Concurrency analysis in `xv6`

Peyman Morteza[1] and Yating Tian[1]

[1]UW-Madison CS Department

November 23, 2021

Roadmap This paragraph contains the outline of the report. Section 1 contains our report for part I of the project. We choose two spinlock from `xv6` [1] for our analysis. Subsection 1.1, contains our analysis for the `ptable.lock` and Subsection 1.2 contains our analysis for the `tickslock`. Section 2 contains our report for Part II of the project. In Section 2, we first explain how `sleep`, `wakeup1`, and `wakeup` works. Next, in Subsection 2.1, we analyze sleep and wakeup behaviour on `wait` and `exit` and in Subsection 2.2, we analyze the sleep and wakeup behaviour on `ticks`. We benefited [2] and [1] for preparing this report.

1 Part I: Lock Analysis

In the first part of the project, we analyze the behaviour of `ptable.lock` and `tickslock`. Both of these are spinlocks. Recall that (generally) spinlocks are used to prevent race condition among threads to make sure that shared data is not accessed concurrently.

1.1 `ptable.lock`

We start by analyzing `ptable.lock`. The `ptable` structure is defined in `proc.c` and has a lock as one of its members as Listing 1 shows.

```
  struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Listing 1: `ptable.lock` structure

This is lock is initialized inside `proc.c` as Listing 2 shows.

```
  void pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

Listing 2: `ptable.lock` initialization

1.1.1 Where `ptable.lock` is used

Roughly speaking, this lock is used when some of the `xv6` systems calls needs to make a change or update on the process table and also when a process needs to be created. The lock is used in the following system calls: `allocproc()`, `userinit()`, `fork()`, `exit()`, `wait()`, `scheduler()`, `yield()`, `forkret()`, `sleep()`, `wakeup()`, and `kill()`, and also mentioned in `sched()`.

The process table protected by this lock, and any access to process table must be done with `ptable.lock` held. In most use cases, a process in the kernel mode acquires `ptable.lock`, changes the process table, and then releases the lock. We explain one special case in more detail to show the how acquiring and release works. Let us say that **T1** acquires `ptable.lock` is during context switch from a thread **T1** to **T2**, where **T1** acquires the lock, switches to the scheduler, switches to **T2**, and **T2** releases the lock. This is because process table structure is being changed all through the context switch. Otherwise, we will have race condition between **T1** and **T2**.

1.1.2 Analysis of the critical section and lock frequency of lock usage

In this subsection, we analyze the critical section and understand how `ptable.lock` is used from calling system calls we mentioned above.

`allocproc()` The critical section is included in lines 1-11 (about 10 lines of code) as illustrated in Listing 3. The critical section is a for loop that goes over all processes to check if there exist a process with state `UNUSED`. If there is one, it updates the state of the process to `EMBRYO` and sets its id (lines 9 and 10 in Listing 3) and then releases the lock. If not, the function releases the lock and return(lines 6 and 7 in Listing 3).

```
1 acquire(&ptable.lock);
2 for(p = ptable.proc;
```

```
3          p < &ptable.proc[NPROC];p++)
4 if(p->state == UNUSED)
5   goto found;
6 release(&ptable.lock);
7 return 0;
8 found:
9   p->state = EMBRYO;
10  p->pid = nextpid++;
11 release(&ptable.lock);
```

Listing 3: `allocproc()` critical section

Next, we explain why the critical section should run atomically. Assume there exist no lock around the critical section. Moreover, assume (for simplicity) there exist exactly one process with state `UNUSED` in the process table. Also assume two threads called **T1** and **T2** enter the critical section in the following way: **T1** executes the for loop (line 2 in the Listing 3) and also checks the if statement (line 4 in the Listing 3) and right after checking the if statement, the timer interrupt goes off and **T2** starts running. **T2** also executes the for loop and the if statement, and again the time interrupt goes off. Thus, both **T1** and **T2** changed the state of the single `UNUSED` process to `EMBRYO` and 0 is not returned at all in either case. However, we would like the first thread that calls `allocproc()` (say **T1**) set the state of the `UNUSED` process to `EMBRYO`, and the second thread that calls `allocproc()` returns 0. For this reason, we want to critical section run atomically by putting a lock around it. Similar reasoning shows why we want Lines 9 and 10 in Listing 3 to run atomically.

Lastly, we explain how often the `ptable.lock` in `allocproc()` is in use. `userinit()` and `fork()` call this function because they want to allocate new process. The frequency of using lock in function depends on how often we initialize new process, which is relatively low.

