Memory Configurations

Flat, Embedded Build

The normal build of Nuttx for the typical embedded environment uses a single blob of code in a flat address space. For most lower end CPUs (such as the ARM Cortex-M family), this means executing directly out of the physical address space. Even if the CPU has an MMU (such as with the ARM Cortex-A family), the typical Nuttx build still uses a flat address space with the MMU providing only an identity mapping. In this case, there is still benefit from using the MMU because the MMU provides fine control over caching and memory behavior over the address space.

On-Demand Paging

On-Demand Paging. Nuttx also supports on-demand paging via CONFIG_PAGING. On-demand paging is a method of virtual memory management and requires the the CPU architecture support a MMU. In a system that uses on-demand paging, the OS responds to a page fault by copying data from some storage media into physical memory and setting up the MMU to provide the necessary virtual address mapping. The CPU can then continue from the page fault with the necessary memory in place for the virtual address.

Execution Image. The execution image is still built as one blob and appears as one blob on the storage media. But the execution image is paged into arbitrary physical addresses with non-contiguous virtual addresses. The physical and virtual address spaces are then “checker boards” of memory in use.

Advantages. The main advantage of on-demand paging is that you can execute a single program that is much larger than the physical address space or a collection of programs that together are much larger than the physical address space.

Current Implementation. On-demand paging is currently implemented only for the NXP LPC31xx family. The LPC31xx has a 192KIB internal SRAM and with on-demand paging the LPC31xx can execute a huge program residing in SerialFLASH by bringing in new pages as needed from the SerialFLASH when page faults occur.

Protected Build

Protected Build Mode. Nuttx also supports a protected build mode for certain CPU architectures if CONFIG_BUILD_PROTECTED is selected. In this mode, Nuttx is built as two blobs, one privileged and one unprivileged. The one, privileged blob contains the RTOS, and the other, unprivileged blob holds all of the applications. The build supports system calls via a call gate so that the unprivileged, application code can access the privileged RTOS services.

Memory Protection. Within each blob, the address space is flat. No MMU is required to support the protected build since no address mapping is performed. In fact, this feature is currently available only for the ARM Cortex-M family. In this case the Cortex-M’s MPU provides the security in the address spaces of the two blobs. This feature could also be implemented with a CPU that supports an MMU, but there has thus far been no reason to implement such a configuration.

Dynamic Memory Allocation. The purpose of protected build then is focused primarily on securing the OS and CPU resources from potential rogue applications. The MPU simply protects the hardware, code regions, and data regions of the RTOS. But dynamic memory allocations become more complex: Protection is also required for (certain) memory allocations made by the RTOS. The RTOS must also be capable of allocating memory that is accessible by user applications (such as the user thread stacks).

Dual Heaps. In systems with MMUs, the privilege of each page of memory can be controlled and there are established architectures for memory management of processes (see below). However, with only an MPU with a limited number of pages (the Cortex-M has 8 pages only!) we are forced to resolve this problem by dividing available heap memory into two heaps: A privileged heap and an unprivileged heap, using different allocations mechanisms for each (kmalloc and malloc, respectively).

Address Environments

Address Environments. If the option CONFIG_ARCH_ADDRENV is selected, then NuttX will support address environments in the following way: The base code is still one blob and identical in every way to the "Flat Embedded Build" discussed above. But all applications are loaded into RAM from executable files, separately compiled and separately linked programs, that reside in a file system. Instead of starting the user application at a fixed, in-memory address (such as nsh_main()), the system will start the program contained in an executable file, given its path. That initial user program can then start additional applications from executable files in a file system.

Per Program. As each program is started, a new address environment is created for the new task. This address environment is then unique for each task. A task does not have the capability to access just anything in the address environment. A task may only access addresses within its own address space and within the address space of the base code.

MMU Required. The CPU must support an MMU in order to provide address environments. This feature was originally implemented to support the ZiLOG Z180 which is an 8-bit CPU (basically a Z80) that also supports a simple MMU. Specifically for the P112 platform. Unfortunately, due to complex tool issues and fading interest, that port was fully implemented but never tested. As of this writing, the implementation of address environment support for the Cortex-A family is complete and verified. An example configuration is available at nuttx/boards/arm/sama5/sama5d4-ek/configs/elf.

Kernel Build

The Kernel Build. And finally, there is the kernel build that is enabled with CONFIG_BUILD_KERNEL=y. The NuttX kernel build mode is similar to building with address environments:

- Each application process executes from its own, private address environment.

