Concurrency and Synchronisation

Advanced Concepts

Donato Capitella

Department of Computer Science
University of Birmingham

Linux Kernel Programming, 2011

Outline

Outline

Kernel threads (I)

- In kernel space sometimes we need
  - to carry out **background** tasks
  - to answer **asynchronous** events
- Linux provides kernel threads for these purposes
  - like user-space threads BUT
  - they live in kernel-space
- Thus, a kernel thread can
  - access kernel data structures
  - execute kernel functions

Kernel threads (I)

- In kernel space sometimes we need
  - to carry out **background** tasks
  - to answer **asynchronous** events
- Linux provides kernel threads for these purposes
  - like user-space threads BUT
  - they live in kernel-space
- Thus, a kernel thread can
  - access kernel data structures
  - execute kernel functions

Kernel threads (I)

- In kernel space sometimes we need
  - to carry out **background** tasks
  - to answer **asynchronous** events
- Linux provides kernel threads for these purposes
  - like user-space threads BUT
  - they live in kernel-space
- Thus, a kernel thread can
  - access kernel data structures
  - execute kernel functions

Kernel threads (II)

- Kernel threads are used extensively thoughout the kernel
- To see a list of all the kernel threads running
  - `ps aux`
  - kernel threads are in square brakets

```
$ ps aux
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 2888 1696 ?  Ss 14:05 0:00 /sbin/init
root 2 0.0 0.0 0 0 ?  S 14:05 0:00 [kthreadd]
root 3 0.0 0.0 0 0 ?  S 14:05 0:00 [ksoftirqd/0]
root 4 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/0]
root 5 0.0 0.0 0 0 ?  S 14:05 0:00 [watchdog/0]
root 6 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/1]

...
```

Note

- kthreadd is the default parent of all kernel threads

Kernel threads (II)

- Kernel threads are used extensively thoughout the kernel
- To see a list of all the kernel threads running
  - `ps aux`
  - kernel threads are in square brakets

```
$ ps aux
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 2888 1696 ?  Ss 14:05 0:00 /sbin/init
root 2 0.0 0.0 0 0 ?  S 14:05 0:00 [kthreadd]
root 3 0.0 0.0 0 0 ?  S 14:05 0:00 [ksoftirqd/0]
root 4 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/0]
root 5 0.0 0.0 0 0 ?  S 14:05 0:00 [watchdog/0]
root 6 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/1]

...
```

Note

- kthreadd is the default parent of all kernel threads

Kernel threads (II)

- Kernel threads are used extensively thoughout the kernel
- To see a list of all the kernel threads running
  - `ps aux`
  - kernel threads are in square brakets

```
$ ps aux
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 2888 1696 ?  Ss 14:05 0:00 /sbin/init
root 2 0.0 0.0 0 0 ?  S 14:05 0:00 [kthreadd]
root 3 0.0 0.0 0 0 ?  S 14:05 0:00 [ksoftirqd/0]
root 4 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/0]
root 5 0.0 0.0 0 0 ?  S 14:05 0:00 [watchdog/0]
root 6 0.0 0.0 0 0 ?  S 14:05 0:00 [migration/1]

...
```

Note

- kthreadd is the default parent of all kernel threads

Outline

Creating a kernel thread

- `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`
  - `fn` is a pointer to the function that the thread will execute
  - `arg` is passed to `fn` and is a pointer to some arguments (can be `NULL`)
- Flags
  - `CLONE_FS`
  - `CLONE_FILES`
  - `CLONE_SIGHAND`
- Usually `CLONE_KERNEL`
  - `#define CLONE_KERNEL (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)`

Creating a kernel thread

- `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`
  - `fn` is a pointer to the function that the thread will execute
  - `arg` is passed to `fn` and is a pointer to some arguments (can be `NULL`)
- Flags
  - `CLONE_FS`
  - `CLONE_FILES`
  - `CLONE_SIGHAND`
- Usually `CLONE_KERNEL`
  - `#define CLONE_KERNEL (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)`

