ECEN 3753: Real-Time Operating Systems

Memory Management

Memory Management Topics

- Types of Memory
- Caching
- Heap vs. Stack
- Static vs. Dynamic Allocation
- Debug Stack Issues
- Address Heap Issues
- Memory Protection Unit
- Memory Management Unit
- Virtual Memory

Types of Memory

- Volatile: lose info on power off
  - Random Access Memory (RAM), Electrical read/write data
    - DRAM (Dynamic RAM)
    - SRAM (Static RAM)
- Non-Volatile: retain info on power off
  - Info not erasable
    - Mask Programmable Read-only Memory (ROM)
  - Info written electrically but not electrically erasable
    - Erasable Programmable ROM (EPROM)
  - Info written and erased electrically
    - Electrically Erasable Programmable ROM (EEPROM)
    - Flash (NAND, NOR)

SRAM vs DRAM

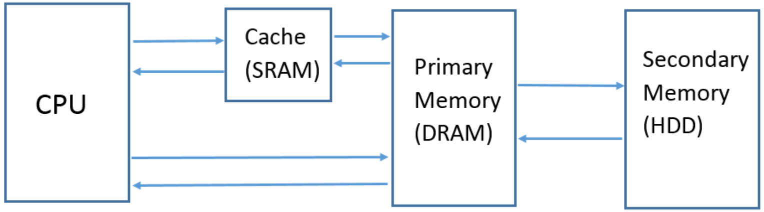
	SRAM	DRAM
Memory Density	Low	High
Cost per Bit	High	Low
Speed	Fast	Slow
Power Consumption	Low	High
Reliability	Good	Bad

<modified slightly from Cooling, Table 6.1>

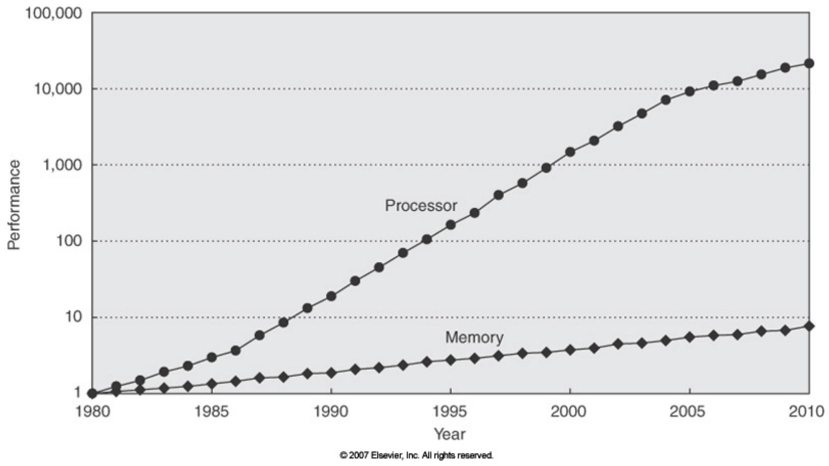
Data Reliability (one view of it. . .)

- SRAM data won't decay, so no data protection is needed
- DRAM data may decay, so data quality protection is needed

Basic Memory Structure: example



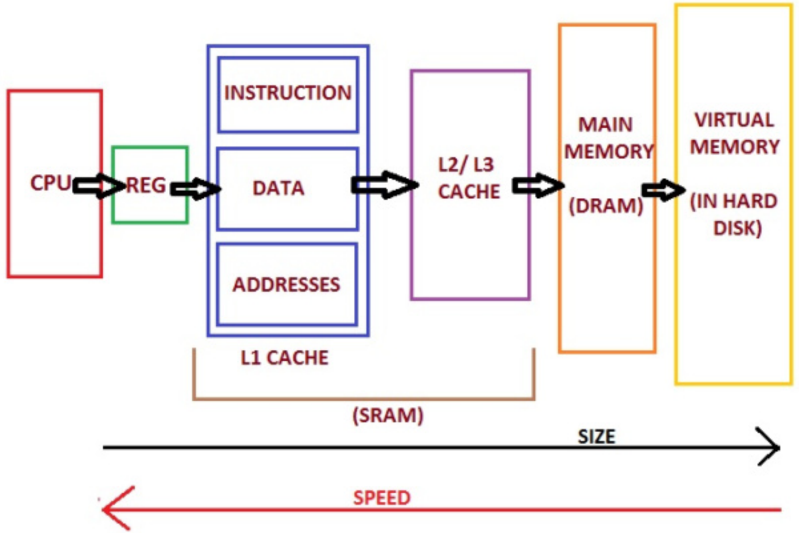
CPU time vs Memory access time



<source: https://www.extremetech.com/computing/261792-what-is-speculative-execution>