`userinit() and fork()` The critical section for both of these functions is only one line of code as illustrated in Listing 4. `userinit ()` function sets up the first user process, by calling the `allocproc()` successfully.

```
1 acquire(&ptable.lock);
2 np->state = RUNNABLE;
3 release(&ptable.lock);
```

Listing 4: `userinit()` and `fork()` critical section

Then, the function will eventually change the state of the user process from `EMBRYO` to `RUNNABLE`. `fork()` function has the same critical section as the `userinit()`, which is change the state of the process. We must use the lock while we change a state of a process because we also have other threads that might try to update the process state. For example, if **T1** runs Line 2 in Listing 4 and after setting the state of a process to `RUNNALE`. If bad time interrupts happens, **T2** comes in and changes the state of the same process to `SLEEPING`, then time interrupt again, back to **T1**. In such a case, state of newly created process never becomes `RUNNABLE` which is not what we want to happen.

Lastly, we explain how often the lock in `userinit() and fork()`is in use. Lock in `userinit()` was used relatively less because it only used for sets up the first user process. Lock in `fork()` was used a lot by `runcmd()` in `sh.c` which is using to fork child process to run required command in shell. Thus, the frequency of lock in those case is relatively high.

`wait()` The critical section in `wait()` is icluded in Lines 1-26 (about 17 lines of code) as illustrated in Listing 5.

```
1  acquire(&ptable.lock);
2   for(;;){
3     // Scan through table looking for exited children.
4     havekids = 0;
5     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
6       if(p->parent != curproc)
7         continue;
8       havekids = 1;
9       if(p->state == ZOMBIE){
10        // Found one.
11        pid = p->pid;
12        kfree(p->kstack);
13        p->kstack = 0;
14        freevm(p->pgdir);
15        p->pid = 0;
16        p->parent = 0;
17        p->name[0] = 0;
18        p->killed = 0;
19        p->state = UNUSED;
20        release(&ptable.lock);
21        return pid;
22      }
23    }
24    // No point waiting if we don't have any children.
25    if(!havekids || curproc->killed){
26      release(&ptable.lock);
27      return -1;
28    }
29    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
30    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
31  }
```

Listing 5: `wait()` critical section

The critical section in `wait()` does the following:

1. Checking (line 9 of Listing 5) if there is an exited child (i.e. with state `Zombie`). If so, release the lock and return the process id of the child (line 20 of Lisitng 5).

2. Check if there is any child under this parent

(Line 25 of Listing 5). If not, release the lock and returns −1(Line 27 of Listing 5).

3. If the parent have running child, put parent into sleep and the lock will be released when all children exit (Line 30 of Listing 5). (Also note that we will explain `sleep` behaviour in detail in Section 2).

We need to have lock around critical section for the following reason: If **T1** runs the if statement (line 9 in the Listing 5) and get one child process is `ZOMBIE`, the bad time interrupt happen, **T2** changed it state to `UNUSED`. **T3** run `userinit()` and using the process (for example have new process id or name). However, time interrupt happen again and jump back to **T1**, we put all of the information (process id, name, etc) back to 0. This will cause some issue.

Lastly, we explain how frequent this lock is used when `wait()` is called. File `init.c` called `wait()` with the lock in its main function in order to check the If the new created process run into `ZOMBIE` state. File `sh.c`'s `runcmd()` function use `wait()` many times in order to wait for its child process to running command. For example, the `pipe` case must wait for its command finish executing. Thus, lock in `wait()` has relatively high frequency in use because `runcmd()`, which is a parent process to wait for its child process.

`scheduler()` The critical section contains 10 lines of code as illustrated in the Listing 6.

```
// Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler),p->context);
      switchkvm();
      // Process is done running for now
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
```

Listing 6: `scheduler()` critical section

The critical section in the scheduler first goes over the entire process table and look for a process to run with state `RUNNABLE`. If find one, it does context switch to the chosen process. While switching, the newly scheduled process releases the `ptable.lock` lock and then re-acquire the lock again. Before come back from context switch, it also changes it's state. Finally cleaning up the CPU and then release the lock. There are many reason to put lock around this critical section.

1. To protect context switch when we have multiple CPUs.

2. Bad timer interrupt might lead to a not `RUNNABLE` process get scheduled.

3. Two different threads in `xv6` run the context switch function at the same time could mess up with the registers.

Lastly, we explain how frequent the lock is used when `scheduler()` is called. In File `main.c`, `mpmain()` function called `scheduler()` in order to set up CPUs to let it start running process. Thus, lock in `scheduler()` has low frequency to use because only called by each CPU.

