CS350: Assignment 2 – Kernel-side Synchronization

Emil Tsalapatis

1 Introduction

In this assignment we will be implementing enough kernel functionality in CastorOS to enable job control for userspace applications. The main parent process of an application often spawns multiple child processes that execute a task then exit. The main process must wait for the children to finish and exit before continuing execution. Our task is to implement the mechanism that the parent process uses to wait for the children to finish.

CastorOS currently does not implement a mechanism for a parent process to wait for children to exit. The missing API is CastorOS's `OSWait` system call that has equivalents in virtually every operating system. A parent process calls `OSWait` to query the exit status of either a specific child process or all of its children. The `OSWait` system call blocks until the child being inspected exits and notifies the parent. The semantics of the call ensure that the parent will not continue executing until the child finishes running. CastorOS is missing this call, so parent processes have no way to query the state of their children.

The `OSWait` must use synchronization between the parent and child process that is currently missing from the kernel. The CastorOS kernel does have semaphores, but we will not be using them in our own code. We will instead be coding mutexes and condition variables that we will then use for mutual exclusion and event signalling. We will then use these primitives to implement the `OSWait` call logic.

2 Overview

2.1 Job control and processes

In this exercise we will add enough code to the kernel to implement the `OSWait` system call that is needed for job control. Processes in CastorOS and UNIX (and almost all other) systems are organized into a tree where the parents point to the children. Parents retain this reference to a child in the form of a process ID (PID) used to query the status of the child process.

For example, shells receive commands from the user and spawn a new child process that the command runs in. The shell then waits until the child process by querying the child's status using its PID. Only after the child has exited does the shell resume taking input from the command line.

3 Implementing locks

We will first be implementing mutexes for locking. Mutexes are primitives used for mutual exclusion, meaning that a critical section of code between a `Lock` and `Unlock` on a lock never runs concurrently with any other critical section for the same lock.

Below is a simple example of a lock that protects data by wrapping the code that accesses it into a critical section:

```
void increment(lock *a, int num) {
   int tmp1;

   Lock(a);
   tmp1 = num;
   tmp1 += 1;
   num = tmp1;
   Unlock(a);
}

void decrement(lock *a, int num) {
   int tmp2;

   Lock(a);
   tmp2 = num;
   tmp2 -= 1;
   num = tmp2;
   Unlock(a);
}
```

The critical section prevents bugs that rise from executing these functions at the same time or by interleaving the code of these functions in a problematic way, for example, because of the scheduler forcing a context switch. An example of a buggy execution caused by the absence of locks would be the following:

```
// num is 0
tmp1 = num;
tmp2 = num
tmp1 += 1;
// num is 1
num = tmp1;
tmp2 += 1;
num = tmp2;
// num is -1 (BUG)
```

Mutexes ensure that from the point of view of the thread executing the code in blue, the code in black executes atomically and vice versa. Locks thus prevent the above interleaving and eliminate the bug.

The Mutex API is relatively minimal:

```
void Mutex_Init(Mutex *mtx, const char *name);
void Mutex_Destroy(Mutex *mtx);
void Mutex_Lock(Mutex *mtx);
int Mutex_TryLock(Mutex *mtx);
void Mutex_Unlock(Mutex *mtx);
\captionof{lstlisting}{A \NAME mutex}.}
```

Threads call into the Mutex API to manage and use the in-kernel mutexes. New mutexes are allocated by the thread and initialized by passing them `Mutex_Init`. Threads then use the mutex by calling `Mutex_Lock` to take ownership of it. The `Mutex_Lock` call blocks the thread and puts it to sleep if the mutex is already in use. The system will wake up the thread when the owner of the lock calls `Mutex_Unlock` to release ownership. A thread that wants to take the lock but does not want to block if the lock is already owned calls `Mutex_TryLock` instead of `Mutex_Lock`. The `Mutex_TryLock` calls returns failure instead of blocking.

