[FreeBSD Taskqueue 1](#_Toc197646185)

[Key Components 1](#_Toc7165550)

[struct taskqueue 1](#_Toc1583024448)

[struct task 1](#_Toc817995894)

[Taskqueue APIs 1](#_Toc682460138)

[taskqueue\_create() 2](#_Toc881907532)

[taskqueue\_enqueue() 3](#_Toc1795330142)

[taskqueue\_drain() 3](#_Toc166574810)

[taskqueue\_free() 4](#_Toc1034487591)

[TASK\_INIT() 4](#_Toc300729845)

[Sample Program - taskqueue 4](#_Toc662584280)

[linuxkpi : workqueue to taskqueue Mapping 4](#_Toc2121781523)

[Mapping of workqueue\_struct to taskqueue: 5](#_Toc1934863492)

[INIT\_WORK() maps to TASK\_INIT() 5](#_Toc196700631)

[create\_workqueue()/alloc\_workqueue() maps to taskqueue\_create(), taskqueue\_start\_threads() 6](#_Toc5283302)

[schedule\_work()/queue\_work() maps to taskqueue\_enqueue() 7](#_Toc865822872)

[flush\_work() maps to taskqueue\_drain() 8](#_Toc104923517)

[destroy\_workqueue() maps to taskqueue\_free() 8](#_Toc1456835329)

[Sample Program - workqueue 9](#_Toc1567161523)

[linuxkpi: Delayed Work 9](#_Toc582773322)

[INIT\_DELAYED\_WORK() 9](#_Toc1492238831)

[queue\_delayed\_work()/schedule\_delayed\_work() 10](#_Toc1158465403)

[cancel\_delayed\_work\_sync() 11](#_Toc690952363)

[Sample Program – Delayed Workqueue 12](#_Toc1785303089)

# FreeBSD Taskqueue

* Bottom half mechanism that execute deferred tasks asynchronously in kernel space
* Tasks are represented by ‘struct task’
* Tasks are processed by one or more worker threads which are scheduled outside interrupt context

## Key Components

### struct taskqueue

* Manages scheduling and execution of tasks

|  |
| --- |
| struct taskqueue {  TAILQ\_HEAD( , task) tq\_queue;  struct mtx tq\_mutex;  int tq\_flags;  int tq\_active;  struct thread \*tq\_thread;  …  }; |

* tq\_queue: holds pending tasks.
* tq\_mutex: lock to ensure concurrency control
* tq\_thread: a thread or threads that executes the queued tasks when appropriate

### struct task

* The task

|  |
| --- |
| struct task {  TAILQ\_ENTRY(task) ta\_link;  int ta\_pending;  task\_fn\_t \*ta\_func;  void \*ta\_context;  }; |

* ta\_func: points to the function to be executed
* ta\_context: context/argument that is passed to ta\_func

# Taskqueue APIs

### taskqueue\_create()

* Creates a new taskqueue

**Prototype:**

|  |
| --- |
| struct taskqueue \*taskqueue\_create(const char \*name, int mflags, taskqueue\_enqueue\_fn enqueue, void \*context); |

**Parameters:**

* **name**: string representing name of the taskqueue. Useful for debugging purpose and when using diagnostic tools to monitor taskqueues.
* **mflags**: memory allocation flags that control how memory for taskqueue will be allocated.

Valid values

1. M\_WAITOK – wait for memory allocation to succeed
2. M\_NOWAIT – do not block if memory is not available

* **enqueue**: function pointer to the enqueue function. This function will be invoked when a task is enqueued to the taskqueue. It is responsible for waking up the thread that will execute the tasks
* **context**: pointer to context that will be passed to enqueue function

**Return Value:**

* On success: pointer to newly created taskqueue
* On failure: NULL

taskqueue\_start\_threads()

- Start a specific number of worker threads for a given taskqueue.

**Prototype:**

|  |
| --- |
| int taskqueue\_start\_threads(struct taskqueue \*\*tqp, int count, int pri, const char \*namefmt, ...); |

**Parameters:**

- **tqp**: pointer to taskqueue where threads will be started

- **count**: number of threads to start on the taskqueue

- **pri**: priority of the threads

PI\_NET for network processing

PI\_DISK for disk I/O

- **namefmt**: format string for naming threads.

