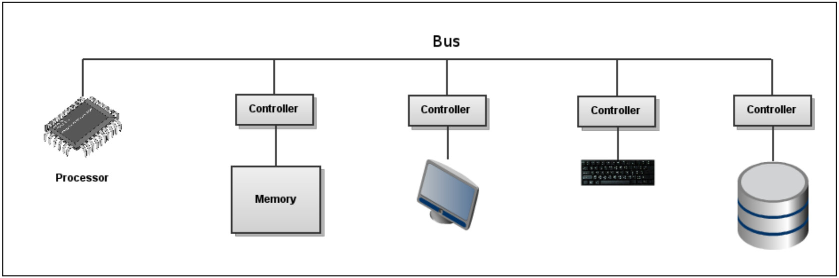
CI583: Data Structures and Operating Systems
Basic OS Concepts

Outline

Input/Output architectures

How does the OS coordinate the activity of the various I/O devices? A simplistic setup, where the bus is a subsystem for transferring data:



In reality, there are several buses with specialised purposes.

Input/Output architectures

The details of device management are complicated and hardware-specific – no interest to us. We will consider an idealised memory-mapped architecture.

Each device has a controller and each controller has a set of registers for managing the operation of the device. As far as the processor is concerned, these registers are memory locations.

Input/Output architectures

Each controller is connected to the bus.

When the processor wants to send a message to a device it broadcasts a memory/register address on the bus.

The appropriate device picks up the message and decodes the task that has been sent to it: read data from a particular location, write to memory etc.

Input/Output architectures

There are two kinds of devices in our idealised architecture: programmed I/O (PIO) and direct memory access (DMA).

PIO devices do I/O by reading and writing data in the controller registers one byte at a time.

In DMA devices, the controller performs the I/O itself: the processor sends a description of the task to be performed over the bus, then the controller takes over.

An example of a PIO device is a terminal or keyboard. An example of a DMA device is a hard disk.

Input/Output architectures

PIO devices have four one-byte registers: *Control*, *Status*, *Read* and *Write*.

Setting certain bits in the *Control* register indicates the task the processor wants done.

The controller sets bits in the *Status* register to indicate whether it is ready, busy, etc.

The *Read* and *Write* registers are used to transfer data.

Input/Output architectures

Programmed I/O

GoR	GoW	IER	IEW					Control
RdyW	RdyR							Status
								Read
								Write

GoR: Start a read op
GoW: Start a write op
RdyR: Ready to read

IER: Enable read-completion interrupts
IEW: Enable write-completion interrupts
RdyW: Ready to write

Adapted from Doeppner, *Operating Systems in Depth*.

Input/Output architectures

To write one byte to, say, the terminal, the processor does the following (the details of which are encapsulated in device drivers):

1. Store the byte in the *Write* register.
2. Set the GoW and IEW bits in the *Control* register.
3. When the device sends an interrupt, the write is done.

Laborious, but the controller is very simple.

Input/Output architectures

A DMA controller also has four registers: *Control*, *Status*, *Memory Address* and *Device Address*.

The first two are one-byte and work in the same way as for PIO.

The rest are four-bytes long and contain memory locations.

In DMA, the controller rather than the processor, does the work.

Input/Output architectures

To write the contents of a buffer to disk, for example:

1. Set the disk address in the *Device Address* register.
2. Set the buffer address in the *Memory Address* register.
3. Set the appropriate bits in the *Control* register to indicate the task to be done and the fact that the processor wants an interrupt when it completes.

The controller will then carry out the whole operation and send an interrupt when ready. Less laborious, but the controller is more complex.

Dynamic storage

Whilst all this context switching is taking place, memory is being allocated and freed at a massive rate, as the contexts required by threads, interrupts and so on are stored and discarded.

It is essential that this takes place as efficiently as possible, both with regard to time and the best use of available storage.

Two common algorithms for allocating storage are best-fit and first-fit.

Dynamic storage

To allocate storage for K bytes of data:

- Best-fit: allocate the smallest free block $>= K$. This is slow, as we need to check all available free blocks. Leads to the creation of many small blocks which might not be much use, or external fragmentation.
- First-fit: allocate the first free block $>= K$.
  Counter-intuitively, this generally leads to less fragmentation.

Dynamic storage
The Buddy System

The Buddy System (Knuth, 1968) is a dynamic storage scheme in which the size of blocks all blocks is some power of two.

Requests for storage are rounded up to the nearest power of two, p. If a block of size p is free, it's taken.