- How can we fill the gap? Cache, more levels of cache

Memory Structure w/ Caching



Memory Access time

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 Cache access	0.9 ns	2 s
Level 2 Cache access	2.8 ns	7 s
Level 3 Cache access	28 ns	1 min
Main memory access (DDR DIMM)	~100 ns	4 min
NVMe SSD I/O	~25 us	17 hrs
SSD I/O	50-150 us	1.5-4 days
Rotating HDD	1-10ms	1-9 months
Internet call: SFO to HKG	141 ms	11 years

<distilled from: https://www.prowesscorp.com/computer-latency-at-a-human-scale>

Caching Basics

- Caching works by two principles:
  - Temporal locality:
    - if a program accesses one memory address, there is a good chance that it will access the same address again.
  - Spatial locality:
    - if a program accesses one memory address, there is a good chance that it will access other nearby addresses.

Where are examples, below?

```
sum = 0;
for (i = 0; i < Size; i++) {
    sum = sum + a[i];
}
```

Caching basics

- Cache Hit
  - the cache contains the data/code that the program is looking for
  - good: data/code is accessed faster in cache than main memory
- Cache Miss
  - the cache does not contain the data/code that the program is looking for
  - bad: CPU has to wait for the data/code from slower main memory
- Goal for HW and SW engineer: Improve Cache Hit Rate
  - Speculative execution pipeline
  - Branch taken/not-taken variants
  - Coding choices (loops, data structures)

How can a RealTime system POSSIBLY tolerate caching?

After all, doesn't this add variability that is not predictable?

How can a RealTime system POSSIBLY tolerate caching?

After all, doesn't this add variability that is not predictable?

- Cache pages can be locked in memory for Hard/Firm RT tasks
- Write back/through options to decache writes can be handled via another bus
- Gross speed gains from caching can buy considerable margin, so that MOST of the time, Firm/Soft RT demands are much more easily fulfilled
- Hard/Firm demands can be further supported by dedicated hardware (we'll get to that in DMA and Multi-CPU lectures)

But yes, it needs to be **carefully** analyzed.

Write Cache Types

Write Back:

- Postpones the write to main memory until need or good opportunity

Write Through:

- Simpler logic (no “Dirty” flag), but ties up back-end of write pipeline to main memory immediately.
- Doesn’t inherently protect multi-processor system.

Stack

- LIFO (last in, first out)
- Limit on size (specified when task is created)
- Managed efficiently by CPU
- Store local varialables
- Stack usage grows and shrinks (as functions push and pop variables)

Stack Overflow

- How it happens:
  - Long function call path
  - Recursive function
  - Large local variables (e.g. big array)
  - Overly stingy SoC planners
- How to debug (more to come)

Heap

- Variable can be accessed globally
- No limit on size
- Managed by programmer (allocating and freeing variable)
- (Relatively) Slower access
- Memory leak (allocated but not released by programmer)
- Fragmented (more to come)

Memory Allocation Type Comparison

	Static Allocation	Dynamic Allocation
Time	Performed at static or compile time	Performed at dynamic or runtime
Memory	Assigned to stack	Assigned to heap
Size	Size must be known at compile time	Size may be unknown at compile time
Order	First In, Last Out	No particular order of assignment

Code example on variable in Stack

```
// All the variables in below program
// are statically allocated.
void fun()
{
   int a;
}

int main()
{
   int b;
   int c[10];
   fun();
}
```

Code example on variables in Heap

```
int main()
{
   // Below variables are allocated memory dynamically.
   int *ptr1 = new int;
   int *ptr2 = new int[10];

   // Dynamically allocated memory is deallocated
   delete ptr1;
   delete [] ptr2;
}
```