But, in addition, there are some features similar to the protected build mode:

- NuttX is built as a monolithic kernel, similar to the way that NuttX is built in the Protected Build Mode.
- All of the code that executes within the kernel executes in privileged, kernel mode. Again, this is analogous to the Protected Build Mode.
- All user applications are executed with their own private address environments in unprivileged, user-mode.

MMU Required. In order to support this kernel build mode, the processor must provide a Memory Management Unit (MMU). The MMU is used to provide both the address environment for the user application as well as to enforce the user-/kernel-mode privileges. This kernel build feature has been fully implemented and verified on the Cortex-A family of processes. A functioning example can be found at nuttx/boards/arm/sama5/sama5d4-ek/configs/knsh.

Processes. Such user applications that execute their own private, unprivileged address environments are usually referred to as processes. The CONFIG_BUILD_KERNEL=y build is the first step toward support for processes in NuttX. The full roadmap for support for processes under NuttX is addressed in the next section of this Wiki page.

The Roadmap Toward Processes

Processes. Let's call these programs with their own address spaces processes. The term process may bring along some additional baggage and imply more than is intended, at least in the short term. Conceptually, processes are a natural extension of what is referred as a task group in NuttX parlance.

Binding to the Base Code

Binding to the Base Code. So how does the process communicate with the base code? There are two ways now. Both are fully implemented, either would work:
Symbol Tables: In this case, the object file is only partially linked. It contains references to undefined external symbols. In this case, the base code must provide a symbol table, that is, a table that maps a symbol to its address in the base code. The NuttX dynamic loader (binfmt/) will automatically link the the program to base code using this symbol table when the program is loaded from the file system into RAM.

Symbol Table Helpers. NuttX provides several helpers to make dealing with symbol tables exported from the base code less painful. All base code symbols are maintained in comma-separated-value (CSV) files. There are three: nuttx/libc/libc.csv, nuttx/libc/math.csv, and nuttx/syscall/syscall.csv. These CSV files contain descriptions of all symbols that could be exported by the base code. Then there is the program at nuttx/tools/mksymtab.c that can be used to generate symbol tables from these CSV files.

Call Gates: A second way for the program loaded into memory to communicate with the base code is via a call gate, i.e., via system calls. A call gate is normally used to change the privilege level of the thread when calling into system code. However, the same system call mechanism can be used to simply call into the OS without having any a priori knowledge of the address of the RTOS service in the base code. Thus, the symbol tables could be eliminated completely at least for the case of OS system calls (the C library is another story).

Status. Both the symbol table logic and the system call logic are already fully implemented and verified and fully integrated with address environments up to this point. A sample configuration that does all of these things is here: nuttx/boards/arm/sama5/sama5d4-ek/configs/elf and additional information is available in nuttx/boards/arm/sama5/sama5d4-ek/README.txt.

Protection and Privileges

Protection. One of the greatest benefits of processes, however, is the security that they can provide. As described above, on program cannot access memory resources of any other program because those resources lie in a different address environment. But none of the address space outside of the program private address environment is protected: The base code and its private memory are not protected; the hardware is not protected; and anything allocated from the heap is not protected. So a misbehaving, rogue program can still crash the system or corrupt the stack or memory allocation made by other programs.

MMU Protection. Of course, the MMU can also be used to protect the resources outside of programs address environment. It would be a simple matter to protect the hardware and the base code memory resources from programs. The base code would run in privileged mode with full access; the user applications would run in unprivileged mode and only have the ability to access memory resources within their own address environment. This use of the MMU, however, raises some additional issues:

No Symbol Tables. Symbol Tables could not be used in such a protected environment to call into the base code. Only system calls could be supported. These call gates could switch to privileged mode temporarily in order execute the RTOS service, then return to unprivileged mode when returning to the program.

The Line

The Line. Let's draw a line right here. Everything above this point is history and just a summary of the way things are. Every thing below this point is a roadmap that I will be following in the further development of these features. Some bits and pieces and been implemented and indicated.

So continuing from the other side of this line...

Libraries

Libraries. But not all of the symbols that might be exported from the base code map to system calls. Many of the NuttX facilities operate simply in user mode: Think of strlen(), printf(), rand(), etc. With this
strict enforcement of address spaces, the only way that these addition functions can be called is if they are brought into the same address space as the program.

