NAME

pool_init, **pool_destroy**, **pool_get**, **pool_put**, **pool_prime**, **pool_sethiwat**, **pool_setlowat**, **pool_sethardlimit** — resource-pool manager

SYNOPSIS

#include <sys/types.h>
#include <sys/pool.h>

void
pool_init(*struct pool *pool*, *size_t size*, *u_int align*, *int ipl*, *int flags*, *const char *wmesg*, *struct pool_allocator *palloc*);

void
pool_destroy(*struct pool *pp*);

*void **
pool_get(*struct pool *pp*, *int flags*);

void
pool_put(*struct pool *pp*, *void *item*);

int
pool_prime(*struct pool *pp*, *int nitems*);

void
pool_sethiwat(*struct pool *pp*, *int n*);

void
pool_setlowat(*struct pool *pp*, *int n*);

int
pool_sethardlimit(*struct pool *pp*, *unsigned n*, *const char *warnmess*, *int ratecap*);

DESCRIPTION

These utility routines provide management of pools of fixed-sized areas of memory. Resource pools set aside an amount of memory for exclusive use by the resource pool owner. This can be used by applications to guarantee the availability of a minimum amount of memory needed to continue operation independent of the memory resources currently available from the system-wide memory allocator (malloc(9)). The pool manager obtains memory by using the special-purpose memory allocator *palloc* passed to **pool_init**(), for extra pool items in case the number of allocations exceeds the nominal number of pool items managed by a pool resource. This temporary memory will be automatically returned to the system at a later time.

CREATING A POOL

The function **pool_init**() initializes a resource pool. The arguments are:

pool
Specifies the pool storage to be initialized.

size
: Specifies the size of the memory items managed by the pool.

align
: Specifies the memory address alignment of the items returned by **pool_get**(). This argument must be a power of two. If zero, the alignment defaults to an architecture-specific natural alignment.

ipl
: The interrupt protection level used to protect the pool's internals, and at what level the pool can be safely used. See spl(9) for a list of the IPLs.

flags
: The bitwise OR of zero or more of the following values:

  `PR_WAITOK`
  : The pool doesn't need to be interrupt safe. It is recommended to specify this flag if the pool will never be accessed in interrupt context.

  `PR_RWLOCK`
  : The pool will use an rwlock(9) instead of a mutex(9) for exclusion. Requires `PR_WAITOK` to be specified as well, both to **pool_init**() and on all **pool_get**() calls on this pool.

wmesg
: The message passed on to tsleep(9) if **pool_get**() must wait for items to be returned to the pool.

palloc
: The back-end allocator used to manage the memory for the pool. *palloc* may be `NULL`, in which case the pool manager chooses an appropriate back-end allocator.

DESTROYING A POOL

The **pool_destroy**() function destroys a resource pool. It takes a single argument *pp* identifying the pool resource instance.

ALLOCATING ITEMS FROM A POOL

pool_get() allocates an item from the pool and returns a pointer to it.

pp
: The handle identifying the pool resource instance.

flags
: One or more flags. Either `PR_WAITOK` or `PR_NOWAIT` must be specified to define behaviour in case the pooled resources are depleted. If no resources are available and `PR_NOWAIT` was specified, this function returns `NULL`. If `PR_WAITOK` was specified but `PR_LIMITFAIL` was not, **pool_get**() will wait until items are returned to the pool. If both `PR_WAITOK` and `PR_LIMITFAIL` were specified, and the pool has reached its hard limit, **pool_get**() will return `NULL` without waiting, allowing the caller to do its own garbage collection; however, it will still wait if the pool is not yet at its hard limit. If `PR_ZERO` was specified and an item has been successfully allocated, it is zeroed before being returned to the caller.

RETURNING ITEMS TO A POOL

pool_put() returns the pool item pointed at by *item* to the resource pool identified by the pool handle *pp*. If the number of available items in the pool exceeds the maximum pool size set by **pool_sethiwat**() and there are no outstanding requests for pool items, the excess items will be returned to the system if possible.

pp
: The handle identifying the pool resource instance.

item
: A pointer to a pool item previously obtained by **pool_get**().

If a non-interrupt safe allocator has been selected by passing the `PR_WAITOK` flag to **pool_init**(), **pool_put**() may sleep when completely unused pages are released.

PRIMING A POOL

pool_prime() adds items to the pool. Storage space for the items is allocated by using the page allocation routine specified to **pool_init**().

pool_prime()

pp The handle identifying the pool resource instance.

nitems
The number of items to add to the pool.

SETTING POOL RESOURCE WATERMARKS

A pool will attempt to increase its resource usage to keep up with the demand for its items. Conversely, it will return unused memory to the system should the number of accumulated unused items in the pool exceed a programmable limit. The limits for the minimum and maximum number of items which a pool should keep at hand are known as the high and low **watermarks**. The functions **pool_sethiwat**() and **pool_setlowat**() set a pool's high and low watermarks, respectively.

pool_sethiwat()

pp The handle identifying the pool resource instance.

n The maximum number of items to keep in the pool. As items are returned and the total number of pages in the pool is larger than the maximum set by this function, any completely unused pages are released immediately. If this function is not used to specify a maximum number of items, the pages will remain associated with the pool until the system runs low on memory, at which point the VM system will try to reclaim unused pages.

pool_setlowat()

pp The handle identifying the pool resource instance.

n The minimum number of items to keep in the pool. The number of pages in the pool will not decrease below the required value to accommodate the minimum number of items specified by this function. Unlike **pool_prime**(), this function does not allocate the necessary memory up-front.

SETTING HARD LIMITS

The function **pool_sethardlimit**() sets a hard limit on the pool to *n* items. If the hard limit is reached *warnmess* will be printed to the console, but no more than every *ratecap* seconds. Upon successful completion, a value of 0 is returned. The value EINVAL is returned when the current size of the pool already exceeds the requested hard limit.

POTENTIAL PITFALLS

Note that undefined behaviour results when mixing the storage providing methods supported by the pool resource routines.

The pool resource code uses a per-pool lock to protect its internal state. If any pool functions are called in an interrupt context, the caller must block all interrupts that might cause the code to be reentered.

CONTEXT

pool_init(), **pool_destroy**(), **pool_prime**(), **pool_sethiwat**(), **pool_setlowat**(), and **pool_sethardlimit**() can be called during autoconf or from process context.

pool_get() and **pool_put**() can be called during autoconf or from process context. If the pool has been initialised with an interrupt safe pool allocator they can also be called from interrupt context at or below the interrupt level specified by a call to **pool_init**().

RETURN VALUES

pool_get() will return a pointer to an item allocated from the pool. If `PR_NOWAIT` or `PR_LIMITFAIL` were passed as flags to the pool it may return `NULL` if there are no resources available or if the pool hard limit has been reached, respectively.

pool_prime() will return `ENOMEM` if the requested number of items could not be allocated. Otherwise, the return value is 0.

pool_sethardlimit() will return `EINVAL` if the current size of the pool exceeds the requested hard limit. Otherwise, the return value is 0.

CODE REFERENCES

The pool manager is implemented in the file *sys/kern/subr_pool.c*.

SEE ALSO

free(9), km_alloc(9), malloc(9), mutex(9), rwlock(9), spl(9)

HISTORY

The pool manager first appeared in NetBSD 1.4 and was ported to OpenBSD by Artur Grabowski <art@openbsd.org>.