Creating a kernel thread

- `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`
  - `fn` is a pointer to the function that the thread will execute
  - `arg` is passed to `fn` and is a pointer to some arguments (can be `NULL`)
- Flags
  - `CLONE_FS`
  - `CLONE_FILES`
  - `CLONE_SIGHAND`
- Usually `CLONE_KERNEL`
  - `#define CLONE_KERNEL (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)`

Daemonize()

- The first thing that should be done is call `daemonize()`
  - all the required gunge to detach thread from user resources
  - reparents the thread to `kthreadd`

```
...
static int k_thr(void *unused) {
  daemonize();
  ...
  return 0; /* end of the thread */
}
..
ret = kernel_thread(k_thr, NULL, CLONE_KERNEL);
if (ret < 0) {
  printk("Error...");
}

...
```

Outline

Waiting for the completion of a task

- Usually
  - we create threads to carry out background tasks
  - then we wait for them to complete their tasks
- There are many ways of waiting for the completion of a thread
  - can you think of a solution with semaphores?
- Completion structures
  - a clean way to wait for the termination of a thread
  - based on `struct completion` (`linux/completion.h`)

Waiting for the completion of a task

- Usually
  - we create threads to carry out background tasks
  - then we wait for them to complete their tasks
- There are many ways of waiting for the completion of a thread
  - can you think of a solution with semaphores?
- Completion structures
  - a clean way to wait for the termination of a thread
  - based on `struct completion` (`linux/completion.h`)

Waiting for the completion of a task

- Usually
  - we create threads to carry out background tasks
  - then we wait for them to complete their tasks
- There are many ways of waiting for the completion of a thread
  - can you think of a solution with semaphores?
- Completion structures
  - a clean way to wait for the termination of a thread
  - based on `struct completion` (`linux/completion.h`)

Outline

Completion structures (I)

- Initialisation
  - `DECLARE_COMPLETION(name)`
- To wait for the completion a thread calls
  - `wait_for_completion(struct completion *comp)`
  - of course, this is a blocking function if the thread has not already exited
- To complete the execution and wake up the threads waiting
  - `complete_and_exit(struct completion *comp, long code)`
  - the second argument is the return code

Note

- the desirable property of complete_and_exit() is its atomicity.

Completion structures (I)

- Initialisation
  - `DECLARE_COMPLETION(name)`
- To wait for the completion a thread calls
  - `wait_for_completion(struct completion *comp)`
  - of course, this is a blocking function if the thread has not already exited
- To complete the execution and wake up the threads waiting
  - `complete_and_exit(struct completion *comp, long code)`
  - the second argument is the return code

Note

- the desirable property of complete_and_exit() is its atomicity.

Completion structures (I)

- Initialisation
  - `DECLARE_COMPLETION(name)`
- To wait for the completion a thread calls
  - `wait_for_completion(struct completion *comp)`
  - of course, this is a blocking function if the thread has not already exited
- To complete the execution and wake up the threads waiting
  - `complete_and_exit`(struct completion *comp, long code)
  - the second argument is the return code

Note

- the desirable property of complete_and_exit() is its atomicity.

Completion structures (I)

- Initialisation
  - `DECLARE_COMPLETION(name)`
- To wait for the completion a thread calls
  - `wait_for_completion(struct completion *comp)`
  - of course, this is a blocking function if the thread has not already exited
- To complete the execution and wake up the threads waiting
  - `complete_and_exit`(struct completion *comp, long code)
  - the second argument is the return code

Note

- the desirable property of complete_and_exit() is its atomicity.

