# On-Demand Paging

# *On-Demand Paging*

Last Updated: February 4, 2019

## Table of Contents



# Introduction

## Overview

This document summarizes the design of NuttX on-demand paging. This feature permits embedded MCUs with some limited RAM space to execute large programs from some non-random access media.

What kind of platforms can support NuttX on-demang paging?

1. The MCU should have some large, probably low-cost non-volatile storage such as serial FLASH or an SD card. This storage probably does not support non-random access (otherwise, why not just execute the program directly on the storage media). SD and serial FLASH are inexpensive and do not require very many pins and SPI support is prevalent in just about all MCUs. This large serial FLASH would contain a big program. Perhaps a program of several megabytes in size.
2. The MCU must have a (relatively) small block of fast SRAM from which it can execute code. A size of, say 256K (or 192K as in the NXP LPC3131) would be sufficient for many applications.
3. The MCU has an MMU (again like the NXP LPC3131).

If the platform meets these requirement, then NuttX can provide on-demand paging: It can copy .text from the large program in non-volatile media into RAM as needed to execute a huge program from the small RAM.

## Terminology

`g_waitingforfill`
: An OS list that is used to hold the TCBs of tasks that are waiting for a page fill.

`g_pftcb`
: A variable that holds a reference to the TCB of the thread that is currently be re-filled.

`g_pgworker`
: The *process* ID of the thread that will perform the page fills.

`pg_callback()`
: The callback function that is invoked from a driver when the fill is complete.

`pg_miss()`
: The function that is called from architecture-specific code to handle a page fault.

`TCB`
: Task Control Block

# NuttX Common Logic Design Description

## Initialization

The following declarations will be added.

- **`g_waitingforfill`**. A doubly linked list that will be used to implement a prioritized list of the TCBs of tasks that are waiting for a page fill.
- **`g_pgworker`**. The *process* ID of the thread that will perform the page fills

During OS initialization in `sched/init/nx_start.c`, the following steps will be performed:

- The `g_waitingforfill` queue will be initialized.
- The special, page fill worker thread, will be started. The `pid` of the page will worker thread will be saved in `g_pgworker`. Note that we need a special worker thread to perform fills; we cannot use the "generic" worker thread facility because we cannot be assured that all actions called by that worker thread will always be resident in memory.

Declarations for `g_waitingforfill`, `g_pgworker`, and other internal, private definitions will be provided in `sched/paging/paging.h`. All public definitions that should be used by the architecture-specific code will be available in `include/nuttx/page.h`. Most architecture-specific functions are declared in `include/nuttx/arch.h`, but for the case of this paging logic, those architecture specific functions are instead declared in `include/nuttx/page.h`.

## Page Faults

**Page fault exception handling**. Page fault handling is performed by the function `pg_miss()`. This function is called from architecture-specific memory segmentation fault handling logic. This function will perform the following operations:

1. **Sanity checking**. This function will ASSERT if the currently executing task is the page fill worker thread. The page fill worker thread is how the page fault is resolved and all logic associated with the page fill worker must be "locked" and always present in memory.
2. **Block the currently executing task**. This function will call `up_block_task()` to block the task at the head of the ready-to-run list. This should cause an interrupt level context switch to the next highest priority task. The blocked task will be marked with state `TSTATE_WAIT_PAGEFILL` and will be retained in the `g_waitingforfill` prioritized task list.
3. **Boost the page fill worker thread priority**. Check the priority of the task at the head of the `g_waitingforfill` list. If the priority of that task is higher than the current priority of the page fill worker thread, then boost the priority of the page fill worker thread to that priority. Thus, the page fill worker thread will always run at the priority of the highest priority task that is waiting for a fill.
4. **Signal the page fill worker thread**. Is there a page already being filled? If not then signal the page fill worker thread to start working on the queued page fill requests.

When signaled from `pg_miss()`, the page fill worker thread will be awakenend and will initiate the fill operation.

**Input Parameters.** None -- The head of the ready-to-run list is assumed to be that task that caused the exception. The current task context should already be saved in the TCB of that task. No additional inputs are required.

**Assumptions**.

- It is assumed that this function is called from the level of an exception handler and that all interrupts are disabled.
- The `pg_miss()` must be "locked" in memory. Calling `pg_miss()` cannot cause a nested page fault.
- It is assumed that currently executing task (the one at the head of the ready-to-run list) is the one that cause the fault. This will always be true unless the page fault occurred in an interrupt handler. Interrupt handling logic must always be available and "locked" into memory so that page faults never come from interrupt handling.
- The architecture-specific page fault exception handling has already verified that the exception did not occur from interrupt/exception handling logic.
- As mentioned above, the task causing the page fault must not be the page fill worker thread because that is the only way to complete the page fill.

## Fill Initiation

The page fill worker thread will be awakened on one of three conditions:

- When signaled by `pg_miss()`, the page fill worker thread will be awakenend (see above),
- From `pg_callback()` after completing last fill (when `CONFIG_PAGING_BLOCKINGFILL` is defined... see below), or
- A configurable timeout expires with no activity. This timeout can be used to detect failure conditions such things as fills that never complete.

