

NAME

counter - SMP-friendly kernel counter implementation

SYNOPSIS

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

```
#include <sys/systm.h>
```

```
#include <sys/counter.h>
```

```
counter_u64_t
```

```
counter_u64_alloc(int wait);
```

```
void
```

```
counter_u64_free(counter_u64_t c);
```

```
void
```

```
counter_u64_add(counter_u64_t c, int64_t v);
```

```
void
```

```
counter_enter();
```

```
void
```

```
counter_exit();
```

```
void
```

```
counter_u64_add_protected(counter_u64_t c, int64_t v);
```

```
uint64_t
```

```
counter_u64_fetch(counter_u64_t c);
```

```
void
```

```
counter_u64_zero(counter_u64_t c);
```

```
int64_t
```

```
counter_ratecheck(struct counter_rate *cr, int64_t limit);
```

```
COUNTER_U64_SYSINIT(counter_u64_t c);
```

```
COUNTER_U64_DEFINE_EARLY(counter_u64_t c);
```

```
#include <sys/sysctl.h>
```

SYSCTL_COUNTER_U64(*parent, nbr, name, access, ptr, descr*);

SYSCTL_ADD_COUNTER_U64(*ctx, parent, nbr, name, access, ptr, descr*);

SYSCTL_COUNTER_U64_ARRAY(*parent, nbr, name, access, ptr, len, descr*);

SYSCTL_ADD_COUNTER_U64_ARRAY(*ctx, parent, nbr, name, access, ptr, len, descr*);

DESCRIPTION

counter is a generic facility to create counters that can be utilized for any purpose (such as collecting statistical data). A **counter** is guaranteed to be lossless when several kernel threads do simultaneous updates. However, **counter** does not block the calling thread, also no atomic(9) operations are used for the update, therefore the counters can be used in any non-interrupt context. Moreover, **counter** has special optimisations for SMP environments, making **counter** update faster than simple arithmetic on the global variable. Thus **counter** is considered suitable for accounting in the performance-critical code paths.

counter_u64_alloc(*wait*)

Allocate a new 64-bit unsigned counter. The *wait* argument is the malloc(9) wait flag, should be either *M_NOWAIT* or *M_WAITOK*. If *M_NOWAIT* is specified the operation may fail and return NULL.

counter_u64_free(*c*)

Free the previously allocated counter *c*. It is safe to pass NULL.

counter_u64_add(*c, v*)

Add *v* to *c*. The KPI does not guarantee any protection from wraparound.

counter_enter()

Enter mode that would allow the safe update of several counters via **counter_u64_add_protected**(). On some machines this expands to critical(9) section, while on other is a nop. See *IMPLEMENTATION DETAILS*.

counter_exit()

Exit mode for updating several counters.

counter_u64_add_protected(*c, v*)

Same as **counter_u64_add**(), but should be preceded by **counter_enter**().

counter_u64_fetch(*c*)

Take a snapshot of counter *c*. The data obtained is not guaranteed to reflect the real cumulative value for any moment.

counter_u64_zero(*c*)

Clear the counter *c* and set it to zero.

counter_ratecheck(*cr, limit*)

The function is a multiprocessor-friendly version of **ppsratecheck()** which uses **counter** internally. Returns non-negative value if the rate is not yet reached during the current second, and a negative value otherwise. If the limit was reached on previous second, but was just reset back to zero, then **counter_ratecheck()** returns number of events since previous reset.

COUNTER_U64_SYSINIT(*c*)

Define a SYSINIT(9) initializer for the global counter *c*.

COUNTER_U64_DEFINE_EARLY(*c*)

Define and initialize a global counter *c*. It is always safe to increment *c*, though updates prior to the SI_SUB_COUNTER SYSINIT(9) event are lost.

SYSCALL_COUNTER_U64(*parent, nbr, name, access, ptr, descr*)

Declare a static syscall(9) oid that would represent a **counter**. The *ptr* argument should be a pointer to allocated *counter_u64_t*. A read of the oid returns value obtained through **counter_u64_fetch()**. Any write to the oid zeroes it.

SYSCALL_ADD_COUNTER_U64(*ctx, parent, nbr, name, access, ptr, descr*)

Create a syscall(9) oid that would represent a **counter**. The *ptr* argument should be a pointer to allocated *counter_u64_t*. A read of the oid returns value obtained through **counter_u64_fetch()**. Any write to the oid zeroes it.

SYSCALL_COUNTER_U64_ARRAY(*parent, nbr, name, access, ptr, len, descr*)

Declare a static syscall(9) oid that would represent an array of **counter**. The *ptr* argument should be a pointer to allocated array of *counter_u64_t*'s. The *len* argument should specify number of elements in the array. A read of the oid returns len-sized array of *uint64_t* values obtained through **counter_u64_fetch()**. Any write to the oid zeroes all array elements.

SYSCALL_ADD_COUNTER_U64_ARRAY(*ctx, parent, nbr, name, access, ptr, len, descr*)

Create a syscall(9) oid that would represent an array of **counter**. The *ptr* argument should be a pointer to allocated array of *counter_u64_t*'s. The *len* argument should specify number of elements in the array. A read of the oid returns len-sized array of *uint64_t* values obtained through **counter_u64_fetch()**. Any write to the oid zeroes all array elements.

IMPLEMENTATION DETAILS

On all architectures **counter** is implemented using per-CPU data fields that are specially aligned in memory, to avoid inter-CPU bus traffic due to shared use of the variables between CPUs. These are allocated using *UMA_ZONE_PCPU* uma(9) zone. The update operation only touches the field that is private to current CPU. Fetch operation loops through all per-CPU fields and obtains a snapshot sum of all fields.

On amd64 a **counter** update is implemented as a single instruction without lock semantics, operating on the private data for the current CPU, which is safe against preemption and interrupts.

On i386 architecture, when machine supports the `cpxchg8` instruction, this instruction is used. The multi-instruction sequence provides the same guarantees as the amd64 single-instruction implementation.

On some architectures updating a counter require a `critical(9)` section.

EXAMPLES

The following example creates a static counter array exported to userspace through a `sysctl`:

```
#define MY_SIZE 8
static counter_u64_t array[MY_SIZE];
SYSCTL_COUNTER_U64_ARRAY(_debug, OID_AUTO, counter_array, CTLFLAG_RW,
    &array[0], MY_SIZE, "Test counter array");
```

SEE ALSO

`atomic(9)`, `critical(9)`, `locking(9)`, `malloc(9)`, `ratecheck(9)`, `sysctl(9)`, `SYSINIT(9)`, `uma(9)`

HISTORY

The **counter** facility first appeared in FreeBSD 10.0.

AUTHORS

The **counter** facility was written by Gleb Smirnoff and Konstantin Belousov.