Next we inspect the `struct Mutex` data structures that represents the mutex to better understand how to implement it. The data structure is found in `sys/include/mutex.h` and is:

```
typedef struct Mutex {
    uint64_t     status;
    Thread     *owner;
    Spinlock     lock;
    WaitChannel     chan;
    LIST_ENTRY(Mutex) buckets;
} Mutex;
\captionof{lstlisting}{A \NAME mutex}.}
```

We will inspect the structure line by line:

```
typedef struct Mutex {
    ...
} Mutex;
```

We use `typedef` keyword in C to create shorthand names for data structures. All instances of data structures in C are normally declared by the `struct` keyword, so without typedef we would be writing code like:

```
struct Mutex mtx;
```

With the above `typedef` we can use just write:

```
Mutex mtx;
```

The shorthand name does not need to match the longer form, e.g., we could use `Mtx` as an even shorter name.

```
    uint64_t     status;
```

The `status` field describes whether the mutex is held or free. The only two possible values are `MTX_STATUS_UNLOCKED` and `MTX_STATUS_LOCKED`. The structure is updated by the `Lock`, `TryLock`, and `Unlock` routines.

```
    Thread    *owner;
```

The `owner` field holds a pointer to the `Thread` structure of the thread that holds the lock. We use this structure to find bugs related to threads trying to unlock a lock they do not own. This structure is updated in tandem with the `status` variable.

```
    Spinlock     lock;
```

The `lock` field provides locking for the `Mutex` data structure itself. When using the mutex API we modify multiple data structures at once, and we must avoid races between threads trying to inspect or modify the same lock.

```
    WaitChannel     chan;
```

The `chan` variable represents the *wait channel* of the lock. A wait channel is a queue used to notify threads when the lock is released. A `Mutex_Lock` call on an already owned lock results in the thread instead registering itself with the wait channel going to sleep. A `Mutex_Unlock` call similarly wakes up a thread sleeping on the wait channel if such a thread exists while unlocking the mutex.

```
    LIST_ENTRY(Mutex) buckets;
```

The `buckets` field is a *linked list* node. Linked lists hold a variable number of elements of the same type. Unlike arrays, they grow and shrink as elements are added and removed. Please refer to the Appendix 6 for an explanation on how linked lists are implemented in practice, and for a more detailed explanation of how the above field is used.

`Mutex_Lock` and `Mutex_Unlock` The `Mutex_Lock` function repeatedly inspects the state of the mutex and until it finds it unlocked, then puts itself as the owner. The caller takes the spinlock of the mutex to prevent races, then checks whether the mutex is taken. If the mutex is free then the function sets its state to be held and the owner to be the calling thread, then releases the spinlock. If the mutex is taken then the thread gets the wait channel spinlock, releases the mutex spinlock, then goes to sleep. The function will continue executing when the thread gets woken up, at which point the thread will reacquire the spinlock and repeat the loop until it finds the mutex unlocked.

The most complex part of the locking code is properly taking and releasing the spinlocks. The order in which we lock and unlock must be the same as the

Algorithm 3.1: The `Mutex_Lock` code

```
acquire the mutex spinlock
while the lock is taken do
    get the wait channel spinlock
    release the mutex spinlock
    sleep on the wait channel
    reacquire the mutex spinlock
mark the mutex as held
mark the mutex with the new owner
release the mutex spinlock
```

one in the pseudocode to avoid subtle deadlocks and data races. For example, our code must release the spinlock before sleeping on the wait channel to avoid deadlocking. If the thread kept the spinlock during the sleep then other threads would not be able to call `Mutex_Unlock` to free the mutex because they would block on the spinlock lock. The sleeping thread loops after waking up because some other thread may have locked the mutex while the newly woken thread was trying to reacquire the spinlock. In that case the woken thread goes back to sleep and waits to be woken up again.

Taking the wait channel spinlock before unlocking the mutex spinlock is called "hand-over-hand" locking and is a common pattern when using fine-grained locking. Hand-over-hand locking is critical to avoid subtle bugs like the "lost wakeup" problem that we demonstrate in the code below:

Algorithm 3.2: The "lost wakeup" problem

```
acquire the mutex spinlock
while the lock is taken do
    release the mutex spinlock
    Mutex_Unlock() called from another thread
    get the wait channel spinlock
    sleep on the wait channel
    reacquire the mutex spinlock
mark the mutex as held
mark the mutex with the new owner
release the mutex spinlock
```

This implementation of `Mutex_Lock` differs from the correct one in that the function releases the mutex spinlock *before* locking the wait channel. In this case another thread may fully run `Mutex_Unlock` before the locking thread blocks on the wait channel. The result is that the thread goes to sleep even though the mutex is now unlocked. Even worse, threads waiting on a lock only wake up by other threads calling to `Mutex_Unlock` and notifying them that the mutex is now free. Threads running the buggy code above may miss the wakeup call and remain blocked indefinitely, effectively deadlocking.