Completion structures (II)

```
static DECLARE_COMPLETION(thread_exit);
...
static int k_thr(void *unused) {
  daemonize("k_thr");
  ...
  while (keep_going) {
  ...
  }
  complete_and_exit(&thread_exit, 0);
}
...
static void __exit clean(void) {
  ...
  wait_for_completion(&thread_exit);
}
```

Outline

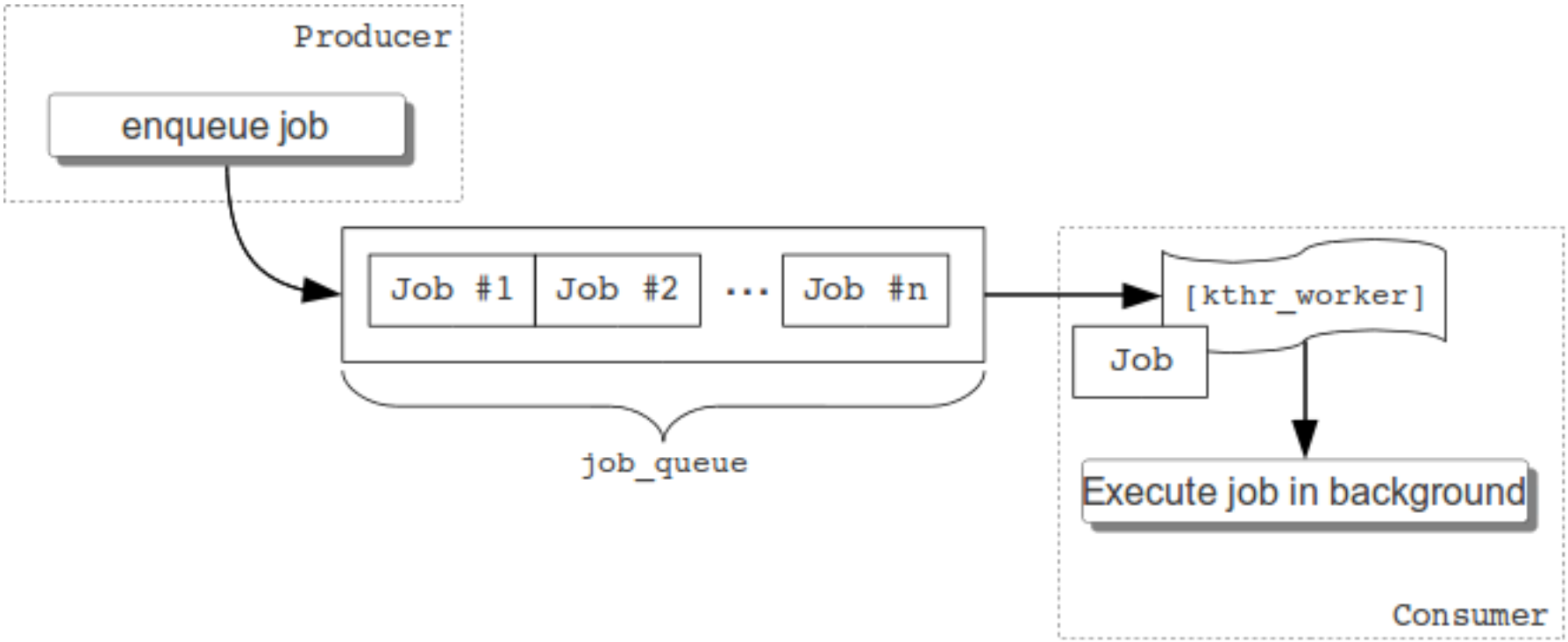
Background job execution(I)

- LKM >> `bg_job.c`
  - small kernel module
  - skeleton architecture for *background job execution*
- Based on a *shared queue of jobs* to be executed in background
  - the jobs are extracted from the queue and executed by a kernel thread `[kthr_worker]`
  - it is possible to extend the architecture with more worker threads

Background job execution(I)

- LKM >> `bg_job.c`
  - small kernel module
  - skeleton architecture for *background job execution*
- Based on a *shared queue of jobs* to be executed in background
  - the jobs are extracted from the queue and executed by a kernel thread `[kthr_worker]`
  - it is possible to extend the architecture with more worker threads

Background job execution(II)



Job structure (bg_jobs.c)

- A job consists of
  - a name
  - a pointer to a function that executes that job
  - a pointer to arguments to be passed to the previous function

```
#define JOB_NAME_MAX_SIZE 25
struct job {
  struct list_head list;
  char *name;
  void (*do_job)(void *args);
  void *args;

};
```

Note

- This skeleton implementation provides two very simple jobs, one that doesn't take arguments(hello_job) and another one that does(sleep_job).