**NuttX Libraries.** User programs are separately compiled and separately linked. NuttX currently builds all of the user callable functions into static libraries. So, as part of their build process, user programs can simply link with these static libraries to include to bring the callable code into the *address space* of each program. Libraries created by NuttX include: **libsyscall.a** that holds the system call “proxies”, **libc.a** that holds the NuttX C library, and **libnx.a** that hold the graphics interface library. (*FULLY implemented*).

**Shared Libraries.** The downside to using static libraries like this is that a function in the static libraries would be duplicated in the *address environment* of *every* process that uses that function. Some very commonly used functions, such as `printf()`, can be quite large and the penalty for duplicating `printf()` in every process *address environment* could be a significant problem. The solution is to use *shared libraries*, that is, libraries of functions that (1) have only one .text space in RAM, but (2) separate .data and .bss space, and are (3) separately linked into with the program in each *address environment* (*not implemented*).

**Dynamic Loader.** Such shared library support would be a significant yet natural extensions to the existing NuttX dynamic loader. (*not implemented*).

**Partially Linked vs Fully Linked Objects**

**Partially Linked Objects.** The NuttX ELF loader currently accepts partially linked objects, that is, programs that do not have the final addresses of functions and data resolved. Instead, these ELF program files include relocation and symbol information inside them. The NuttX ELF loader then accesses this relocation information and resolves all of the function and data addresses when the program is loaded into memory. (*FULLY implemented*).

**The Downside.** Two things to note about this approach: First, note that retaining the relocation information in the ELF program files makes the partially linked object files much bigger than necessary because they have to carry all of this relocation and symbol information along with all of the code and data. A second thing to note is that every in-memory representation is unique; each is a one-off, resolved version of the partially linked object file. Each might have been relocated differently. A consequence of this is that if there are multiple copies of the same program running, all of the code must be duplicated in memory. It is not possible to share anything even though the programs may be identical.

**Fully Linked Objects.** These partially linked objects are required in systems that have no MMU. In that case, each ELF executable will be loaded into a unique physical memory location and, hence, truly will be unique and truly not shareable. But there is a difference if the ELF programs are loaded in the virtualized, kernel build; in that case, all ELF executables are loaded into the *same* virtual memory space! The executables can be fully linked at build time because the final addresses are known and program files can be stripped of all unnecessary relocation and symbolic information. In the fully linked build, all function and data addresses are fully resolved to their final virtual addresses when the ELF executable is built. (*Possibly functional, but not tested*).

**Shared .text Regions.** So in the case of fully linked objects, there is no obstacle to sharing the code. All that is required is a minor modification to the way that the ELF .text region is allocated. If the ELF .text region is mapped into memory from the ELF file using the `mmap()` interface instead of being allocated from the page memory pool, then the .text region is naturally share-able. NuttX does include a partial implementation of the `mmap()` file mapping interface but that implementation is tailored for use only in the flat, embedded build and cannot be used with the kernel build, at least in its current form. That full implementation of `mmap()` plus the minor changes to the NuttX ELF loader are all that are required to support fully share-able .text sections – as well as the memory savings from not carrying around the relocation and symbol information (*Not implemented*).

Memory Management
Memory Management. When memory is allocated by the privileged base code or by an unprivileged application. The resulting memory allocation must be accessible in and only in the address space where the memory was allocated. The strategy of using two heaps as was described above for the simple “Protected Build” cannot work in this case. Rather, each address space must have its own heap! The function `malloc()` must exist in each address environment and memory allocated via `malloc()` must be available only in that address space. (FULLY implemented, not tested).

Page Allocator. In such an environment, memory is used controlled by a simple “page allocator”. A page allocator is really a very simple memory allocator that allocates physical memory in pages that can then be mapped into the appropriate address environment using the MMU. The interface between each instance of `malloc()` and the base page allocator are via the `brk()` and `sbrk()` system calls. For historic reasons, these system calls deal with something called the break value, hence their names, but let's just think of this as how much memory is available in the process local heap. (FULLY implemented).

```c
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t incr);
```

- The `brk()` function sets the break value to `addr` and changes the allocated space accordingly (not implemented).
- The `sbrk()` function adds `incr` bytes to the break value and changes the allocated space accordingly. If `incr` is negative, the amount of allocated space is decreased by `incr` bytes. The current value of the program break is returned by `sbrk(0)`. (FULLY implemented).
- See OpenGroup.org for further information about these system calls.