Otherwise, we take the smallest block larger than p and split it in two – the two halves are called buddies.

Dynamic storage
The Buddy System

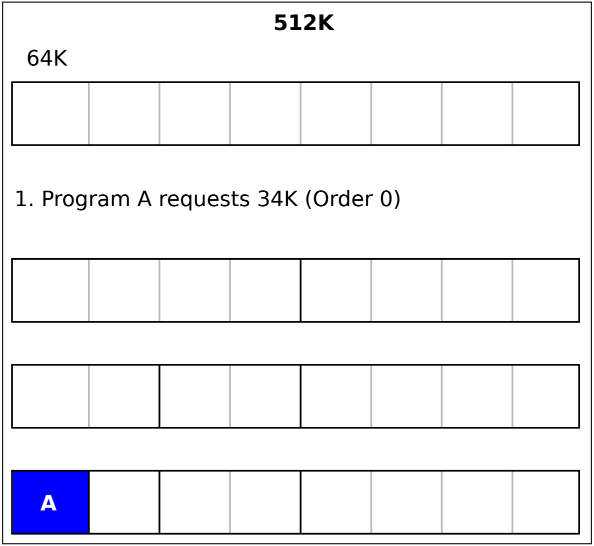
If the buddies have size p, one of them is taken.

Otherwise, we take one buddy, split it again and continue until a block of size p is reached.

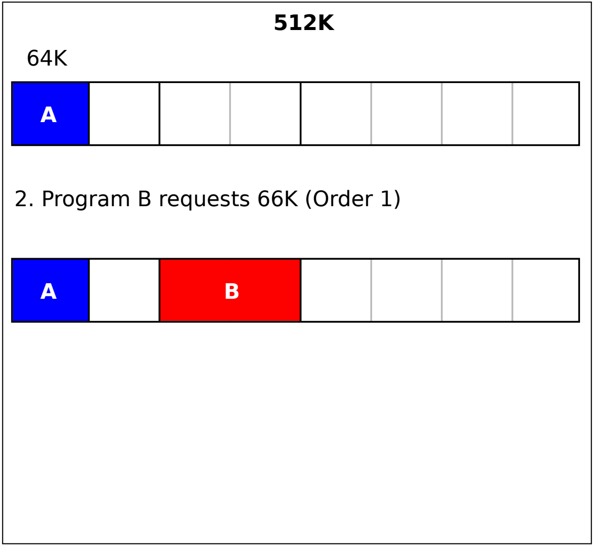
When freeing space, if the buddy of the block to be freed is free, join them together.

If the buddy of the new block is free, join them and keep going until the largest possible block is formed.