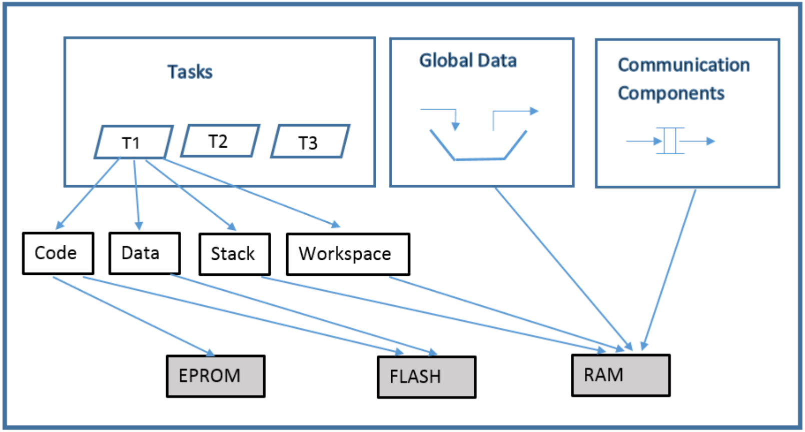
Stack vs. Heap Comparison

	Stack	Heap
Memory allocation	Contiguous block	Arbitrary order
Alloc/Dealloc Invoked	Auto by compiler	Manual by programmer
Alloc/Dealloc Time	Short	Long (and variable)
Access Time	Faster	Slower (indirect ref)
Locality of Reference	Excellent	Adequate
Flexibility	Fixed size	Data can be resized
Main Issue	Stack Overflow	Leakage and Fragmentation

The principles of locality mean that a processor tends to access the same set of memory locations repetitively over a relatively short period of time. Odds are somewhat better with Stack.

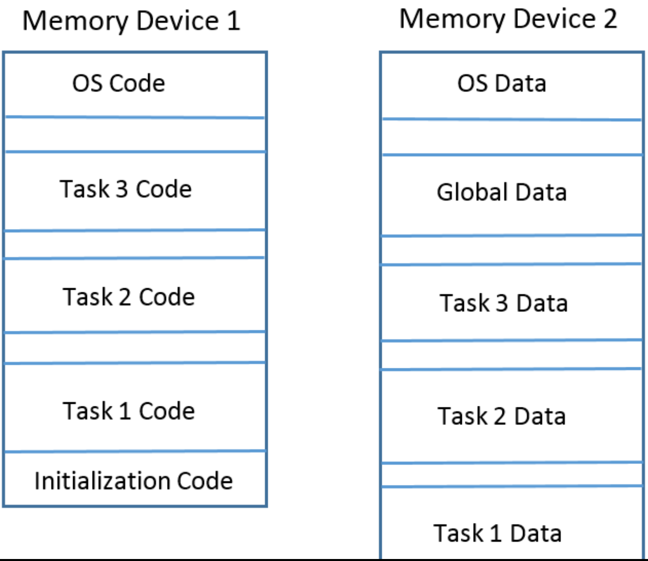
Heap may be used by multiple tasks–thereby requiring coordination (later in Shared Resources)

Task and Memory



This picture shows the tie-together of Tasks, Memory, and multi-task access of global variables and needed inter-task coordinating methods.

Task and Memory, physical view



Those pesky problems with Stack and Heap

When comparing Stack and Heap, it was noted that:

	Stack	Heap
Main Issue	Stack Overflow	Leakage and Fragmentation

Stack issue: Stack Overflow

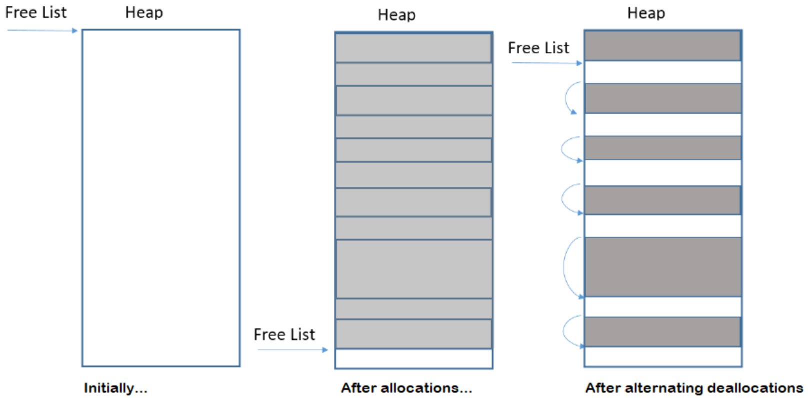
Recall in the "Task & Memory, physical view" picture, a task's data was allocated adjacent to another task's

- When a stack overflows, one task's data overwrites another task's (or OS's) data, which is **VERY** hard to debug
- Weak detection:
  - If there is gap between stacks
    - Fill gap w/ pattern, periodically check whether pattern changed
  - If there is not gap between stack, we can try to get lucky
    - Can run periodic checks (Stack data changing w/o code changing it)
    - Detect Invalid data value (e.g. invalid enum value)
- Robust detection:
  - Detect stack overflow in HW (Memory Protection Unit (MPU))

As long as stack overflow is prevented, the Stack solution looks pretty awesome compared to Heap!

