) example in C/C++ to illustrate the idea.

Question 5B - Context - Ben

Issues that can slow down superscalar processing are:

- data dependencies
- **anti-dependencies**
- procedural dependencies
- Or resource dependencies

Anti-dependency:
Exists when we have chronological instructions 1 and instruction 2. And instruction 2 writes to an input operand for instruction 1. Ie, WAR, write after read.

Question 5B - Answer - Ben

Example:

1. `int a = 1, b, total;` #
2. `b = a + 2;` #
3. `a = 4;` # redefining an input to line 2 -> anti-dependency (3 with 2).
4. `total = a + b` #
5. `a = a + 1` # redefining an input to lines 2, 4 -> anti-dependency (5 with 2,4)