`yield()` The critical section for `yield()` contains 2 lines of code as illustrated in Listing 7. While executing, the state of the process is changed to `RUNNABLE`, and it gives up CPU.

```
1  acquire(&ptable.lock);
2  myproc()->state = RUNNABLE;
3  sched();
4  release(&ptable.lock);
```

Listing 7: `yield()` critical section

The critical section needs to have lock around it to prevent other threads accidentally change the state of the calling thread at the same time. If for example there are no lock then after **T1** enters the critical section, the timer interrupt might happen right after line 2 (in the Listing 7) and another process **T2** might change the state of **T1** to `SLEEPING` and that is not what we want.

Lastly, we explain how frequent the lock is used when `yield()` is called. The `trap()` called `yield()` to force process to give up CPU on clock tick. Thus, the calling frequency of lock in `yield()` might depends on the frequency of user mode `trap()` to the kernel mode.

`exit()` and `forkret()` The critical section for these system calls is across multiple processes.

```
exit(void){
...
acquire(&ptable.lock);
 // Parent might be sleeping in wait().
 wakeup1(curproc->parent);
 // Pass abandoned children to init.
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
  }

  forkret(void){
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);
  ...
```

Listing 8: `exit()` and `forkret()` critical section

The code for `exit()` and `forkret()` are illustrated in Listing 8. The critical section starts with acquiring the lock in `exit()`, then wakeup the parent and clean up all abandoned children by setting the root process as parent of them. Lastly, `sched()` will do a context switch and enters back to `scheduler()` at `switchkvm()` and continue from there. So, the `ptable.lock` will be held and passed on to the next process selected by scheduler. The lock will be held until the newly switched process is interrupted. For example, interrupted by `forkret()`. Thus, the purpose of `forkret()` is to release `ptable.lock` after context switch, before returning from trap to user space.

Lastly, we explain how frequent the lock is used when `exit()` and `forkret()` is called. There are many functions called `exit()` to stop the calling process. For example, `runcmd()` in `sh.c)` and `main()` in `kill.c`. For each child process to run command, it uses the critical section between `exit()` and `forkret()`. Thus, the frequency of lock used in `exit()` and (forkret() depends on the number of child process is being used and maybe relatively high.

`sleep()` The `sleep` routine is illustrated in Listing 14. The entire sleep system call is a critical section for its input spinlock `lk` and the `ptable.lock`. After checking the current process is not `NULL` and held a valid lock `lk`, `sleep()` swap the input spinlock `lk` with `ptable.lock`. After update the channel and state (to `SLEEPING`) for the process, it give up the CPU while store sleeping channel address, the reset the channel to `NULL` (i.e. set the chan number back to 0 for the process). It will not swap if input lock `lk` is `ptable.lock` already in order to avoid locking it twice. Once it hold `ptable.lock`, it can be guaranteed that we won't miss any `wakeup` while `wakeup()` runs with `ptable.lock` locked. Thus, it's okay to release input lock `lk` and only use `ptable.lock`.

This section must acquire `ptable.lock` in order to change process state and then call `sched()` to help the scheduler update the process list without causing any race situations. We will also analyze the behaviour of `sleep()` routine in Section 2 in more detail. We will also analyze the call trace of `sleep` and `wakeup` in 2.

Lastly, we explain how frequent the lock is used when `sleep()` is called. `sleep()` is called whenever `wait()` is called. So similar discussion that we presented for `wait()` shows that it has relatively high frequency.

`wakeup(),wakeup1()` The `wakeup`, `wakeup1` routines are illustrated in Listing 16. The critical section of `wakeup()` is the entire `wakeup1()`, which goes over all processes, and to find all processes that is sleeping on the input `chan`. Then update their state from `SLEEPING` to `RUNNABLE`. We will explain the behaviour of `wakeup` and `wakeup1` routines in more detail in Section 2. The critical section needs a lock because it referenced the `p->state`. Without a lock, those reference will have race condition. For example: **T1** sets the state of `p` to `RUNNABLE` and then the timer interrupt goes off and an alternate threas **T2** set the state to `SLEEPING` which is not what we want. We will also analyze the call trace of `wakeup` in 2.