Shared Memory. Once you have all of the user application logic encapsulated as processes within their own private address environment, then you following strict rules about how these different user processes communicate with each other. Of course, all of the standard Inter-Process Communication (IPC) methods work fine: Semaphores, signals, message queues, etc. But what about data? How do processes share large, in-memory data sets?

From the title of this section you can see that the answer is via Shared Memory, that is via chunks of memory which are mapped into each process’ virtual address space. Here are the set of interfaces implemented for this purpose in NuttX:

- `shmget()`. Get the shared memory identifier, the `shmid`, of a shared memory region. The `shmid` is like the file descriptor that you get when you open a file and `shmget()` is much like `open()`. And like opening a file, there are flags that can control where you want to open the shared memory object for read/write or read-only purpose or if you want to create the shared memory region if it does not exist. (FULLY implemented but without protection modes and also untested).
- `shmctl()`. Once you have the `shmid`, you can use that value with other interfaces to manage the shared memory interface. `shmctl()` will, for example, let you get and modify the characteristics of the shared memory region. `shmctl()` will also let you remove a shared memory region when it is no longer needed. (FULLY implemented but without privilege modes and also untested).
- `shmat()`. Given the `shmid`, `shmat()` can be used to attach the shared memory region, i.e., to map it into the user process address space. (FULLY implemented but untested)
- `shmdt()`. Complementing `shmat()`, `shmdt()` can be used to detach a shared memory region from the user process address space. (FULLY implemented but untested)

Closely related to to these interfaces are the `mmap()` and `munmap()` interfaces. While these interfaces are implemented in NuttX and available in the `flat` build, they have not yet been extended to provide full shared memory support as described above for the `shm` interfaces. (not implemented).

Dual Stacks

**Dual Stacks.** Having a program loading from a file system is only interesting if that program can also run other programs that reside in a file system. But that raises another level of complexity: We cannot instantiate the new program environment without also destroying the current program environment. So all data must be preserved by copying the caller’s data into the common, neutral kernel address environment before the switch.

But what about the callers stack? That stack lies in the calling process’ address environment. If in the system call, the kernel logic runs on the caller’s user stack, then there will almost surely be some disaster down the road when switching process contexts. How do you avoid losing the caller’s stack contents that the C logic needs to run while also instantiating the new program's address environment?

The usual solution is to have two stacks per thread: The large, possibly dynamically sized, user stack and a much smaller kernel stack. When the caller issues the system call, the system call logic switches stacks: It replaces the stack pointer with the reference to the user stack with a new reference to the thread’s kernel stack. Then the system all executes with a process neutral kernel stack avoiding the stack transition problems. The system call logic then restores the program's stack pointer before returning from the system call (this is, of course, complicate by the possibility of nested system calls and system calls that can generate context switches or can dispatch signals).

Doesn’t it cost a lot of memory to have two stacks? Yes and no, depending on what kind of memory usage you are used to. Remember that with the MMU, the user memory space is quantized into units of pages which are typically 4KiB and can grow upward from there in multiples of 4KiB as needed. The kernel stack has limited depth. It does not need to be dynamically sized and can probably be very small (perhaps as little as 1KiB). So, yes, the dual stacks do use more memory but the impact is not as significant as might first think.

(Dual stack support is FULLY implemented in NuttX).

Further Down the Road

**Other Topics.** If all of the above were implemented, then NuttX could probably rightfully claim to be a small Unix work-alike. From there, several additional topics could be addressed but this is too far down the roadmap for me the contemplate in any real detail:

- **On-Demand Paging and Swap.** Why not keep programs and data in a file system and swap the state into physical memory as needed?(not implemented).
- **mmap().** True shared memory and true file mapping could be supported. I am repeating myself (not implemented).
- **fork().** The fork() interface could be supported. NuttX currently supports the “crippled” version, vfork() but with these process address environments, the real fork() interface could be supported.(not implemented).
- **Dynamic Stack Allocation.** Completely eliminate the need for constant tuning of static stack sizes.(not implemented).
- **Shared Libraries.** Am I repeating myself again?(not implemented).
- **Program build tools.** Then how do we make building programs for NuttX easy. It should be as easy as arm-nuttx-eabi-gcc myprogram.c.(not implemented).

Some Consequences

**task_create()**

In the traditional, flat NuttX build, the interface task_create() and task_spawn() are the standard interfaces for creating new tasks. Both take the entry point address of the new task as an argument. These task creation interface, however, cannot be used within code executing in an address environment.

