CS350: Assignment 2 – Kernel-side Synchronization

Tavian Barnes, Emil Tsalapatis

1 Introduction

In this assignment we will be implementing the synchronization primitives and process control system calls that are missing in CastorOS. You will implement kernel mutexes and condition variables, then use them to implement the `Process_Wait` function that underpins the `waitpid` system call.

Processes in CastorOS (and almost all other UNIX-like systems) are organized into a tree where the parents point to their children. Parents can query the status of their child processes and wait for them to complete using their PID (process ID) which uniquely identifies each process in the system.

For example, shells receive commands from the user and spawn a new child process to execute each command. The shell then waits for the child process by passing its PID to `waitpid`. Only after the child exits will the shell resume taking input from the command line. Prior to this assignment, you may have seen the shell prompt print too early, like this:

```
Shell> echo hello world
Shell> hello world
```

Once you complete the assignment, the shell will be able to properly wait for its children:

```
Shell> echo hello world
hello world
Shell>
```

2 Implementing Mutexes

We will first be implementing mutexes for locking. Mutexes are short for *mut*ual *ex*clusion locks that allow us to create a critical section in our code, enclosed by `Mutex_Lock` and `Mutex_Unlock`. Only one thread will enter the critical section at a time, to prevent concurrently executing code when manipulating shared memory. Furthermore, unlike spinlocks, mutexes sleep efficiently when waiting for long periods of time.

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

```
int my_counter;

void increment(Mutex_Lock *a, int num) {
    Mutex_Lock(a);
    my_counter += num;
    Mutex_Unlock(a);
}

void decrement(lock *a, int num) {
```

```
    Mutex_Lock(a);
    my_counter -= num;
    Mutex_Unlock(a);
}
```

The critical section(s) prevent both functions from manipulating `my_counter` concurrently. Recall that even though our operation on `my_counter` is on a single line, it can be made up of multiple instructions and is not guarenteed to be atomic. Furthermore, only certain instructions guarantee atomicity; otherwise, we need mutexes or spinlocks to protect shared memory accesses.

The kernel Mutex API is made up of just a few functions:

```
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);
```

New mutexes are initialized by passing them `Mutex_Init`. Threads then use the mutex by calling `Mutex_Lock` to take ownership of it. If no other thread holds the mutex, it will be acquired immediately. Otherwise, the `Mutex_Lock` call blocks the thread and puts it to sleep. The system will wake up the thread when the owner of the lock calls `Mutex_Unlock` to release it. A thread that wants to take the lock if possible, but does not want to block if the lock is already owned can call `Mutex_TryLock` instead of `Mutex_Lock`. The `Mutex_TryLock` function returns failure instead of blocking.

The `struct Mutex` data structures that contains the state of 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;
```

Listing 1: A CastorOS mutex.

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

The `owner` field holds a pointer to the `Thread` structure of the thread that holds the lock. We use the field to help debug invalid uses of the Mutex API, such as trying to unlock a mutex the current thread does not own. threads trying to unlock a lock they do not own. This field is updated in tandem with the `status` field.

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

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 and going to sleep. A `Mutex_Unlock` call will wake up a thread sleeping on the wait channel, if such a thread exists.

The mutex implementation belongs in `sys/kern/mutex.c`. Currently, there are three lines that say

```
/* XXXFILLMEIN */
```

for each of `Mutex_Lock`, `Mutex_TryLock`, and `Mutex_Unlock`.

Tip: Each function should begin by locking the mutex's internal spinlock with `Spinlock_Lock`. Be sure to unlock it before returning.

The locking functions should check the `status` field to see if the mutex is already held (`mtx->status == MTX_STATUS_LOCKED`). If it is, `Mutex_TryLock` should return `EBUSY`. `Mutex_Lock`, on the other hand, should use the wait channel to sleep until it is unlocked. The wait channel API looks like this:

```
// Lock a wait channel before sleeping on it.
void WaitChannel_Lock(WaitChannel *wc);

// Put the current thread to sleep on a wait channel.
// The wait channel must be locked before this call,
// and will be unlocked after it returns.
void WaitChannel_Sleep(WaitChannel *wc);

// Wake up one thread currently sleeping on the wait channel.
// The wait channel should not be locked before this call.
void WaitChannel_Wake(WaitChannel *wc);
// Wake up all threads currently sleeping on the wait channel.
// The wait channel should not be locked before this call.
void WaitChannel_WakeAll(WaitChannel *wc);
```

Tip: `Mutex_Lock` should use hand-over-hand locking for the wait channel. The spinlock should be unlocked before sleeping, then re-locked afterwards.

Once the two locking functions see that the mutex is unlocked, they should lock it by setting the `status` to `MTX_STATUS_LOCKED`, and setting the `owner` to `Sched_Current()`. `Mutex_Unlock` should do the opposite: set `status` to `MTX_STATUS_UNLOCKED`, set `owner` to `NULL`, and wake up a waiting thread, if there is one.

3 Implementing Condition Variables

Next we will implement condition variables (CVs). CVs are a synchronization primitive used to wait for an arbitrary condition to become true. Condition variables have similar semantics to the wait channels that were used to implement mutexes. 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, which it will unlock while it waits, and re-lock before returning.

Using CVs makes it easy to avoid deadlocks and lost wakeups. This 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);
}
```

Fill in the CV implementation in `sys/kern/cv.c`. `CV_Signal` and `CV_Broadcast` are simple wrappers over the underlying wait channel APIs. `CV_Wait` is slightly more complicated: it must use hand-over-hand locking to lock the wait channel and unlock the mutex before sleeping.