### taskqueue\_enqueue()

* Enqueues a task to the taskqueue for execution

**Prototype:**

|  |
| --- |
| int taskqueue\_enqueue(struct taskqueue \*queue, struct task \*task); |

**Parameters:**

* **queue**: pointer to taskqueue in which task will be enqueued
* **task**: pointer to struct task that represents the task to be executed. This task should have been initialized using TASK\_INIT()

**Return Value:**

* On success: 0
* On failure: error code

### taskqueue\_drain()

* Ensures that all tasks in the queue are executed
* Waits for task to complete execution if the task has been enqueued.
* It does not remove the task from the queue, rather, it ensures the task will complete before the function returns.

**Prototype:**

|  |
| --- |
| void taskqueue\_drain(struct taskqueue \*queue, struct task \*task); |

**Parameters:**

- **queue**: pointer to taskqueue from which task needs to be drained.

- **task:** pointer to struct task that you want to drain

### taskqueue\_free()

* Destroys a taskqueue and releases any associated resources
* Typically called after all taska have been completed or drained

**Prototype:**

|  |
| --- |
| void taskqueue\_free(struct taskqueue \*queue); |

**Parameters:**

- **queue**:pointer to structtaskqueue instance to be freed

### TASK\_INIT()

* Macro to initialize a task

**Definition:**

|  |
| --- |
| TASK\_INIT(task, priority, func, context); |

**Parameters:**

- **task**: pointer to struct task instance to be initialized

- **priority**: integer value specifying task’s priority. Set to 0 if task is critical

- **func**: pointer to function that will be called when the task is executed.

- **context**: pointer to an argument passed to func when it is executed.

# Sample Program - taskqueue

Refer to sample-taskqueue/bsd\_tq.c

From procstat output, new thread “my\_taskqueue\_thread” is created

# linuxkpi : workqueue to taskqueue Mapping

In FreeBSD linuxkpi, workqueues are implemented using FreeBSD taskqueue infrastructure to allow FreeBSD to provide Linux-compatible functionality.

### Mapping of workqueue\_struct to taskqueue:

In linuxkpi, work\_struct and workqueue\_struct is essentially a wrapper around FreeBSD task and taskqueue.

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| struct workqueue\_struct {  struct taskqueue \*taskqueue;  struct mtx exec\_mtx;  TAILQ\_HEAD(, work\_exec) exec\_head;  atomic\_t draining;  }; |
| struct work\_struct {  struct task work\_task;  struct workqueue\_struct \*work\_queue;  work\_func\_t func;  atomic\_t state;  }; |

### INIT\_WORK() maps to TASK\_INIT()

In linuxkpi, INIT\_WORK() macro initializes a work\_struct, binds function to work struct that will be executed on when work is scheduled and does TASK\_INIT()

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define INIT\_WORK(work, fn) \  do { \  (work)->func = (fn); \  (work)->work\_queue = NULL; \  atomic\_set(&(work)->state, 0); \  TASK\_INIT(&(work)->work\_task, 0, linux\_work\_fn, (work)); \  } while (0) |

### create\_workqueue()/alloc\_workqueue() maps to taskqueue\_create(), taskqueue\_start\_threads()

In linuxkpi, create\_workqueue()/alloc\_workqueue() internally calls taskqueue\_create() to allocate a new taskqueue in FreeBSD.

After creating the taskqueue, call taskqueue\_start\_threads() to spawn the necessary worker threads associated with this workqueue.

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define create\_workqueue(name) \  linux\_create\_workqueue\_common(name, mp\_ncpus) |
| #define alloc\_workqueue(name, flags, max\_active) \  linux\_create\_workqueue\_common(name, max\_active) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| struct workqueue\_struct \*  linux\_create\_workqueue\_common(const char \*name, int cpus)  {  struct workqueue\_struct \*wq;  ...  wq->taskqueue = taskqueue\_create(name, M\_WAITOK,  taskqueue\_thread\_enqueue, &wq->taskqueue);  ...  taskqueue\_start\_threads(&wq->taskqueue, cpus, PWAIT, "%s", name);  ...  } |

### schedule\_work()/queue\_work() maps to taskqueue\_enqueue()

In linuxkpi, schedule\_work()/queue\_work() internally calls taskqueue\_enqueue()

- schedule\_work() enqueues work\_struct on the default system workqueue(i.e., system\_wq)

- queue\_work() enqueues work\_struct on the specific workqueue which in freeBSD is represented by a specific taskqueue

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define queue\_work(wq, work) \  linux\_queue\_work\_on(WORK\_CPU\_UNBOUND, wq, work) |
| #define schedule\_work(work) \  linux\_queue\_work\_on(WORK\_CPU\_UNBOUND, system\_wq, work) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| bool linux\_queue\_work\_on(int cpu \_\_unused, struct workqueue\_struct \*wq,  struct work\_struct \*work)  {  ...  work->work\_queue = wq;  taskqueue\_enqueue(wq->taskqueue, &work->work\_task);  ...  } |