Why not? It is kind of a long story.

Remember that when a new task is created, a new task group is also created. That task group provides all of the resources shared between the parent task and all of its child threads. One of these resources that is shared is
the address environment.

The behavior of `task_create()` is to create a new task with no address environment, or more correctly with the common kernel address environment.

When the parent process was created, it was created in its own private address. The address of the new task entry point passed to `task_create()` must lie in this same private address environment.

So you can see that we have a perverse configuration here: The `.text` address of the new task lies in the private address space of the parent task; but the task itself uses the common OS address space. There are two problems here: (1) because of privilege issues, the new task may not have access either to the parent task's address environment or to the common kernel address environment. It may just crash immediately, depending upon how these things are configured.

Or assuming that there are no privilege issues, (2) the new task certainly will depend on the address space of the parent task being in place in order to run. As soon as the parent's address space disappears (due, perhaps, because the parent task's address space was de-activated or perhaps the parent task exits destroying its address environment). The child task would crash immediately thereafter.

So how do you create new tasks/processes in such a context. There is only one way possible; by using an interface that takes a file name as an argument (rather than absolute address). New processes started with `vfork()` and `exec()` or with `posix_spawn()` should not have any of these issues.

ARM Memory Management
ARM Page Table Organization

Cortex-A Page Tables. Let's focus on the ARMv7-A (aka, Cortex-A) MMU for now. That MMU uses a two-level page table:

Level 1 Page Table
- The size of virtually addressable region is 4GiB.
- The size of first level page table is 4096 entries (16KiB).
- Each each entry in the page table provides the mapping for 1MiB of virtual memory. One entry may be either:
  - A section mapping which maps 1MiB of contiguous virtual addresses to 1MiB of physical memory (only occasionally used), or
  - It may refer to a beginning of a second level page table (much more commonly used).

Level 2 Page Table
- Differing pages sizes are possible, but use of 4KiB pages used (this is the smallest page size for the Cortex-A).
- With a 4KiB page table, 256 page table entries (PTEs) are required to span the 1MiB region.

Advantages/Disavantages

Advantages. The most important benefits of the ARMv7-A page table are:
- The TLB reloads are done automatically by the hardware without software intervention. This is a huge performance advantage to other architecture where each mapping must be instantiated via logic in a page fault exception handler.
- The multiple levels and configurable page sizes add flexibility.

Disadvantages. The ARMV7-A page table is, in fact, well-suited for higher end platforms that do not suffer from memory and performance constraints. But for the most constrained platforms, the following is a big issue:
- Each page table required 16KiB of memory PLUS 1KiB of memory for each level 2 page table (assuming 4KiB page size). That would result in a maximum size of over 4MiB!
- The state of the page table is part of the process' context and must be saved and restored on every context switch.

switch!

Let’s look first at how Linux deals with the ARM page tables; then let’s propose a scaled down approach for NuttX:

**ARM/Linux Page Table Notes**

**Linux Summary.** This is a summary of how the Linux kernel uses the page table to control a process' memory mapping with a Cortex-A CPU. This is not based upon my authoritative knowledge, but is rather based on Google for explanations.

**Virtual Address Space Partitioning**
- The 4GiB virtual address is partitioned with 3GiB of user space and 1GiB of kernel space: Virtual address 0x0000:0000-0xbfff:ffff is user space while 0xc000:0000-0xffff:ffff is kernel space.
- Level 1 page table entries 0-3071 map user space virtual addresses and entries 3072-4095 map the kernel space addresses.

**Process Page Tables**
- The ARM co-processor register TTBR0 holds the address for the current page directory (the page table that the MMU is using for translations).
- Each user process has its own page table located in the kernel address space.
- For each process context switch, the kernel changes the TTBR0 to the new user process page table.
- Only TTBR0 is used. TTBR1 only holds the address of the initial swapper page (which contains all the kernel mappings) and isn't really used for virtual address translations.
- For each new user process, the kernel creates a new page table, copies all the kernel mappings from the swapper page(page frames from 3-4GiB) to the new page table and clears the user pages (page frames from 0-3GiB). It then sets TTBR0 to the base address of this page directory and flushes cache to install the new address space. The swapper page is also always kept up to date with changes to the mappings.
- The swapper page is usually located at addresses 0xc0004000-0xc0008000.