Lastly, we explain how frequent the lock is used when `wakeup()` is called. Roughly speaking, most of the time that a process goes to `SLEEPING` state by calling `sleep` routine, another will eventually wake it up by calling `wakeup()` routine so based on the discussion we had for `sleep` we can conclude that this happens relatively high. Moreover, `wakeup()` is called by `exit()` to wake the parent processes which is another indication that the frequency is relatively high.

`kill()` The critical section for `kill()` is illustrated in Listing 9. The critical section is included in lines 1-12 (about 10 lines of code).

```
1  acquire(&ptable.lock);
2    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3      if(p->pid == pid){
4        p->killed = 1;
5        // Wake process from sleep if necessary.
6        if(p->state == SLEEPING)
7          p->state = RUNNABLE;
8        release(&ptable.lock);
9        return 0;
10     }
11   }
12   release(&ptable.lock);
13   return -1;
```

Listing 9: `kill()` critical section

The critical section goes over all processes, match the `pid` with input `pid` in order to kill a specific process. Setting the `p->killed` to non zero which indicates the process is already killed. Then, it wakes up the process from sleeping if the process is in `SLEEPING` state. This section need a lock to avoid race condition. For example, if **T1** runs `kill()` and sets the state of a process is `RUNNABLE`. If bad time interrupts here, **T2** comes in and changes the state of the same process to `SLEEPING`, then time interrupt again, back to **T1**. The state of the process is never become `RUNNABLE` by **T1** which is not what we want.

Lastly, we explain how frequent the lock is used when `wakeup()` is called. In `kill.c`, it calls `kill()` to kill particular process. Thus, the frequency of lock use in `kill()` is depends on the number of process is being killed.

1.1.3 How often the lock is used

We analyzed how often `ptable.lock` is used within each system call separately in Subsection 1.1. This gives an overall measure for how often `ptable.lock` is used and released.

1.2 `tickslock`

1.2.1 Where the code is used

`tickslock` is defined and initialized in `trap.c` as illustrated in Listing 10.

```
  struct spinlock tickslock;
uint ticks;
void tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  initlock(&tickslock, "time");
}
```

Listing 10: `tickslock` initialization

`tickslock` is used in the following files in `xv6`: `trap.c`, and `sysproc.c`. Generally speaking, `tickslock` is used to protect when `xv6` decids to increment the global variable `ticks`. Recall that `ticks` is used to keep track of the machine time. Next, we analyze the critical section for each part.

1.2.2 Analysis of the critical section and lock frequency of lock usage

`trap.c` The code illustrated in Listing 11 is taken from `trap.c`

```
1       acquire(&tickslock);
2       ticks++;
3       wakeup(&ticks);
4       release(&tickslock);
```

Listing 11: `tickslock` in `trap.c`

The critical section contains two lines of code (lines 2-3 in Listing 11). Clearly, we want incrementing the `ticks` variable happens atomically and that is why we need to have a lock around this critical section.

`trap` function calls `tickslock` in the timer case (i.e. `T-IRQ0+ IRQ-TIMER`).

`sysproc.c` Next, the code illustrated in Listing 12 is taken from `sysproc.c`

```
1 int sys_sleep(void)
2 {
3   int n;
4   uint ticks0;
5   if(argint(0, &n) < 0)
6     return -1;
7   acquire(&tickslock);
8   ticks0 = ticks;
9   while(ticks - ticks0 < n){
10    if(myproc()->killed){
11      release(&tickslock);
12      return -1;
13    }
14    sleep(&ticks, &tickslock);
15  }
16  release(&tickslock);
17  return 0;
18 }
```

Listing 12: `tickslock` in `sysproc.c`

The critical section contains about 8 lines of code (lines 9-16) in Listing 12. The code in Listing 12, puts the calling process into sleep for `n` ticks on machine time (We went over the details of `sleep` routine in Section 1 and Section 2). We clearly need to have a lock around this section to make sure that the `ticks` variable is incremented correctly and not in a race condition with another process.

Whenever the user program calls `sleep` system call the `tickslock` in Listing 12 will be called as well.

Next, we have the following code that uses `tickslock` as well:

```
// return how many clock tick interrupts have occurred
// since start.
int sys_uptime(void)
{
  uint xticks;
  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
```

Listing 13: `tickslock` in `sysproc.c`

The critical section is one line of code (line 8 in Listing 13). The process that calls `uptime` system call wants to get the current value of `ticks`. We clearly need to have a lock around the critical section to make that the correct values of `ticks` is reported. Notice that `ticks` can not be incremented by a thread (say **T1**) when the `tickslock` is held by another thread (say **T2**) so having a lock makes sure the correct value of `ticks` is reported.

