Week 10: Inter-Process Communication

Learning Objectives

- Understand inter-process communication mechanisms
- Implement basic IPC examples

Pre-Lab Preparation

Watch the following video lectures before attending the lab session:

- Inter-Process Communication Mechanisms
- Pipes and Redirection in Unix
- Signals and Process Communication
- Review: IPC Theoretical Foundations - test preparation

What you will do today:

- Explore inter-process communication mechanisms
- Implement IPC examples (pipes, signals, file-based communication)
- Address any remaining gaps
- Ensure all evidence is documented

Part 1: Inter-Process Communication Mechanisms

Task 1.1: Pipes and Redirection

1. Understand standard streams:

```
# Standard output
echo "Hello World"

# Redirect output to file
echo "Hello World" > output.txt
cat output.txt

# Append to file
echo "Second line" >> output.txt
cat output.txt
```

2. Redirect standard error:

```
# This command will generate an error
ls /nonexistent 2> error.txt
cat error.txt
```

```
# Redirect both stdout and stderr
ls /nonexistent /etc 2>&1 > combined.txt
cat combined.txt
```

3. Use pipes to connect processes:

```
# Simple pipe
ls -l | grep "txt"

# Chain multiple commands
ps aux | grep "bash" | wc -l

# Process log files
sudo cat /var/log/auth.log | grep "Failed" | tail -10

# Complex pipeline
cat /etc/passwd | cut -d: -f1,3 | sort -t: -k2 -n | tail -5
```

4. Advanced pipeline example:

```
# Find top 10 processes by memory usage
ps aux | sort -k4 -r | head -10
```

For your journal: Document pipeline examples with explanations. Explain how pipes enable process cooperation and data flow between programmes.

Task 1.2: Named Pipes (FIFOs)

1. Create a named pipe:

```
mkfifo mypipe
ls -l mypipe
```

Note the 'p' indicating pipe type

2. In one terminal, write to the pipe:

```
echo "Message through pipe" > mypipe
```

This will block until someone reads

3. In another terminal, read from the pipe:

```
cat mypipe
```

4. Demonstrate bidirectional communication:

```
# Terminal 1
tail -f mypipe
```

```
# Terminal 2
echo "First message" > mypipe
echo "Second message" > mypipe
echo "Third message" > mypipe
```

5. Named pipe with producer/consumer pattern:

```
# Terminal 1 (producer)
for i in {1..10}; do
    echo "Data item $i" > mypipe
    sleep 1
done

# Terminal 2 (consumer)
while read line; do
    echo "Received: $line"
done < mypipe
```

6. Clean up:

```
rm mypipe
```

For your journal: Screenshots showing named pipe communication. Explain the difference between anonymous pipes (|) and named pipes (FIFO), including use cases for each.

Task 1.3: Process Signals

1. List all available signals:

```
kill -l
```

2. Start a process and experiment with signals:

```
sleep 300 &
PID=$!
echo "Process ID: $PID"
```

3. Send SIGTERM (graceful termination):

```
kill $PID
```

4. Start another process and force kill:

```
sleep 300 &
PID=$!
kill -9 $PID  # SIGKILL
```

5. Suspend and resume a process:

```
sleep 300
# Press Ctrl+Z (sends SIGTSTP - suspend)
jobs
```

```
bg  # Resume in background
fg  # Bring to foreground
```

6. Create a signal handling script:

```
nano signal-demo.sh
```

```
#!/bin/bash
# Signal Handling Demonstration

# Function to handle SIGINT (Ctrl+C)
handle_sigint() {
    echo ""
    echo "SIGINT received! Cleaning up..."
    echo "Programme terminated gracefully"
    exit 0
}

# Function to handle SIGTERM
handle_sigterm() {
    echo ""
    echo "SIGTERM received! Shutting down..."
    exit 0
}

# Trap signals
trap handle_sigint SIGINT
trap handle_sigterm SIGTERM

echo "Signal handler running (PID: $$)"
echo "Press Ctrl+C to send SIGINT"
echo "Or use 'kill $$' from another terminal to send SIGTERM"
echo ""

# Main loop
counter=0
while true; do
    echo "Running... (iteration $counter)"
    sleep 2
    ((counter++))
done
```

7. Make executable and test:

```
chmod +x signal-demo.sh
./signal-demo.sh
# Test with Ctrl+C
# Or from another terminal: kill [PID]
```

For your journal: Signal handling script with explanations. Discuss the importance of signal handling in robust applications.

Task 1.4: File-Based IPC Example

1. Create a producer script:

```
nano producer.sh
```

```
#!/bin/bash
# Producer: Writes data to shared file

DATAFILE="shared_data.txt"
LOCKFILE="data.lock"

echo "Producer started (PID: $$)"

for i in {1..10}; do
    # Wait for lock to be available
    while [ -f "$LOCKFILE" ]; do
        sleep 0.1
    done

    # Create lock
    touch "$LOCKFILE"

    # Write data
    timestamp=$(date +"%H:%M:%S")
    echo "[$timestamp] Data item $i" >> "$DATAFILE"
    echo "Produced: Data item $i"

    # Release lock
    rm "$LOCKFILE"

    sleep 1
done

echo "Producer finished"
```

2. Create a consumer script:

```
nano consumer.sh
```

```
#!/bin/bash
# Consumer: Reads data from shared file

DATAFILE="shared_data.txt"

echo "Consumer started (PID: $$)"
echo "Waiting for data..."
```

```
# Wait for file to exist
while [ ! -f "$DATAFILE" ]; do
    sleep 0.5
done

# Monitor file for new data
tail -f "$DATAFILE" &
TAIL_PID=$!

# Run for 15 seconds
sleep 15

# Stop monitoring
kill $TAIL_PID 2>/dev/null

echo ""
echo "Consumer finished"
```

3. Make both executable:

```
chmod +x producer.sh consumer.sh
```

4. Test in separate terminals:

```
# Terminal 1
./consumer.sh

# Terminal 2
./producer.sh
```

5. Clean up:

```
rm shared_data.txt data.lock
```

For your journal: Producer/consumer scripts with explanations. Discuss the challenges of inter-process synchronisation and race conditions.