The `Mutex_Unlock` code is simpler because it never blocks. The code takes the mutex spinlock, marks the mutex as free and wakes up any threads attempting to lock the mutex that are sleeping on the waitqueue:

Algorithm 3.3: The `Mutex_Unlock` code

```
1 acquire the mutex spinlock
2 mark the mutex as not held
3 remove the current thread as the owner
4 wake up a thread waiting on the wait channel
5 release the mutex spinlock
```

We now have all the necessary components to implement mutexes in CastorOS. The only task left is to write the C code in `sys/kern/mutex.c` that corresponds to the pseudocode above. For that we will use the APIs from `waitchannel.h` and `spinlock.h` that we can find in the `sys/include` directory.

4 Condition Variables

In this section we will be implementing condition variables (CV). CVs are a data structure that processes use to optimize thread scheduling. Condition variables have similar semantics to the wait channels data structure that is part of `struct Mutex`. Threads must sometimes wait for some system-wide event to happen, and cannot proceed otherwise. In that case they wait on a CV, putting themselves to sleep indefinitely. When the event finally takes place, the running thread that triggered the event notifies the sleeping thread by signalling to the condition variable.

In a sense we have already how condition variables are used in this very exercise. The `Mutex_Lock` code solves the same problem as CVs because it enables the following pattern:

Algorithm 4.1: An example of a CV

```
1 take lock
2 while some condition is not met do
3     atomically drop lock and wait
4     wakeup and take lock
5 do work
6 drop lock
```

The main problem is how to proceed into a critical section only when some condition in the program is met. As we saw in `Mutex_Lock` writing the above code using wait channels and spinlocks is complex and error-prone. Condition variables provide a more intuitive alternative. The definition of the data structure and its API are in `include/sys/cv.h`:

```
typedef struct CV {
    WaitChannel    chan;
} CV;

void CV_Init(CV *cv, const char *name);
void CV_Destroy(CV *cv);
void CV_Wait(CV *cv, Mutex *mtx);
void CV_Signal(CV *cv);
void CV_Broadcast(CV *cv);
```

CVs are a simple wrapper over wait channels, and have an almost identical API. The `CV_Signal` call corresponds to `WaitChannel_Wake`, and`CV_Broadcast` corresponds to `WaitChannel_WakeAll`. The only difference in the API is that the `CV_Wait` call also takes a mutex, because the call internally takes the wait channel lock, drops the mutex, enters the wait channel, then locks back the mutex before returning.

Using CVs makes it easy to avoid deadlocks and lost wakeups. The C code uses mutexes and CVs to solve the problem of waiting for a condition to become true before entering the critical section:

```
int exampleWaiter(CV *cv, Mutex *mtx) {
    Mutex_Lock(mtx);
    while (!necessaryCondition())
      CV_Wait(cv, mtx);
    doWork();
    Mutex_Unlock(mtx);
}

int exampleSignaler(CV *cv) {
    Mutex_Lock(mtx);
    setConditionToTrue();
    CV_Signal(cv, mtx);
    Mutex_Unlock(mtx);
}
```

The CV API replaces the complex locking and removes the need for the developer to code in `Mutex_Lock/Unlock` calls before and after going to sleep.

We can now fill in the CV API in `sys/kern/cv.c`. The signalling APIs are wrappers over the underlying wait channel APIs, while the wait call includes the logic for hand-over-hand locking between the mutex and the wait channel lock, while also locking the mutex back after waking back from `WaitChannel_Wait`.

5 Implementing `wait`

The system call used to query the child's status is called `wait` in most OSes. The parent calls the function with the PID of the child whose exit status it wants to inspect. If the child has exited, the call returns a status variable that includes the child's exit value. This way the parent can check whether the child executed successfully or exited because of an error. A process that calls `wait` with the PID of a child that has not yet exited will block until the child exits.

The `wait` call provides the option to wait for any child to exit instead of taking a single PID. Some processes have multiple children that may finish executing in any order. An example is a shell that is running two background processes with PIDs P_1 and P_2. The shell must notify the user when either P_1 or P_2 exits, but if it calls `wait` with P_1 and P_2 exits first the shell will not be notified until P_1 also exits. To avoid this situation `wait` takes a special value that means "return the exit status of the first child that exits. The special value is `-1` in UNIX and `0` in CastorOS.

CastorOS processes that call `OSWait` in userspace ultimately execute the `Process_Wait` call in the kernel that implements the above logic. We will be using the locks and CVs we implemented in the previous sections for our implementation. The pseudocode in the next page the describes the complete function, with the missing logic marked in red:

Our task is to use and modify the `zombieProc` queue of zombie children. CastorOS currently does not use the queue because it does not need it, as it does not implement `wait`. We will be using the `TAILQ` macros to remove exited children from the queue. For the assignment we will be working with the `zombieProc` queue of the process. This queue is where children processes register themselves when turning into zombies while exiting. Please refer to this page for more on the `TAILQ` macros.

