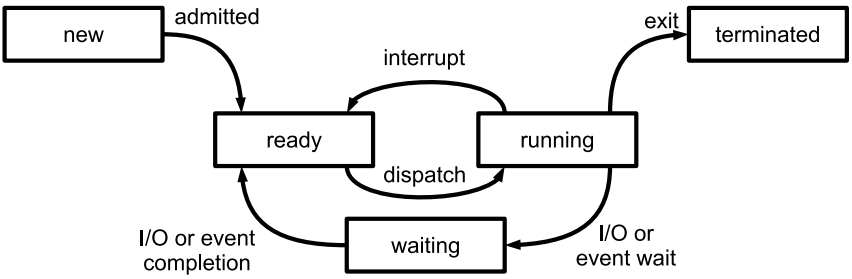
# 159.341 Programming Languages, Algorithms & Concurrency

## Processes - (Part 2)

Daniel Playne
d.p.playne@massey.ac.nz

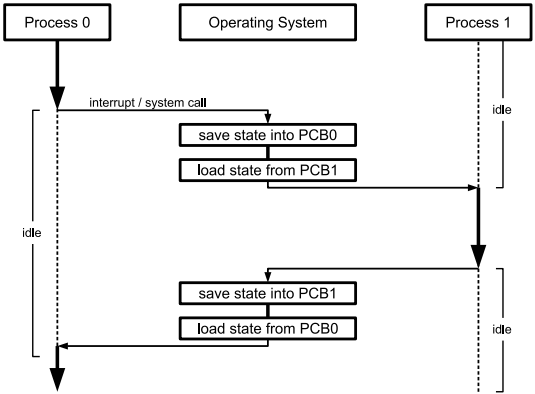
# Process State

The transitions between the different process states are shown below:



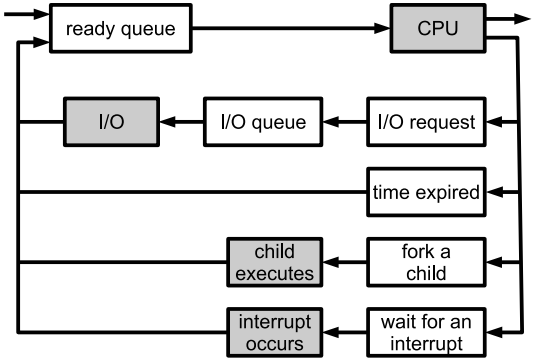

Process state diagram.

# Process Switching




Process switching diagram.

# Scheduling Queues




Queueing diagram.

# Interprocess Communication

Creating ***independent*** processes is straightforward but is limited to performing independent tasks.

Processes are **independent** if they are not affected by the execution of any another processes.

To really make use of multiple processes, we want them to communicate and cooperate on solving the same problem.

# Interprocess Communication

A **cooperating** process can affect (or be affected by) the execution of another process. When two processes share data with each other, they are cooperating.

Reasons for using process cooperation:

- computational speed-up
- modularity
- convenience

# Interprocess Communication

Cooperating process that communication must use an **interprocess communication** mechanism or IPC.

The two fundamental models for IPC are:

- **shared memory** - a region of memory is shared between the two processes.
- **message passing** - communication takes place by sending messages between processes.

# Producer-Consumer

One paradigm for cooperating processes is the **producer-consumer** paradigm. The **producer** is a process that produces information and sends it to the **consumer** that is another process that will read (or consume) that information.

If there is an **unbounded buffer** the producer can send as many items as it likes, the consumer may still have to wait for an item to arrive to consume it.

With a **bounded buffer** there is a maximum number of items a producer can send before it has to wait for the consumer to consume some of them.

# Producer-Consumer - Shared Memory

The shared memory approach to implementing the producer-consumer paradigm will use some area of shared data that both process can access.

## Shared Data

```
typedef ... item;
item buffer[N];
int in, out;
in = 0;
out = 0;
```

# Producer-Consumer - Shared Memory

The producer will produce items and write them to the buffer (if space is available).

## Producer

```
while(true) {
    ...
    // produce an item (next_p)
    ...

    while((in+1)%N == out) {
        // no-operation
    }
    buffer[in] = next_p;
    in = (in+1)%N;
}
```

# Producer-Consumer - Shared Memory

The consumer will consume the next available item (and do something with it)

### Consumer

```
while(true) {
   while(in == out) {
      // no-operation
   }
   next_c = buffer[out];
   out = (out+1)%N;
   ...
   // consume the item (next_c)
   ...
}
```

# Producer-Consumer - Shared Memory

This solution uses **busy-waiting**, the consumer process will continually check the condition over and over until an element from the producer arrives.

```
while(in == out) {
   // no-operation
}
```

The CPU will be doing nothing useful during this time.

# Example - POSIX

POSIX or Portable Operating System Interface is an API (defined by IEEE) for variants of UNIX and other operating systems.

