Memory Management II

an OS view

Dr Alun Moon

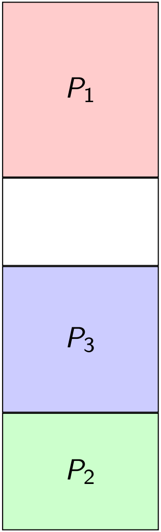
Computing, Engineering and Information Sciences

1st November 2011

Processes in memory

Memory Architecture

- Processes are in non-overlapping places in physical memory
- Managed by Operating System
- Processes must not interfere with each other
- Memory must be protected from *rogue* processes trying to write outside of their allocated memory

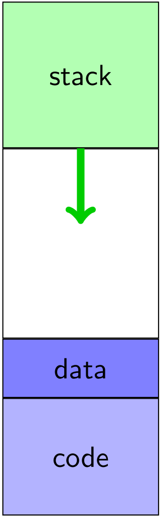


Process memory

View from a single process

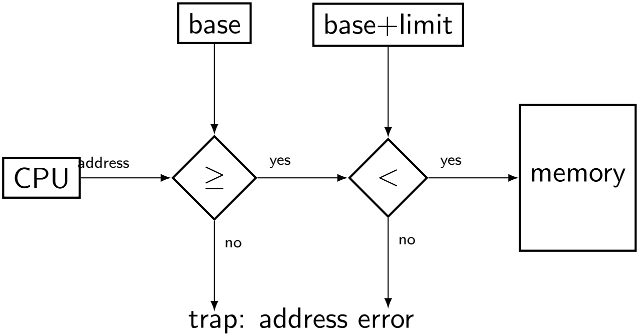
Each process is structured

- **code** (historically called text) is the process code from the program (includes constants).
- **data** Data used by the process, *c.f.* static
- **stack** Each process has it's own stack for function calls and local variables.



Memory Protection

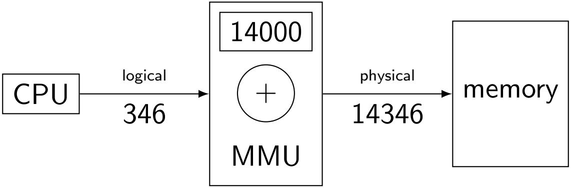
The CPU has a pair of registers to check the address of every memory operation.



The Kernel loads these with the base and limit for each process as part of the process context switch.

Memory Relocation

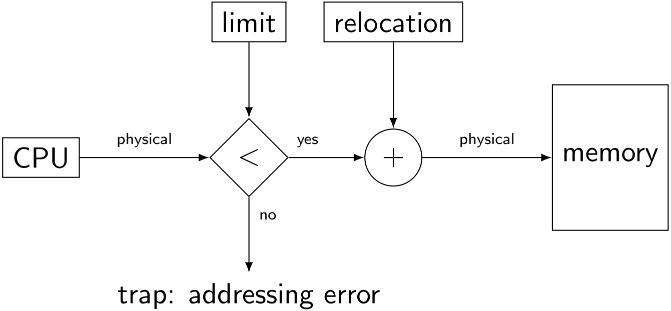
A Memory Management Unit (MMU) can provide a mapping between a *logical* address and a *physical* address.



The process can now exist as if it was in its own address space $0 \ldots n$ no matter where in memory it physically is.

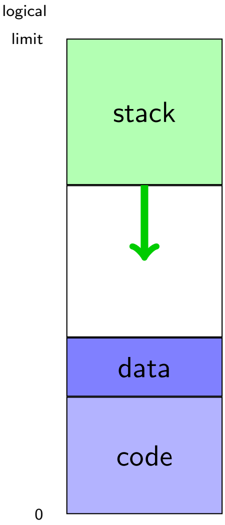
Memory mapping and Protection

The Relocation and Protection can be combined into a single opertation.



Process Memory Management

- The OS now maintains a base address and a size limit for every process.
- These are loaded into the MMU during the context switch
- Processes are protected from each other
- Exist in their own *logical* address space.



Shared code

Multiple instances of a process

- the code can be shared between processes, only one copy is neeeded.
- each process has it's own memory, the context switch and MMU manage the mapping to physical memory.
    - The data and stack can be separated
    - the pieces do not even have to be adjacent.
- the OS maintains a table of process with the addresses and limits of the sections in memory

Relocatable Modules

- We now have a block of code that can sit anywhere in memory
- addressing is resolved
  - internally (static)
  - externally (system managed)
- it has a table of entry points corresponding to function calls
- the loader can assemble several modules into a processes code at load-time.

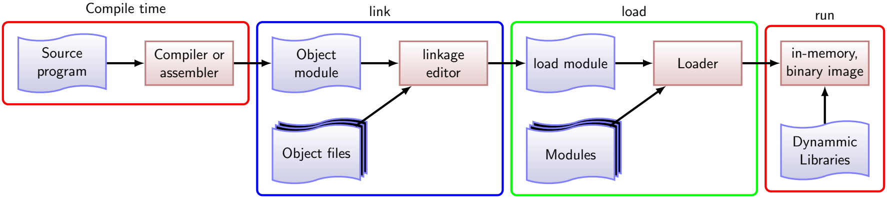
Shared Objects, Dynamic Link Libraries

Virtual Memory

- With relocation a module can exist as one physical copy in memory
- The MMU and OS manage the mapping for each process as it calls functions from the library

Stages in a Process

stages where address are resolved



Paging and swap space

- Allow areas of disk to appear as blocks of memory
- As needed blocks can be moved from physical memory to swap memory
- No different to MMU in managing blocks of code, data and modules
- Allows for apparent increased RAM at a performance cost.

μC tasks

Simple processes

Create Task

```
INT8U OSTaskCreate(void (*task)(void *pd),
                   void *pdata,
                 OS_STK *ptos,
                  INT8U prio);
```

`task` is a pointer to a function (the task)

- the function takes a single pointer `td` as a parameter

`pdata` is a pointer to a data block for passing parameters to the task

`ptos` is a pointer to the area of memory to be used as a stack by the task

`prio` the priority of the task

void * type

- Parameters are sometimes `void *` type.
- This is not a pointer to nothing, this is a pointer to something, anything.
- In order to *read* out the contents it has to be converted (cast) into the right type.
- One interpretation “a pointer to void is a pure address”

void * type

array of integers

```
void sometask(void *data)
{
    int *n = (int *)data;
    int sum = 0
    for(int i=0 ; i<10 ; i++)
        sum += n[i];


    for(;;) ...

}
```

void * type

structure

```
typedef struct modem {
    unsigned int baud;
    enum modes   mode;
    unsigned int port;
} comms;
```

```
void sometask(void *data)
{
    comms *com = data;

    com->baud = 1600;
    switch(com->mode)...

}
```