For our implementation we use two condition variables, each used to wait on a different type of event. When calling `Process_Wait` with a PID of `0`, we use the `zombieProcCV` condition variable of the process that is making the `OSWait` call. The reason is that we are waiting for a child process to be added to the `zombieProc` list. In contrast, when the call waits on a specific process it calls `CV_Wait` on the condition variable of that process, which will be signalled when the process enters a zombie state.

We use the `zombieProcLock` of the process making the `Process_Wait` call when looking for the process. Interestingly, we pass the same lock for both `CV_Wait`, even though one of the calls uses the `zombieProcCV` field of the child being waited on. This is another example of how each condition variable does not correspond to a single mutex and only serves to make dropping and taking mutexes between waits convenient.

Algorithm 5.1: The `Process_Wait` code

```
Input: The current parent process, the PID of the child process
Output: The status of the exited process
Lock $proc.zombieProcLock$
if $childPid = 0$ then
    while $true$ do
        if $zombieProc$ $not$ $empty$ then
            $p \leftarrow$ head of $proc.zombieProc$
            remove head of $proc.zombieProc$
            break;
        wait on $proc$'s $zombieProcCV$
else
    $p \leftarrow Process\_Lookup(childPid)$
    while $p$ $is$ $not$ $a$ $zombie$ do
        wait on $p$'s $zombieProcPCV$
    remove $p$ from $proc.ZombieProc$
    Process_Release($p$)
Unlock $proc.zombieProcLock$
$status \leftarrow mkStatus(p.pid, p.exitCode)$
foreach $thread \in proc.zombieQueue$ do
    Thread_Release($thread$)
Process_Release($p$)
return $status$
```

6 Testing and Submitting Your Work

We will be submitting to the test server the same way as we did for assignment one. We use the `client.py` program with the same username and magic number. We only change the `ASST` variable from `"asst1"` to `"asst2"`. Changing this variable notifies the server that any new submissions will be run against the Assignment 2 tests. For more details on how to submit please refer to Assignment 1's Section 5.

NOTICE: The patch to be submitted to the server should include the changes made to the kernel for assignment 1. Newer versions of the `client.py` utility include all commited changes to the patch, but some older versions required all changes to be included in a a single commit. Please inspect the generated patch and ensure it includes all your code before submitting, including the code for assignment 1.

We will be evaluating our work using three tests built into the CastorOS image. These tests are `spawnanytest`, `spawnsingletest`, and `spawnmultipletest`. The source for these tests is in the `tests/` directory in the `castoros` repository. We run the tests inside CastorOS from the shell the same ways we ran `cat` and `ls` for Assignment 1.

These three tests evaluate our implementation of `OSWait` with different arguments and number of concurrent children processes. The tests create one or multiple children using the `OSSpawn` call we implemented for the previous assignment, then wait for them to be done using `OSWait`. The tests ensure that the call executes successfully, and that its return values are correct.

All three tests require `OSWait`. The `spawnsingletest` program creates a single child process, then waits for it to finish by calling OSWait with the child's PID. The test repeats this process 10 times before exiting successfully. The `spawnmultipletest` program creates 10 children at once, then waits on all of them by calling `OSWait` on each of their PIDs sequentially. The `spawnanytest` program does the same thing but instead waits 10 times for any child to exit by calling OSWait with the special PID value of 0.

Appendix: Linked Lists

There are two ways of coding a linked list for a data type `T`. The first uses an external linked list `E`, a separate data structure whose every instance holds a reference to the next element `E` in the list and a reference to an instance of type `T` that holds the actual data:

```
 struct E {
struct E *next;
struct T *data;
 }
```

The external linked list approach is requires an extra allocation for every element we add to the list and makes data management more complicated. The alternative approach, an internal linked list, embeds the data structure E inside data structure T like in the case of Mutex:

```
 struct T {
...
struct E *next;
 }
```

This approach does not require extra allocations when inserting into a list. The downside of internal linked lists is that struct E is polymorphic and is dependent on struct T, but the C language's type system does not include polymorphism. Most operating systems use the C preprocessor-related techniques to provide an internal linked list API, as is the case with CastorOS. For more details on how internal linked lists are implemented please refer to sys/include/queue.h that holds the definitions for the LIST_ENTRY .