The page fill worker thread will maintain a static variable called `struct tcb_s *g_pftcb`. If no fill is in progress, `g_pftcb` will be NULL. Otherwise, it will point to the TCB of the task which is receiving the fill that is in progress.

> **NOTE**: I think that this is the only state in which a TCB does not reside in some list. Here is it in limbo, outside of the normally queuing while the page file is in progress. While here, it will be marked with TSTATE_TASK_INVALID.

When awakened from `pg_miss()`, no fill will be in progress and `g_pftcb` will be NULL. In this case, the page fill worker thread will call `pg_startfill()`. That function will perform the following operations:

- Call the architecture-specific function `up_checkmapping()` to see if the page fill still needs to be performed. In certain conditions, the page fault may occur on several threads and be queued multiple times. In this corner case, the blocked task will simply be restarted (see the logic below for the case of normal completion of the fill operation).
- Call `up_allocpage(tcb, &vpage)`. This architecture-specific function will set aside page in memory and map to virtual address (vpage). If all available pages are in-use (the typical case), this function will select a page in-use, un-map it, and make it available.
- Call the architecture-specific function `up_fillpage()`. Two versions of the up_fillpage function are supported -- a blocking and a non-blocking version based upon the configuration setting `CONFIG_PAGING_BLOCKINGFILL`.
    - If `CONFIG_PAGING_BLOCKINGFILL` is defined, then up_fillpage is blocking call. In this case, `up_fillpage()` will accept only (1) a reference to the TCB that requires the fill. Architecture-specific context information within the TCB will be sufficient to perform the fill. And (2) the (virtual) address of the allocated page to be filled. The resulting status of the fill will be provided by return value from `up_fillpage()`.
    - If `CONFIG_PAGING_BLOCKINGFILL` is defined, then up_fillpage is non-blocking call. In this case `up_fillpage()` will accept an additional argument: The page fill worker thread will provide a callback function, `pg_callback`. This function is non-blocking, it will start an asynchronous page fill. After calling the non-blocking `up_fillpage()`, the page fill worker thread will wait to be signaled for the next event -- the fill completion event. The callback function will be called when the page fill is finished (or an error occurs). The resulting status of the fill will be providing as an argument to the callback functions. This callback will probably occur from interrupt level.

In any case, while the fill is in progress, other tasks may execute. If another page fault occurs during this time, the faulting task will be blocked, its TCB will be added (in priority order) to `g_waitingforfill`, and the priority of the page worker task may be boosted. But no action will be taken until the current page fill completes. NOTE: The IDLE task must also be fully locked in memory. The IDLE task cannot be blocked. It the case where all tasks are blocked waiting for a page fill, the IDLE task must still be available to run.

The architecture-specific functions, `up_checkmapping()`, `up_allocpage(tcb, &vpage)` and `up_fillpage(page, pg_callback)` will be prototyped in `include/nuttx/arch.h`

## Fill Complete

For the blocking `up_fillpage()`, the result of the fill will be returned directly from the call to `up_fillpage`.

For the non-blocking `up_fillpage()`, the architecture-specific driver call the `pg_callback()` that was provided to `up_fillpage()` when the fill completes. In this case, the `pg_callback()` will probably be called from driver interrupt-level logic. The driver will provide the result of the fill as an argument to the callback function. NOTE: `pg_callback()` must also be locked in memory.

In this non-blocking case, the callback `pg_callback()` will perform the following operations when it is notified that the fill has completed:

- Verify that `g_pftcb` is non-NULL.
- Find the higher priority between the task waiting for the fill to complete in `g_pftcb` and the task waiting at the head of the `g_waitingforfill` list. That will be the priority of he highest priority task waiting for a fill.
- If this higher priority is higher than current page fill worker thread, then boost worker thread's priority to that level. Thus, the page fill worker thread will always run at the priority of the highest priority task that is waiting for a fill.
- Save the result of the fill operation.
- Signal the page fill worker thread.

## Task Resumption

For the non-blocking `up_fillpage()`, the page fill worker thread will detect that the page fill is complete when it is awakened with `g_pftcb` non-NULL and fill completion status from `pg_callback`. In the non-blocking case, the page fill worker thread will know that the page fill is complete when `up_fillpage()` returns.

In this either, the page fill worker thread will:

- Verify consistency of state information and `g_pftcb`.
- Verify that the page fill completed successfully, and if so,
- Call `up_unblocktask(g_pftcb)` to make the task that just received the fill ready-to-run.
- Check if the `g_waitingforfill` list is empty. If not:
  - Remove the highest priority task waiting for a page fill from `g_waitingforfill`,
  - Save the task's TCB in `g_pftcb`,
  - If the priority of the thread in `g_pftcb`, is higher in priority than the default priority of the page fill worker thread, then set the priority of the page fill worker thread to that priority.
  - Call `pg_startfill()` which will start the next fill (as described above).
- Otherwise,
  - Set `g_pftcb` to NULL.
  - Restore the default priority of the page fill worker thread.
  - Wait for the next fill related event (a new page fault).