Stack underflow is a bigger problem when people micro-manage the stack. Compilers are better at this than we are.

Heap issue: How Fragmented Memory Happens

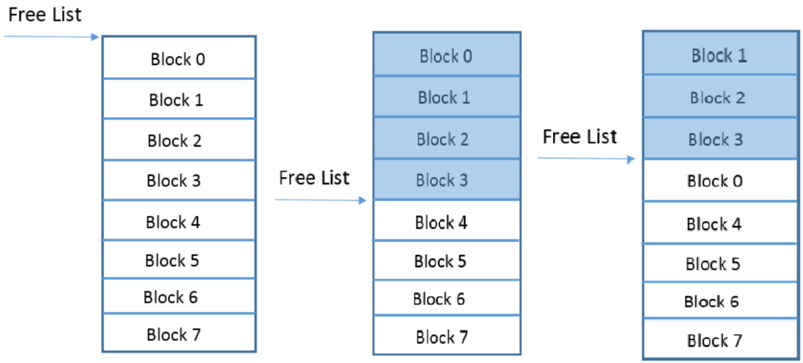


(A very simplified example.)

Fixing Fragmented Memory

- Defragmentation
  - feasible in theory, but generally not realistic in embedded systems (HIGHLY variable free/defrag times)
- Secure Memory Allocation
  - Heap memory is split into partitions (or sections)
  - Memory is allocated from selected partition
  - Only one block is allocated for each request
  - Deallocated memory is always returned to the partition it came from
  - Some RTOS implementations will put a “canary page” between blocks (which can be programmed into an MPU)

Secure Memory Allocation

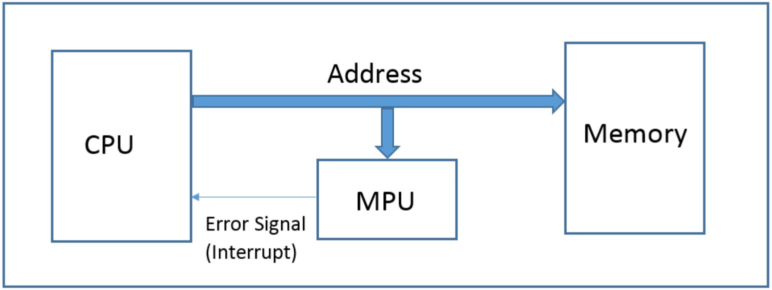


<from Cooling, section 6.4.3>

What are some advantages/disadvantages to returning to the start/end of free list?

Memory Protection Unit (MPU)

- Monitor the address info flowing between CPU and Memory
- Signal error (exception) for any violation

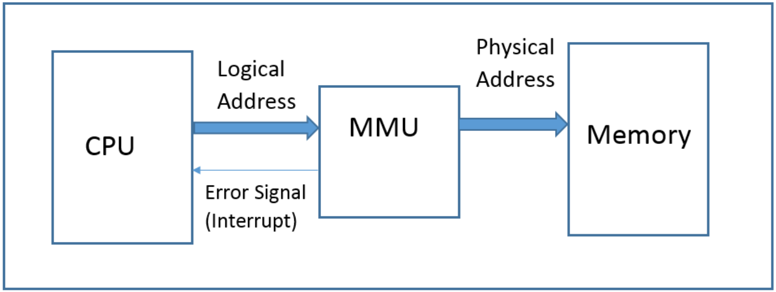


This is what the M3 has.