Whenever the user program calls `uptime` system call the `tickslock` in Listing 13 will be called as well.

1.2.3 How often the lock is used

`tickslock` needs to be acquired whenever the `ticks` variable needs to be incremented and released after that. Overall it should have high frequency.

2 Part II: Sleep and Wake-up Analysis

We start this part by explaining the behaviour of `sleep()`, `wakeup()`, and `wakeup1()` routines from `proc.c`. We then analyze two examples of how these routines are used within `xv6`.

`sleep()` routine At high-level, `sleep` routine changes the state of the calling process to `SLEEPING` until a certain event is over (e.g. waiting for another process to return, or some resource or waiting for certain amount of time). We next explain it in more detail. Listing 14 contains the code for `sleep()` routine and is taken from `proc.c`.

```
1  // Atomically release lock and sleep on chan.
2  // Reacquires lock when awakened.
3  void sleep(void *chan, struct spinlock *lk)
4  {
5    struct proc *p = myproc();
6    if(p == 0)
7      panic("sleep");
8    if(lk == 0)
9      panic("sleep without lk");
10   // Must acquire ptable.lock in order to
11   // change p->state and then call sched.
12   // Once we hold ptable.lock, we can be
13   // guaranteed that we won't miss any wakeup
14   // (wakeup runs with ptable.lock locked),
15   // so it's okay to release lk.
16   if(lk != &ptable.lock){  //DOC: sleeplock0
17     acquire(&ptable.lock);  //DOC: sleeplock1
18     release(lk);
19   }
20   // Go to sleep.
21   p->chan = chan;
22   p->state = SLEEPING;
23   sched();
24   // Tidy up.
25   p->chan = 0;
26   // Reacquire original lock.
27   if(lk != &ptable.lock){  //DOC: sleeplock2
28     release(&ptable.lock);
29     acquire(lk);
30   }
31 }
```

Listing 14: `sleep()` routine

`sleep()` takes two arguments:

1. A `void*` pointer `chan`.
2. A spinlock `lk`.

`chan` is a pointer to the event that the calling process will sleep on. Two typical examples of such an event would be a pointer to another process or pointer to an integer. `lk` is a spinlock that will be acquired after `sleep` is done. Next, we go over the code in more detail. The first two if-statements (line 6 and line 8 in Listing 14) check whether the current process and the `lk` lock are equal to `NULL` (i.e do not have address 0) or not. Next, the code check whether the `lk` is `ptable.lock`. We know from previous section that `ptable.lock` needs to be acquired when we modify the process table and this is the case here since we want to update the state of the calling process. Lines 21 and 22 in the Listing 14, sets the event that the current process is waiting on (notice that `chan` is actually a field for each process table as represented in Listing 15) and sets the status of the calling process to `SLEEPING`. Next, in Line 23 of Listing 14, `sched()` is called. This means that the calling process `p` has to be waken up in-order to continue executing line 25 and after in Listing 14, and this only happens when the event that the calling process is sleeping on is over. So in line 25 (i.e. after the event is over) the `chan` value is set to 0 (i.e. `NULL`) so the process is no longer waiting. Finally, in line 27 of Listing 14 it updates the lock status by checking whether the `lk` is the `ptable.lock` or not.

```
  // Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
```

```c
  int my_read,my_write;          // Keep track of I/O num
  struct proc *parent;           // Parent process
  struct trapframe *tf;          // Trap frame for current syscall
  struct context *context;       // swtch() here to run process
  void *chan;                    // If non-zero, sleeping on chan
  int killed;                    // If non-zero, have been killed
  struct file *ofile[NOFILE];    // Open files
  struct inode *cwd;             // Current directory
  char name[16];                 // Process name (debugging)
};
```

Listing 15: `chan` is a field for each process taken from `proc.h`

`wakeup()` and `wakeup1()` routines At high-level, `wakeup()`,`wakeup1()` go over the process table to find a process that is sleeping on a certain event and change (if found) the state of such a process to `RUNNABLE` and will make sure this happens atomically. We next explain this in more detail by going over the code. The Listing 16 contains code for the `wakeup()` and `wakeup1()` routines and is taken from `proc.c`.

```c
1 // Wake up all processes sleeping on chan.
2 // The ptable lock must be held.
3 static void
4 wakeup1(void *chan)
5 {
6   struct proc *p;
7   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
8     if(p->state == SLEEPING && p->chan == chan)
9       p->state = RUNNABLE;
10 }
11 // Wake up all processes sleeping on chan.
12 void
13 wakeup(void *chan)
14 {
15   acquire(&ptable.lock);
16   wakeup1(chan);
17   release(&ptable.lock);
18 }
```