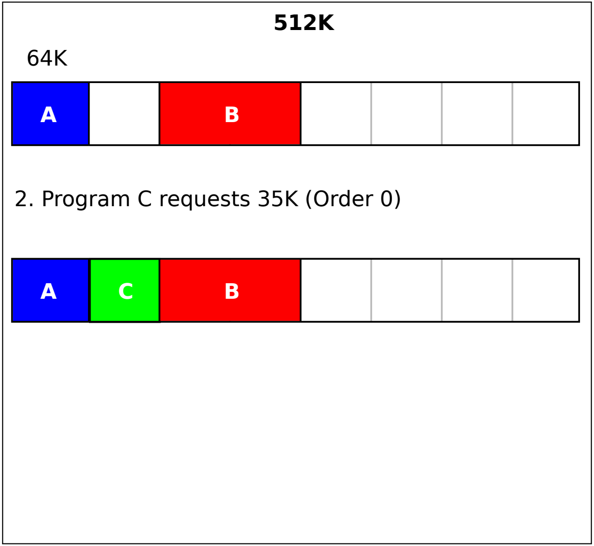
The Buddy System



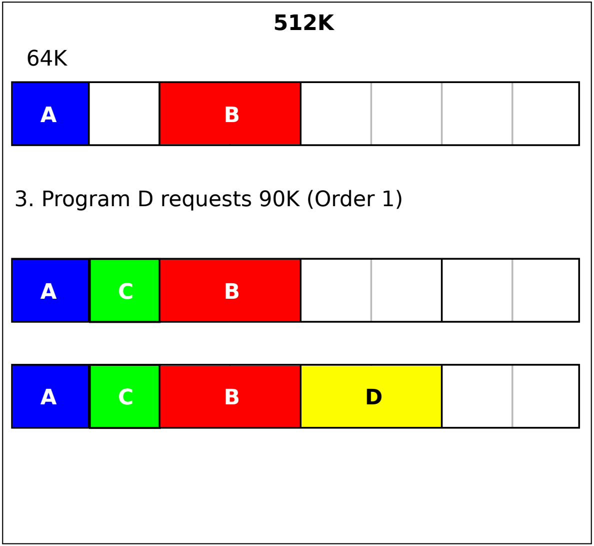
The Buddy System



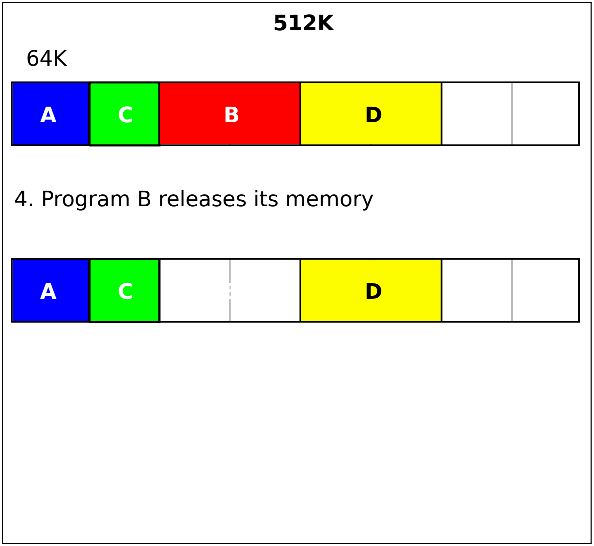
The Buddy System



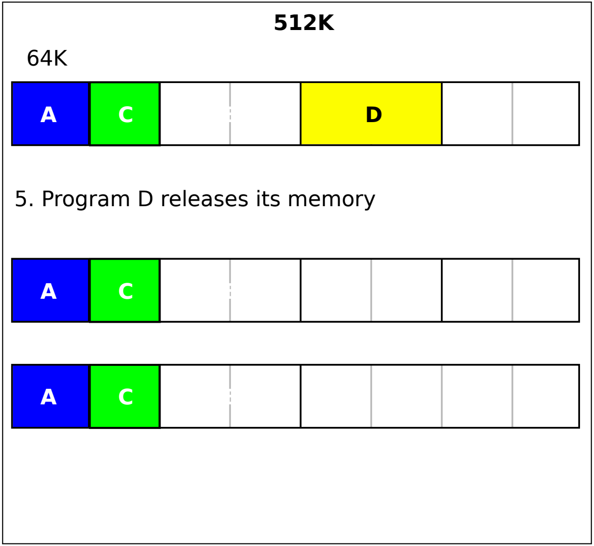
The Buddy System



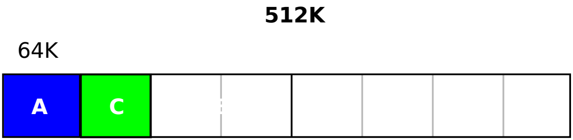
The Buddy System



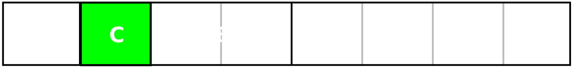
The Buddy System



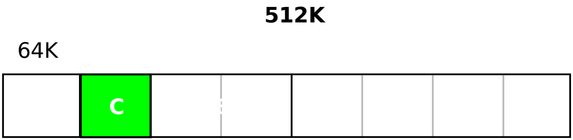
The Buddy System



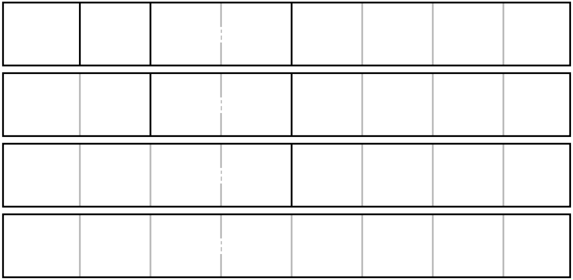
6. Program A releases its memory



The Buddy System



7. Program C releases its memory



The Buddy System

Allocating and deallocating memory using the Buddy System is fast – the system can be represented as a binary tree.

However, internal fragmentation can be high if the amount of storage is requested is slightly larger than a small block but much smaller than the next size up (e.g. 65K, in our example).

The Linux kernel uses the Buddy System with modifications to reduce internal fragmentation.

Next week

Next week we'll be looking at principles of OS design – monolithic kernels versus micro-kernels, and issues around virtualisation.