Tip: Be sure to re-lock the mutex before returning from `CV_Wait`.

4 Implementing `waitpid`

Recall that `waitpid` is the main system call that allows a parent process to wait for a child process to exit. 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 status (either the return value of `main()` or the value it passed to `exit()`). This way the parent can check whether the child completed successfully or exited because of an error. A process that calls `waitpid` 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 a specific PID. This is useful when a process has multiple children and doesn't know which one will complete first. For example, a shell can use `wait` to print a message whenever a background process finishes:

```
$ sleep 2 & echo pid: $!
pid: 1000
$ sleep 1 & echo pid: $!
pid: 1001
$ wait
[2]  + 1001 done       sleep 1
[1]  + 1000 done       sleep 2
```

Using `wait`, the shell can print a message as soon as any background process completes. With `waitpid`, it would have to guess which one will finish first, potentially delaying the messages.

In CastorOS, the C library's `wait` and `waitpid` functions both call the `OSWait` system call wrapper. `wait` passes a pid of 0 to mean "any process." The kernel syscall implementation is in `Syscall_Wait`, which is a thin wrapper around `Process_Wait`.

`Process_Wait` is defined in `sys/kern/process.c`. Line 212 of that file currently says

```
// XXXFILLMEIN
/*
 * Dummy waitpid implementation that returns an error. Remove and replace
 * with the actual implementation from the assignment description.
 */
/* XXXREMOVE START */
return SYSCALL_PACK(ENOSYS, 0);
/* XXXREMOVE END */
```

These lines should be replaced with code that waits for a child process to exit. If `pid == 0`, it should wait for any child; otherwise, it should wait for that specific process.

Tip: In `Process_Wait`, `proc` points to the `Process` structure for the parent, and `pid` is the PID of the child to wait for (or 0). Once you complete the assignment, `p` should point to the `Process` structure of the child.

Processes that have exited but not yet been waited on are called *zombies*, so your code will make use of these fields in the `Process` structure:

```
typedef struct Process {
    ...

    // The list of zombie child processes
    ProcessQueue             zombieProc;
    // Protects the zombieProc list
    Mutex                    zombieProcLock;
    // Signaled whenever a child becomes a zombie
    CV                       zombieProcCV;
    // Signaled when *this* process becomes a zombie
    CV                       zombieProcPCV;

    ...
} Process;
```

You can start with this skeleton code and fill in the missing pieces:

```
Mutex_Lock(&proc->zombieProcLock);

if (pid == 0) {
    // TODO: wait for the zombieProc list to be non-empty
    // HINT: use TAILQ_EMPTY() and CV_Wait()

    // TODO: set p to the first zombie child
    // HINT: use TAILQ_FIRST()
} else {
```

```
    p = Process_Lookup(pid);

    // TODO: wait for p to become a zombie
    // HINT: use p->procState, CV_Wait()

    Process_Release(p);
}

// TODO: remove p from the list of zombies
// HINT: use TAILQ_REMOVE(..., siblingList)

Mutex_Unlock(&proc->zombieProcLock);
```

Tip: `git grep` will search the CastorOS repo for a regular expression. You can use it to find example uses of unfamiliar APIs, for example:

```
$ git grep --line-number procState
...
sys/kern/sched.c:144:   proc->procState = PROC_STATE_ZOMBIE;
```

This tells you that line 144 of `sys/kern/sched.c` turns a process into a zombie. Looking at the surrounding code may help you with the assignment.

5 Testing

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.

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.

6 Submitting Your Solutions

Submitting Assignment 2 works just like Assignment 0. Use `git commit` to create a single commit with your Assignment 2 solution, (on top of your previous commits with your Assignment 0 and 1 solutions). **The commit should only include sys/kern/mutex.c, sys/kern/cv.c, and sys/kern/process.c. If the commit includes any other files, the server will automatically reject the submission.** Next, follow these steps to generate and submit your Assignment 2 patch.

```
$ python client.py patch
$ python client.py submit -a asst2
```

The status of your submission can be monitored using the `client.py status` command:

```
$ python client.py status -a asst2
TOTAL: 9/9
Evaluated at 10/18/2024 12:00
=========END OF SUBMISSION=========
```

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 `Process`:

```
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` .