Listing 16: `wakeup()`, `wakeup1()` routines

`wakeup1()` takes `void * chan` as an argument. `chan` is pointer to an event that some process is waiting on. In line 7 of Listing 16, `for` loop is used to pass over process table and if a sleeping process that is waiting on `chan` is found it's status will be updated to `RUNNABLE`. However, for the reason that we explained in Section 1, we would like all this happen atomically. Therefore, in `wakeup()`, `wakeup1()` is called with `ptable.lock` around it and this ensures atomicity of the wakeup process.

2.1 Sleep and wakeup on `wait()` and `exit()`

In the `wait()` and `exit()` system call, we can find parent process use `sleep()` to wait for their child, and the child use `wakeup()` to call the parent before its exit. We recall the code for `wait()` from `proc.c` in Listing 17.

```c
1  // Wait for a child process to exit and return its pid.
2 // Return -1 if this process has no children.
3 int
4 wait(void)
5 {
6   struct proc *p;
7   int havekids, pid;
8   struct proc *curproc = myproc();
9   acquire(&ptable.lock);
10  for(;;){
11    // Scan through table looking for exited children.
12    havekids = 0;
13    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
14      if(p->parent != curproc)
15        continue;
16      havekids = 1;
17      if(p->state == ZOMBIE){
18        // Found one.
19        pid = p->pid;
20        kfree(p->kstack);
21        p->kstack = 0;
22        freevm(p->pgdir);
23        p->pid = 0;
24        p->parent = 0;
25        p->name[0] = 0;
26        p->killed = 0;
27        p->state = UNUSED;
28        release(&ptable.lock);
29        return pid;
30      }
31    }
32    // No point waiting if we don't have any children.
33    if(!havekids || curproc->killed){
34      release(&ptable.lock);
35      return -1;
36    }
37    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
38    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
39  }
40 }
```

Listing 17: `wait()` system call

When the parent process calls the `wait()` (line 8 in Listing 17 points to the parent), it uses an infinite loop to find an exited child (line 10 in the Listing 17),or find out there is no more children, or find out there is one or more running children. In the last case, the parent process calls: `sleep(curproc, &ptable.lock)` (Line 38 17) in order to wait while the child is running. Here, the parent passes itself into the specific sleep channel with a process table lock. On the child side, while the child process calling the `exit()`, it acquires `ptable.lock` and wakes up its `SLEEPING` parent by `wakeup1(curproc->parent)` (Line 5 in Listing 8). The child will be able to wake up its parent by finding the sleeping channel of its parent.

2.2 Sleep and Wakeup on `ticks`

In this part we analyze the behaviour of `sleep` and `wakeup` on `ticks` The following code is taken from `sysproc.c`

```
1  int sys_sleep(void)
2 {
3   int n;
4   uint ticks0;
5   if(argint(0, &n) < 0)
6     return -1;
7   acquire(&tickslock);
8   ticks0 = ticks;
9   while(ticks - ticks0 < n){
10    if(myproc()->killed){
11      release(&tickslock);
12      return -1;
13    }
14    sleep(&ticks, &tickslock);
15  }
16  release(&tickslock);
17  return 0;
18 }
```

Listing 18: `sleep` on `ticks`

At highlevel, when process calls `sleep()` system call it will sleep for `n` ticks. As Listing 18 shows, `argint(0,&n)` (line 5) is used to pass the value of `n` and the `tickslock` is acquired in line 7. Next, in lines 9-15 a while loop is used to make the calling process into `SLEEPING` state until `n` ticks is passed. Notice that, when a process calls the `sleep` system call the &`ticks` and &`tickslock` are passed to the `sleep` routine (line 14 in the Listing 18). After this, the `tickslock` will be released (line 16). Next, we explain how `xv6` wakes up the process that is called by `sleep` routine. The following code is taken from `trap.c`

```
1  case T_IRQ0 + IRQ_TIMER:
2    if(cpuid() == 0){
3      acquire(&tickslock);
4      ticks++;
5      wakeup(&ticks);
6      release(&tickslock);
```

Listing 19: `wakeup` on `ticks`

As Listing 19 shows, in line 5, the process that is slept on `ticks` will be waken-up.

References

[1] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[2] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.