**ARM/NuttX Page Table Proposal**

**Size Reduction Tradeoff**

**Size Reduction Tradeoff.** We can reduce the memory usage by page tables in NuttX by:

- Keeping only a single level 1 page table that is shared by all task groups.
- Level 2 page tables are, of course, still need to be duplicated for each process.
- To avoid copying the 3GiB mapping used by Linux, we can simply reduce the size of the virtual address space so that instead of copying 3,072 entries on each process switch, we copy perhaps 4. That would limit the virtual address range for each process from 3GiB to only 4MiB. But that is probably reasonable but would also be configurable.
- Reducing the supported virtual address from 3GiB to, say, 4MiB would also reduce the amount of memory that has to be allocated for each process. Continuing with this 4MiB suggestion, it would following that no more than 4KiB would need to set aside for level 2 page table support for each process.
- Further, let's not make any assumptions about what virtual address range corresponds to user space and which corresponds to kernel space. It is simply an agreement that must be made between the platform implementation and the program's linker script.

**Per-Process/Per-Thread Regions**

**Per-Process Regions.** Three regions must set aside for each process to hold:

- A level 2 mapping for the .text region,
- A level 2 mapping for the static data region (.bss and .data), and
A level 2 mapping for the process' heap.

**Instantiation/Extensibility.** The first two mappings would be created when the program is loaded into memory; the first heap mapping would be created when `sbrk()` is first called. All three could be extended at runtime if shared libraries are supported (augmenting `.text`, `.bss` and `.data`) or when the heap is extended by subsequent `sbrk()` calls.

**Configuration.** The following configuration options are proposed:

- **CONFIG_ARCH_TEXT_VBASE** - The virtual address of the beginning the `.text` region
- **CONFIG_ARCH_DATA_VBASE** - The virtual address of the beginning of the `.bss`/.`data` region.
- **CONFIG_ARCH_HEAP_VBASE** - The virtual address of the beginning of the heap region.
- **CONFIG_ARCH_TEXT_NPAGES** - The maximum number of pages that can be allocated for the `.text` region. This, along with knowledge of the page size, determines the size of the `.text` virtual address space. Default is 1.
- **CONFIG_ARCH_DATA_NPAGES** - The maximum number of pages that can be allocated for the `.bss`/.`data` region. This, along with knowledge of the page size, determines the size of the `.bss`/.`data` virtual address space. Default is 1.
- **CONFIG_ARCH_HEAP_NPAGES** - The maximum number of pages that can be allocated for the heap region. This, along with knowledge of the page size, determines the size of the heap virtual address space. Default is 1.

**Implementation.** The task group resources are retained in a single structure, `task_group_s` that is defined in the header file `nuttx/include/nuttx_sched.h`. The type `group_addrenv_t` must be defined by platform specific logic in `nuttx/arch/*/include/arch.h`. This is a first cut proposal at that type might be:

```c
struct group_addrenv_s
{
    FAR uintptr_t *text[ARCH_TEXT_NSECTS];
    FAR uintptr_t *data[ARCH_DATA_NSECTS];
    FAR uintptr_t *heap[ARCH_HEAP_NSECTS];
};
typedef struct group_addrenv_s group_addrenv_t
```

Where each 1MiB section refers to a level 2 page table that maps 256 4KiB pages:

```c
#define __PG2SECT_SHIFT   (20 - MM_PGSHIFT)
#define __PG2SECT_MASK    ((1 << __PG2SECT_SHIFT) - 1)
#define ARCH_PG2SECT(p)   (((p) + __PG2SECT_MASK) >> __PG2SECT_SHIFT)
#define ARCH_SECT2PG(s)   ((s) << __PG2SECT_SHIFT)
#define ARCH_TEXT_NSECTS  ARCH_PG2SECT(CONFIG_ARCH_TEXT_NPAGES)
#define ARCH_DATA_NSECTS  ARCH_PG2SECT(CONFIG_ARCH_DATA_NPAGES)
#define ARCH_HEAP_NSECTS  ARCH_PG2SECT(CONFIG_ARCH_HEAP_NPAGES)
```

These tables would hold the physical address of the level 2 page tables. All would be initially `NULL` and would not be backed up with physical memory until mappings in the level 2 page table are required.

**Per-Thread Regions.** One region must set aside for each thread to hold:

- The thread's stack

This stack would be initially of size zero and would be backed-up with physical pages during page fault exception handling to support dynamically sized stacks for each thread.