The POSIX API provides methods for communicating through shared memory and message passing.

POSIX shared memory is organised using memory-mapped files that associate a shared memory area with a file.

# Example - POSIX

A shared memory object can be created in POSIX using the following:

```
shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
```

Other processes can access this shared memory by referring to the object by name.

The shared memory can be mapped to a file using the POSIX function `mmap()`.

# Example - POSIX

**Example:**
UNIX Producer-Consumer - Shared Memory

# Pipes

A pipe provides a connection between two different processes. From the point of view of the processes, the pipe appears to be just like a file.

When a process writes something, it is buffered in the pipe. When a process reads from the pipe buffer (or blocks if there is no input).

# Pipes

UNIX (and DOS) provides a mechanism for piping input and output from one process to another using the | character.

```
cat data.txt | sort | more
```

The `cat` command will print the contents of the file "data.txt", this output is piped to the `sort` command which will sort the list and this output is then piped to `more` which will print the output one screen at a time.

# Producer-Consumer - Message Passing

Another solution is to use a message-passing system to allow processes with no shared address space to communicate.

Must have at least these two operations
`send(message)`
`receive(message)`

Messages can be either a fixed or variable size but a logical **communication link** must exist between the processes.

# Message Passing

There are a range of different methods for implementing communication links between process, we are more concerned with the logical implementation.

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

# Message Passing

**Direct communication** - each process explicitly names the sender/recipient of the message. Under this scheme the `send()` and `receive()` functions are:

- `send(P, message)` - send a message to process $P$.
- `receive(Q, message)` - receive a message from $Q$.

In this scheme there is exactly one link between each pair of communicating processes.

# Message Passing

This scheme uses **symmetric** addressing - both the sender and receiver identify the other process.

A variant of the scheme uses asymmetric addressing - the sender identifies the receiver but the receiver can query which process sent the message.

- `send(P, message)` - send a message to process $P$.
- `receive(id, message)` - receive a message from id and set id to the name of the sender.

# Message Passing

**Indirect communication** - messages are not sent to processes but to **mailboxes** or **ports**. Mailboxes can be viewed as an object that messages can be placed into and removed from.

Each mailbox has a unique identifier which processes can use to communicate. Processes can communicate through several different mailboxes.

# Message Passing

The `send()` and `receive()` functions are implemented as:

- `send(A, message)` - send a message to mailbox *A*.
- `receive(A, message)` - receive a message from mailbox *A*.

In this scheme, mailboxes provide a link between two or more processes and processes may communicate through one or more mailboxes.

# Message Passing

Another option for implementing communication functions is the type of **synchronisation** used - either **blocking** or **nonblocking**.

- **blocking send** will halt the sender until the message has been received by either the receiving process or mailbox.
- **nonblocking send** will send the message and the sender will continue execution.

# Message Passing

There are similar functions for receive.

- **blocking receive** the receiver will halt until a message has been received.
- **nonblocking send** will immediately return either a valid message or null.

Various combinations of `send()` and `receive` are possible.

# Message Passing

**Buffering** Whether communication is direct or indirect, messages need to be stored in some kind of temporary storage queue. There are three options for these queues:

- **Zero capacity** - the queue has a length of zero. Can only be used when the sender blocks until the recipient receives the message.
- **Bounded capacity** - the queue has a finite length, the sender can send messages until the queue is full at which point it must block until there is sufficient space available to send the next message.
- **Unbounded capacity** - the queue has infinite length, the sender can send any number of messages without blocking.

# Producer-Consumer - Message Passing

The producer-consumer paradigm can be solved in a variety of ways depending on the implementation of the message passing system.

The solution is rather straightforward when blocking communication is used.

**Producer** simply produces an item and sends it to the consumer (blocks until the consumer receives it).

**Consumer** waits for an item to arrive and then consumes it.

# Interprocess Communication

## Producer

```
message next_p;
while(true) {
   // produce an item in next_p
   ...

   send(next_p);
}
```

## Consumer

```
message next_c;
while(true) {
   receive(next_c);

   // consume the item in next_c
   ...
}
```

# Message Passing Interface - MPI

The **Message Passing Interface** or MPI is a message passing standard designed for a range of parallel computing applications.

MPI can be used to pass messages between processes running on different machines, but we can also use it to pass messages between processes on the same system.

Commonly used by cluster and supercomputers.

# Message Passing Interface - MPI

Some important MPI functions:

| | |
|---|---|
| `MPI_Init` | Initialise MPI |
| `MPI_Comm_size` | Number of processes |
| `MPI_Comm_rank` | Process identifer |
| `MPI_Send` | Send a message to a process |
| `MPI_Recv` | Receive a message from a process |

# Summary

- Interprocess communication
- Shared memory, pipes, message passing
- Producer-consumer model