CS460 Notes on Process Management

1. Process Creation

When an OS kernel starts, it creates and runs the initial process P0, which is handcrafted or created by brute force. Thereafter, every other process is created by

```
int newpid = kfork((int (*fptr)(),  int priority);
```

which creates a new process with a specified priority to execute a function fptr(). When creating a new process, the creator is the parent and the newly created process is the child. In the PROC structure, the field ppid records the parent process pid, and the parent pointer points to the parent PROC. Thus, the processes form a family tree with P0 as the root.

2. Process Termination

In a multitasking system with dynamic processes, a process may terminate or DIE, which is a common term of process termination. A process may terminate in two possible ways:

. **Normal termination:** The process has completed its task, which may not be needed again for a long time. To conserve system resources, such as PROC structures and memory, the process calls kexit(int exitValue) in kernel to terminate itself, which is the case we are discussing here.

. **Abnormal termination:** The process terminates due to an exception, which renders the process impossible to continue. In this case it calls kexit(value) with a unique value that identifies the exception.

In either case, when a process terminates, it eventually calls kexit() in kernel. The general algorithm of kexit() is as follows.

```
/*************** Algorithm of kexit ******************/
 kexit(int exitValue)
 {
    1. erase process user-mode context, e.g. close file descriptors,
       release resources, deallocate user-mode image memory, etc.

    2. dispose of children processes, if any.
    3. record exitValue in PROC.exitCode for parent to get.
    4. become a ZOMBIE (but do not free the PROC)
    5. wakeup parent and, if needed, also the INIT process P1.
    6. switch process to give up CPU
 }
```

So far, our system does not yet have USER mode. So we begin by discussing Step 2 of kexit(). Since each process is an independent execution entity, it may terminate at any time. If a process with children terminates first, all the children of the process would have no parent anymore, i.e. they become orphans. The question is then: what to do with such orphans? In human society, they would be sent to grandma's house. But what if grandma already died? Following this reasoning, it immediately becomes clear that there must be a process which should not terminate if there are other processes still existing. Otherwise, the parent-child process relation would soon break down. In all Unix-like systems, the process P1, which is also known as the INIT process, is chosen to play this role. When a process dies, it sends all the orphaned children, dead or alive, to P1, i.e. become P1's children. Following suit, we shall also designate P1 as such a process. Thus, P1 should not die if there are other processes still existing. The remaining problem is how to implement Step 2 efficiently. In order for a dying process to dispose of children, the process must be able to determine whether it has any child and, if it has children, find all the children quickly. If the number of processes is small, both questions can be answered effectively by searching all the PROC structures. For example, to determine whether a process has any child, simply search the PROCs for any one that is not FREE and its ppid matches the process pid. If the number of processes is large, e.g. in the order of hundreds, this simple search scheme would be too slow. For this reason, most large OS kernels keep track of process relations by maintaining a process family tree.

3. Process Family Tree

Typically, the process family tree is implemented as a binary tree by a pair of child and sibling pointers in each PROC, as in

```
                struct proc *child, *sibling, *parent;
```

where child points to the first child of a process and sibling points to a list of other children of the same parent. For convenience, each PROC also uses a parent pointer pointing to its parent. With a process tree, it is much easier to find the children of a process. First, follow the child pointer to the first child PROC. Then follow the sibling pointers to traverse the sibling PROCs. To send all children to P1, simply detach the children list and append it to the children list of P1 (and change their ppid and parent pointer also). Because of the small number of PROCs in all the sample systems of this book, we do not implement the process tree. This is left as a programming exercise. In either case, it should be fairly easy to implement Step 2 of kexit().

Each PROC has a 2-byte exitCode field, which records the process exit status. In Linux, the high byte of exitCode is the exitValue and the low byte is the exception number that caused the process to terminate. Since a process can only die once, only one of the bytes has meaning. After recording exitValue in PROC.exitCode, the process changes its status to ZOMBIE but does not free the PROC. Then the process calls kwakeup() to wake up its parent and also P1 if it has sent any orphans to P1. The final act of a dying process is to call tswitch() for the last time. After these, the process is essentially dead but still has a dead body in the form of a ZOMBIE PROC, which will be buried (set FREE) by the parent process through the wait operation.

4. Wait for Child Process Termination

At any time, a process may call the kernel function

int pid = kwait(int *status)

to wait for a ZOMBIE child. If successful, the returned pid is the ZOMBIE child pid and status contains the exitCode of the ZOMBIE child. In addition, kwait() also releases the ZOMBIE PROC back to the freeList, allowing it to be reused for another process. The algorithm of kwait is

```
int kwait(int *status)
{
    if (caller has no child)
       return -1 for error;
    while(1){          // caller has children
       search for a (any) ZOMBIE child;
       if (found a ZOMBIE child){
           get ZOMBIE child pid;
           copy ZOMBIE child exitCode to *status;
           release child PROC to freeList as FREE;
           return ZOMBIE child pid;
       }
       ksleep(running);  // sleep on its PROC address
    }
}
```

In the kwait algorithm, the process returns -1 for error if it has no child. Otherwise, it searches for a (any) ZOMBIE child. If it finds a ZOMBIE child, it collects the ZOMBIE child pid and exitCode, releases the ZOMBIE PROC to freeList and returns the ZOMBIE child pid. Otherwise, it goes to sleep on its own PROC address, waiting for a child to terminate. Correspondingly, when a process terminates, it must issue

kwakeup(parent); // parent is a pointer to parent PROC

to wake up the parent. When the parent process wakes up in kwait(), it will find a dead child when it executes the while loop again. Note that each kwait() call handles only one ZOMBIE child, if any. If a process has many children, it may have to call kwait() multiple times to dispose of all dead children. Alternatively, a process may terminate first without waiting for any dead child. When a process dies, all of its children become children of P1. As we shall see later, in a real system P1 executes an infinite loop, in which it repeatedly waits for dead children, including adopted orphans. Instead of sleep/wakeup, we may also use semaphore to implement the kwait()/kexit() functions. Variations to the kwait() operation include waitpid and waitid of Linux [Linux man page, 2016], which allows a process to wait for a specific child by pid with many options.