**Configuration.** The following configuration options are proposed:
- **CONFIG_ARCH_STACK_VBASE** - The virtual address of the beginning the stack region
- **CONFIG_ARCH_STACK_NPAGES** - The maximum number of pages that can be allocated for the stack region.

  This, along with knowledge of the page size, determines the size of the stack virtual address space. Default is 1.

**Implementation.** The thread resources are retained in a single structure, tcb which is defined in the header file nuttx/include/nuttx/sched.h. The type xcptcontext must be defined by platform specific logic in nuttx/arch/*/include/irq.h. This structure might be extended to include:

```c
FAR uintptr_t *stack[ARCH_STACK_NSECTS];
```

Where again:

```c
#define ARCH_STACK_NSECTS ARCH_PG2SECT(CONFIG_ARCH_STACK_NPAGES)
```

Then what happens on each context switch?

- Since there is only a single page table, TTBR0 never changes.
- Instead, the particular Level 1 page entries are replace based upon the physical page allocations in group_addrenv_t and xcptcontext.
- Assuming again a 4MiB per-process virtual address space, at most only four elements of the level 1 page table would have to change: .text, .bss/.data, heap, and stack.
- For context switches within the same task group, only the stack level 1 table entry would need to change.
- The MMU TLBs and processor caches would still have to be flushed and invalidated for the (smaller) user virtual address range.

**Terminology**

- **Address Environment**: This is really a generic phrase to refers to the memory addressable by the software. However, in the context of this Wiki page, we will be referring to something more specific. We will be referring a memory architecture that supports multiple, per-task address environments: Each task can execute within its own address environment. Tasks with private address environments may sometimes be called *processes*.
- **Blob**: A block of code and/or data within a restricted range of contiguous addresses.
- **Call Gate**: A call gate is a mechanism for calling into privileged code from unprivileged code, raising the privilege level of the thread temporarily for the call. A typical method of implementing a call gate is through software trap or software interrupt instructions: The interrupt will place the thread into a privileged mode of operation where it can then execute the call. These call gates are used to implement system calls (SYSCALLS), i.e., calls from user application code into OS system services.
- **Flat Address Space**: An address space is flat if either (1) there is no mapping of physical addresses to virtual addresses, or (2) there is a 1-to-1 mapping of the physical address space to a virtual address space.
- **Identity Mapping**: When a CPU has a MMU that is used to map the physical address space to a virtual address space and the virtual addresses map to the same virtual address. In this case, the MMU is is not being used for memory mapping, but rather only for its ability to color memory regions.
- **MMU**: Memory Management Unit. Can be configured to map physical addresses in one address region to virtual addresses in a different address region. Can also color an address region by controlling the privileges required to access the memory, the cache properties of the memory, and other memory attributes.
- **MPU**: Memory Protection Unit. Can be be configured to protect memory.
- **Page**: The size of a block of memory that can be mapped using an MMU. The MMU can handle pages of different sizes. Other terminology may be used for very large pages; ARM calls these *sections*. And *sections* may be divided down to to smaller pages of differing sizes.
- **Page Fault**: When the MMU is unable to map a virtual address to a physical address, then a page fault occurs.

http://www.nuttx.org/Printed on 2019/08/10 21:55
The page fault means that there is no TLB in the MMU that can provided the necessary mapping. In response to a page fault, the MMU may consult a page table in an attempt to resolve the page fault or it may generate a page fault interrupt so that software can resolve the page fault. An unresolvable page fault is fatal and usually results in a crash.

- **Page Table**: A data structure in memory that is accessed by the MMU to reload TLBs. The able has a precise format determined by the MMU hardware: It is configured by software to support the desired mapping and accessed via hardware DMA when a page fault occurs.
- **Physical Address**: The actually address that appears on the CPU bus and provided to the memory parts when the memory is accessed.
- **PTE**: Page Table Entry
- **TLB**: Translation Look-Aside Buffer. An element of the MMU that maps one page of memory. This may be loaded by explicit logic as part of page fault handling (as is typically done with MIPS) or may be loaded automatically by the MMU from a page table (as with the ARM).
- **Virtual Address**: The memory addresses used by the software running in the CPU. This might be mapped to a different virtual address by an MMU.

From: http://www.nuttx.org/ - NuttX Real-Time Operating System


Last update: 2019/08/09 16:48