Parallel Computing with GPUs

Memory
Part 3 – Dynamically managed memory

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

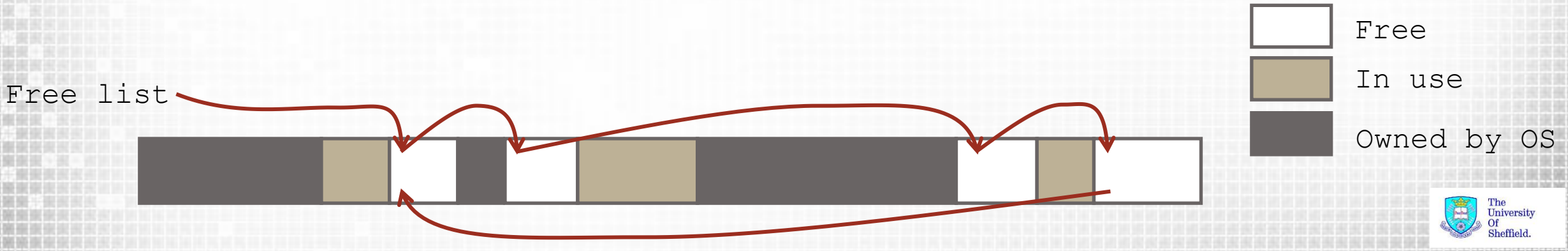
- Dynamically managed memory
  - Perform manual allocations and deletions of memory on the heap
  - Identify scenarios which may result in memory leaks
  - Operate on blocks of memory using library functions

Reminder: Heap vs. Stack

- Stack
  - Memory is managed for you
  - When a function declares a variable it is pushed onto the stack
  - When a function exists all variables on the stack are popped
  - Stack variables are therefore local
  - The stack has size limits
- Heap
  - You must manage memory
  - No size restrictions (except available memory)
  - Accessible by any function

Dynamically allocated memory

- What if we can't specify an array size at compile time (static allocation)
  - The size might not be known until runtime
- We can use the `malloc` system function to get a block of memory on the heap.
  - `malloc` keeps a list of free blocks of memory on the heap
  - `malloc` returns the first free block which is big enough "e.g. first fit"
  - If a block is too big it is split
    - Part is returned to the user and the remainder added to the free list
  - If no suitable block is found `malloc` will request a larger block from the OS
    - Increases the size of the heap
    - Adds the new memory to the free list (flagged as in use)



malloc

- `void *malloc(size_t size)`
  - Returns a pointer to void which must therefore be cast

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

void main()
{
    int *a;
    a = (int*) malloc(sizeof(int) * 10);
}
```

- Use `sizeof` function to ensure correct number of bytes per element
- `a` can now be used as an array (as in the previous examples)
- Result of `malloc` will be implicitly cast (explicit cast is good practice)
  - Implicit cast generates a warning

Memory leaks

- Consider the following
  - `b` is on the stack and is free'd on return
  - `a` points to an area of memory which is allocated
  - `a` then points to `b`, there is no pointer to the area of memory that was allocated

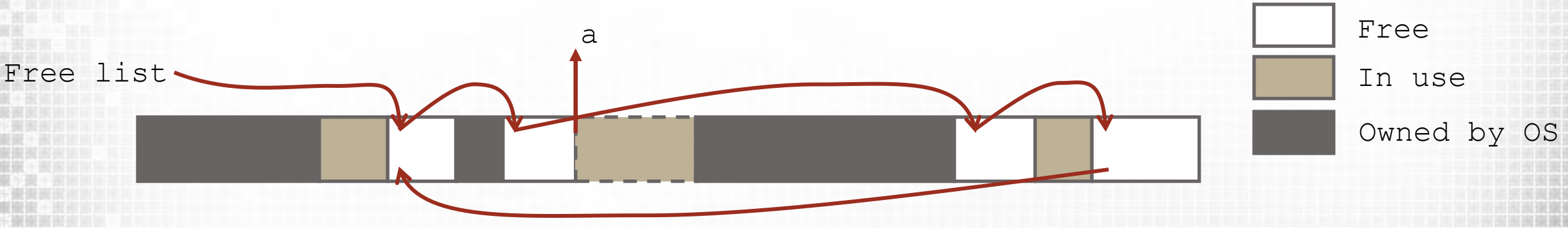
```
void main()
{
    int b[10] = {1,2,3,4,5,6,7,8,9,10};
    int *a;
    a = (int*) malloc(sizeof(int) * 10);
    a = b;

    return;
}
```

- This is known as a memory leak
  - Where we allocate memory we must also free it

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void *);`

```
int *a = (int*) malloc(sizeof(int) * 10); //allocate
free(a);//free
```



free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void *);`

```
int *a = (int*) malloc(sizeof(int) * 10); //allocate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void *);`

```
int *a = (int*) malloc(sizeof(int) * 10); //allocate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void *);`

```
int *a = (int*) malloc(sizeof(int) * 10); //allocate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void *);`

```
int *a = (int*) malloc(sizeof(int) * 10); //allocate
free(a);//free
```

Memory operations

- Set a block of memory to char value
  - `void *memset(void *str, int c, size_t n)`
    - Can be used to set any memory to a value (e.g. `0`)
    - Useful as allocated memory has undefined values

```
int *a;
int size = sizeof(int) * 10;
a = (int*) malloc(size);
memset(a, 0, size);
```

- Coping memory
  - `void *memcpy(void *dest, const void *src, size_t n)`
    - Copies `n` bytes of memory from `src` to `dst`

```
int *a;
int b[] = {1,2,3,4,5,6,7,8,9,10};
int size = sizeof(int) * 10;
a = (int*) malloc(size);
memcpy(a, b, size);
```

Summary

- Dynamically managed memory
  - Perform manual allocations and deletions of memory on the heap
  - Identify scenarios which may result in memory leaks
  - Operate on blocks of memory using library functions

- Next Lecture: Structures and binary files