### flush\_work() maps to taskqueue\_drain()

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define flush\_work(work) \  linux\_flush\_work(work) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| bool linux\_flush\_work(struct work\_struct \*work)  {  ...  tq = work->work\_queue->taskqueue;  taskqueue\_drain(tq, &work->work\_task);  ...  } |

### destroy\_workqueue() maps to taskqueue\_free()

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define destroy\_workqueue(wq) \  linux\_destroy\_workqueue(wq) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| void linux\_destroy\_workqueue(struct workqueue\_struct \*wq)  {  ...  taskqueue\_free(wq->taskqueue);  kfree(wq);  ...  } |

# Sample Program - workqueue

Refer to sample-workqueue/ bsd\_linuxkpi\_wq.c

From procstat output, new threads “my\_workqueue\_0” to “my\_workqueue\_8” are created

# linuxkpi: Delayed Work

In FreeBSD linuxkpi, **struct delayed\_work** provides a compatible layer to simulate Linux’s delayed workqueues, allowing functions to be scheduled to run after a specified delay.

* struct delayed\_work combines standard work\_struct with a timer to handle delay

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| struct delayed\_work {  struct work\_struct work;  struct {  struct callout callout;  struct mtx mtx;  int expires;  } timer;  }; |

### INIT\_DELAYED\_WORK()

* Implemented using a combination of *TASK\_INIT()* and *timer handling* required for delaying work

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define INIT\_DELAYED\_WORK(dwork, fn) \  linux\_init\_delayed\_work(dwork, fn) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| void  linux\_init\_delayed\_work(struct delayed\_work \*dwork, work\_func\_t func)  {  ...  dwork->work.func = func;  TASK\_INIT(&dwork->work.work\_task, 0, linux\_delayed\_work\_fn, dwork);  ...  callout\_init\_mtx(&dwork->timer.callout, &dwork->timer.mtx, 0);  } |

### queue\_delayed\_work()/schedule\_delayed\_work()

* Implemented using a combination of *taskqueue\_enqueue()* and *timer handling*.
* When timer expries, taskqueue\_enqueue() is triggered

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define queue\_delayed\_work(wq, dwork, delay) \  linux\_queue\_delayed\_work\_on(WORK\_CPU\_UNBOUND, wq, dwork, delay) |
| #define schedule\_delayed\_work(dwork, delay) \  linux\_queue\_delayed\_work\_on(WORK\_CPU\_UNBOUND, system\_wq, dwork, delay) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| Bool linux\_queue\_delayed\_work\_on(int cpu, struct workqueue\_struct \*wq, struct delayed\_work \*dwork, unsigned delay)  {  ...  callout\_reset(&dwork->timer.callout, delay, &linux\_delayed\_work\_timer\_fn, dwork);  ...  } |
| static void linux\_delayed\_work\_timer\_fn(void \*arg)  {  ...  linux\_delayed\_work\_enqueue(dwork);  ...  } |
| static void linux\_delayed\_work\_enqueue(struct delayed\_work \*dwork)  {  struct taskqueue \*tq;  tq = dwork->work.work\_queue->taskqueue;  taskqueue\_enqueue(tq, &dwork->work.work\_task);  } |

### cancel\_delayed\_work\_sync()

Drains timer and call taskqueue\_drain() to cancel any pending or running delayed work and waits for completion

**File**: sys/compat/linuxkpi/common/include/linux/workqueue.h

|  |
| --- |
| #define cancel\_delayed\_work\_sync(dwork) \  linux\_cancel\_delayed\_work\_sync(dwork) |

**File**: sys/compat/linuxkpi/common/src/linux\_work.c

|  |
| --- |
| bool linux\_cancel\_delayed\_work\_sync(struct delayed\_work \*dwork)  {  ...  linux\_cancel\_delayed\_work\_sync\_int(dwork);  ...  } |
| static bool  linux\_cancel\_delayed\_work\_sync\_int(struct delayed\_work \*dwork)  {  ...  tq = dwork->work.work\_queue->taskqueue;  ret = taskqueue\_cancel(tq, &dwork->work.work\_task, NULL);    callout\_drain(&dwork->timer.callout);  taskqueue\_drain(tq, &dwork->work.work\_task);  } |

# Sample Program – Delayed Workqueue

Refer to sample-delayed-work/ bsd\_linuxkpi\_dwq.c

From dmesg, my\_work\_handler() is executed after specified delay of 10 seconds

From procstat output, new threads “my\_workqueue\_0” to “my\_workqueue\_8” are created