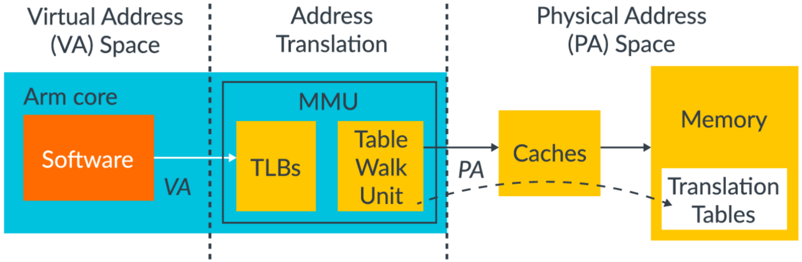
Memory Management Unit (MMU)

- Needed for most OSes that load programs into memory at runtime
  - “Physical address” is not known at compile time
  - Program runs at “logical address”
- Translate logical address to physical address
- Memory protection (same functionality as MPU)
- May handle virtual memory request

Memory Management Unit



Memory Management Unit (Hardware)



From: https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-management-unit-mmu

Memory Managemnet Unit (SW response to HW)

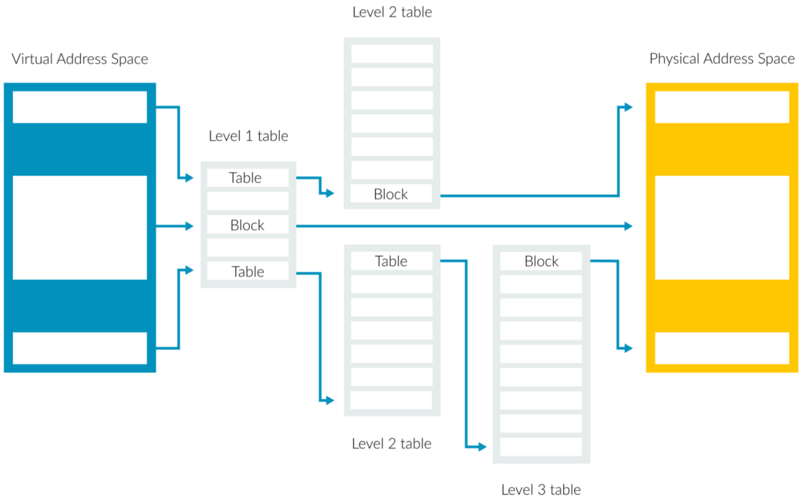
When the TLB indicates a miss, why would a RT engineer want to signal the CPU?

Memory Managemnet Unit (SW response to HW)

When the TLB indicates a miss, why would a RT engineer want to signal the CPU?

- 10 to 1e6 cycles may be needed for fetch
  - 100 for DDR memory (Got any other work to do?)
  - 1e6 (and other code execution to do the fetch) for HDD (Definitely have other work to do, too!)

Memory Management Unit (4-level ARM)

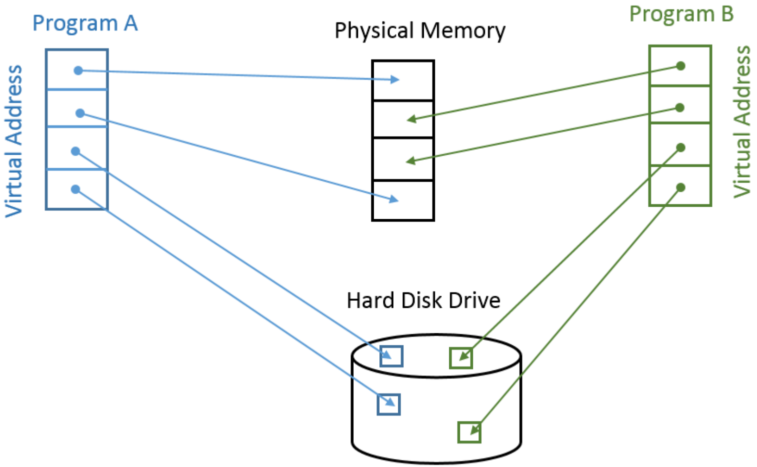


From: https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-

Virtual Memory

- Virtual Memory is handled by MMU while in memories
- Virtual Memory is transparent to user program
- Virtual Memory needs HW (MMU) and SW (OS to set it up) support
- Virtual address be can mapped to physical memory, or to secondary storage (e.g. HDD) by having the OS support the fetch resulting from the “miss”
  - When a “miss”" happens, the data needs to be fetched, then the code that faulted is re-started on the memory access instruction
- Same virtual address for different tasks is mapped to different physical addresses

Virtual Memory


Program A
Virtual Address
Physical Memory
Program B
Virtual Address
Hard Disk Drive