Job structure (bg_jobs.c)

- A job consists of
  - a name
  - a pointer to a function that executes that job
  - a pointer to arguments to be passed to the previous function

```
#define JOB_NAME_MAX_SIZE 25
struct job {
  struct list_head list;
  char *name;
  void (*do_job)(void *args);
  void *args;
};
```

Note

- This skeleton implementation provides two very simple jobs, one that doesn't take arguments(hello_job) and another one that does(sleep_job).

Job structure (bg_jobs.c)

- A job consists of
  - a name
  - a pointer to a function that executes that job
  - a pointer to arguments to be passed to the previous function

```
#define JOB_NAME_MAX_SIZE 25
struct job {
  struct list_head list;
  char *name;
  void (*do_job)(void *args);
  void *args;
};
```

Note

• This skeleton implementation provides two very simple jobs, one that doesn't take arguments(hello_job) and another one that does(sleep_job).

[kthr_worker]

- Background thread that waits for jobs
  - extracts job next job from the queue
  - prints job name
  - calls the `do_job()` function on the current job with the appropriate arguments

```
static int kthr_worker(void *unused) {
  ...
  for(;;) {
    job = dequeue();
    ...
    printk(KERN_INFO "Job:  %s\n", job->name);
    job->do_job(job->args);
    kfree(job);
  }
  ...
 }
```

[kthr_worker]

- Background thread that waits for jobs
  - extracts job next job from the queue
  - prints job name
  - calls the `do_job()` function on the current job with the appropriate arguments

```
static int kthr_worker(void *unused) {
  ...
  for(;;) {
    job = dequeue();
    ...
    printk(KERN_INFO "Job:  %s\n", job->name);
    job->do_job(job->args);
    kfree(job);
  }
  ...
 }
```

Init function

- The init function
  - creates and enqueues the jobs
  - creates an instance of [kthr_worker]

```
int init_module(void) {
  ...
  enqueue(build_job("JOB #1", hello_job, NULL));
  enqueue(build_job("JOB #2", sleep_job, (void*)5));
  ...
  if (!kernel_thread(kthr_worker, NULL, CLONE_KERNEL)) {
    ...
  }
  ...
}
```

Init function

- The init function
  - creates and enqueues the jobs
  - creates an instance of [kthr_worker]

```
int init_module(void) {
  ...
  enqueue(build_job("JOB #1", hello_job, NULL));
  enqueue(build_job("JOB #2", sleep_job, (void*)5));
  ...
  if (!kernel_thread(kthr_worker, NULL, CLONE_KERNEL)) {
    ...
  }
  ...
}
```

Outline

How to stop a kernel thread?

- When the module is unloaded, we need to stop the kernel thread
  - moreover, if there are jobs in the queue, we need to wait for it to complete all of them
- A possible solution
  - insert in the queue a `STOP_JOB` job
  - call `wait_for_completion()`
- When `[kthr_worker]` extracts the `STOP_JOB` job
  - it exits its main loop
  - it calls `complete_and_exit()`

How to stop a kernel thread?

- When the module is unloaded, we need to stop the kernel thread
  - moreover, if there are jobs in the queue, we need to wait for it to complete all of them
- A possible solution
  - insert in the queue a `STOP_JOB` job
  - call `wait_for_completion()`
- When `[kthr_worker]` extracts the `STOP_JOB` job
  - it exits its main loop
  - it calls `complete_and_exit()`

How to stop a kernel thread?

- When the module is unloaded, we need to stop the kernel thread
  - moreover, if there are jobs in the queue, we need to wait for it to complete all of them
- A possible solution
  - insert in the queue a `STOP_JOB` job
  - call `wait_for_completion()`
- When `[kthr_worker]` extracts the `STOP_JOB` job
  - it exits its main loop
  - it calls `complete_and_exit()`

Summary

- Kernel threads
- Completion structures
- Architecture for background job execution