# Architecture-Specific Support Requirements

## Memory Organization

**Memory Regions**. Chip specific logic will map the virtual and physical address spaces into three general regions:

1. A .text region containing "locked-in-memory" code that is always available and will never cause a page fault. This locked memory is loaded at boot time and remains resident for all time. This memory regions must include:
   - All logic for all interrupt paths. All interrupt logic must be locked in memory because the design present here will not support page faults from interrupt handlers. This includes the page fault handling logic and `pg_miss()` that is called from the page fault handler. It also includes the `pg_callback()` function that wakes up the page fill worker thread and whatever architecture-specific logic that calls `pg_callback()`.
   - All logic for the IDLE thread. The IDLE thread must always be ready to run and cannot be blocked for any reason.
   - All of the page fill worker thread must be locked in memory. This thread must execute in order to unblock any thread waiting for a fill. It this thread were to block, there would be no way to complete the fills!
2. A .text region containing pages that can be assigned allocated, mapped to various virtual addresses, and filled from some mass storage medium.
3. And a fixed RAM space for .bss, .text, and .heap.

This memory organization is illustrated in the following table. Notice that:

- There is a one-to-one relationship between pages in the virtual address space and between pages of .text in the non-volatile mass storage device.
- There are, however, far fewer physical pages available than virtual pages. Only a subset of physical pages will be mapped to virtual pages at any given time. This mapping will be performed on-demand as needed for program execution.

| SRAM | Virtual Address Space | Non-Volatile Storage |
|---|---|---|
| | DATA | |
| | Virtual Page *n* (*n* > *m*) | Stored Page *n* |

| | Virtual Page *n-1* | Stored Page *n-1* |
|---|---|---|
| DATA | ... | ... |
| Physical Page *m* (*m* < *n*) | ... | ... |
| Physical Page *m-1* | ... | ... |
| ... | ... | ... |
| Physical Page *1* | Virtual Page *1* | Stored Page *1* |
| Locked Memory | Locked Memory | Memory Resident |

**Example**. As an example, suppose that the size of the SRAM is 192K (as in the NXP LPC3131). And suppose further that:

- The size of the locked, memory resident .text area is 32K, and
- The size of the DATA area is 64K.
- The size of one, managed page is 1K.
- The size of the whole .text image on the non-volatile, mass storage device is 1024K.

Then, the size of the locked, memory resident code is 32K (*m*=32 pages). The size of the physical page region is 96K (96 pages), and the size of the data region is 64 pages. And the size of the virtual paged region must then be greater than or equal to (1024-32) or 992 pages (*n*).

**Building the Locked, In-Memory Image**. One way to accomplish this would be a two phase link:

- In the first phase, create a partially linked objected containing all interrupt/exception handling logic, the page fill worker thread plus all parts of the IDLE thread (which must always be available for execution).
- All of the `.text` and `.rodata` sections of this partial link should be collected into a single section.
- The second link would link the partially linked object along with the remaining object to produce the final binary. The linker script should position the "special" section so that it lies in a reserved, "non-swappable" region.

## Architecture-Specific Functions

Most standard, architecture-specific functions are declared in `include/nuttx/arch.h`. However, for the case of this paging logic, the architecture specific functions are declared in `include/nuttx/page.h`. Standard, architecture-specific functions that should already be provided in the architecture port. The following are used by the common paging logic:

`void up_block_task(FAR struct tcb_s *tcb, tstate_t task_state);`
: The currently executing task at the head of the ready to run list must be stopped. Save its context and move it to the inactive list specified by task_state. This function is called by the on-demand paging logic in order to block the task that requires the page fill, and to

`void up_unblock_task(FAR struct tcb_s *tcb);`
: A task is currently in an inactive task list but has been prepped to execute. Move the TCB to the ready-to-run list, restore its context, and start execution. This function will be called

New, additional functions that must be implemented just for on-demand paging support:

`int up_checkmapping(FAR struct tcb_s *tcb);`
: The function `up_checkmapping()` returns an indication if the page fill still needs to performed or not. In certain conditions, the page fault may occur on several threads and be queued multiple times. This function will prevent the same page from be filled multiple times.

`int up_allocpage(FAR struct tcb_s *tcb, FAR void *vpage);`
: This architecture-specific function will set aside page in memory and map to its correct virtual address. Architecture-specific context information saved within the TCB will provide the function with the information needed to identify the virtual miss address. This function will return the allocated physical page address in `vpage`. The size of the underlying physical page is determined by the configuration setting `CONFIG_PAGING_PAGESIZE`. NOTE: This function must *always* return a page allocation. If all available pages are in-use (the typical case), then this function will select a page in-use, un-map it, and make it available.

`int up_fillpage(FAR struct tcb_s *tcb, FAR const void *vpage, void (*pg_callback)(FAR struct tcb_s *tcb, int result));`
The actual filling of the page with data from the non-volatile, must be performed by a separate call to the architecture-specific function, `up_fillpage()`. This will start asynchronous page fill. The common paging logic will provide a callback function, `pg_callback`, that will be called when the page fill is finished (or an error occurs). This callback is assumed to occur from an interrupt level when the device driver completes the fill operation.