DESCRIPTION OF SYSTEM CALL IMPLEMENTATION

---

1. getppid()

First, we acquire the lock corresponding to the currently running process ***p*** (accessed using *myproc()*).
If the parent of *p* exists, we return its pid using *p->parent->pid*. Else, we return *-1*.
Lastly, we release the lock corresponding to *p* and return its *ppid*.

---

2. yield()

The implementation for yield system call is similar to the yield function defined in *kernel/proc.c*. First, we acquire the lock corresponding to the currently running process. Then, we change its state to *RUNNABLE*. Finally, we call the *sched()* function, and release the earlier acquired lock.

---

3. getpa()

Given a virtual address *va*, we call the *walkaddr()* function with the pagetable of the current process, and *va* as the arguments. The *walkaddr()* call will return the physical address *pa* corresponding to the virtual address *va*.

---

4. forkf()

This call is similar to the *fork()* call but before returning to the child and parent processes, we need to execute the passed function *f*.

First, this value is read in the *sys_forkf().* Then, the child(new) process is allocated. The memory and user registers are copied to the new process. We place the value of *0* in the *a0* register (*np->trapframe->a0*) so that *fork()* returns *0* in the child process.

In the end, we point the program counter of the child process to *f* (*np->trapframe->epc)* so that *f* is executed before moving ahead. In the end, we return the *pid* of the child process as the return value for the parent (calling) process.

<u>Output Explanation:</u>

1. When return value of *f* is 0
   `Hello world! 100`
   `4: Child.`
   `3: Parent.`
   The value in the *a0* register is initialized to *0* by *sys_forkf()* in *proc.c.* Then, we point the program counter to execute the function *f*, which then returns with *0*. Hence, the value in *a0* remains the same. Since this value is *0*, the behavior is like that of a normal *fork()* call. Since the parent calls *sleep(1)*, it allows the child to complete its execution before the parent comes out. The child process prints from the function *f* (which calls *g*) and then executes the *x=0* part in the *main()* function. At the end, the parent prints from the *x>0* part.

2. When return value of *f* is 1
   `Hello world! 100`
   `4:3 : PPaarernt.en`
   `t.`
   Before returning to the calling function, the child prints the first line of output from the function *f.* This time, the value in the *a0* register gets changed to 1. The child receives the return value *1* and hence, both parent and child execute the *x>0* part in the *main()* function. Both sleep and try to print the statement simultaneously and hence the output is shuffled.

3. When return value of *f* is -1
   `Hello world! 100`
   `Error: cannot fork`
   `Aborting...`
   `3: Parent.`
   After printing from the function *f*, the value in the *a0* register is changed to *-1*. Thus, the child executes the *x<0* part, printing the second & third line of output. At the end, the parent prints from *x>0*.

4. When the return value of *f* is some integer *y>1*
   `Hello world! 100`
   `43::  PPaarernentt..`

   As *y>0*, the output is similar to when *y=1.* The outputs are shuffled in a different way, which is expected, as both the processes are printing simultaneously.

5. When the return value of *f* is some integer *y<-1*
   `Hello world! 100`
   `Error: cannot fork`
   `Aborting...`
   `3: Parent.`
   As *y<0*, the output is the same as when *y=-1.*

6. When the return type of *f* is changed to *void* and the return statement in *f* is commented
   `Hello world! 100`
   `43:: PPaarreenntt..`

   The argument of *forkf()* is changed to *void* so that there are no errors.
   The child returns a positive value in the *forkf()* call because of the return value of *fprintf()* in the function *f*. *fprintf()* returns the number of bytes that are printed. Hence, this case becomes similar to the second one. The outputs are shuffled in a different way, which is expected.

---

5. waitpid()

Given the *pid* of the function and a pointer, this system call waits for the child process with the given *pid*.
If the passed value of *id = -1*, we simply call *wait(ptr)* since this scenario will be the same as a normal wait call.
Else, we traverse over the process table until we find a process (say ***np***) which is the child of the currently running process(say ***p***), and whose *pid* matches the one passed as the argument.

Now, if *np* is a "*zombie*" process (i.e. it is already terminated), we clear the resources allocated to *np*, and return its *pid*. Else, we continue scanning the process table until we get a terminated child of process *p*. This is implemented using the outer "*for(;;)*".

At any instant, if we find that *p* has no kids, or p itself gets killed, we return *-1*.

---

6. ps()

We define *ctime, etime, stime* in *struct proc* which account for the creation time, end time, and start time of the process.

Creation time is assigned at the time of process allocation.
Execution time is assigned the value = endtime - starttime for an already terminated process. Else, it is assigned the value = current time - start time.
End time for the process is assigned at the time of exit() call from the function.

To get the ppid, we first check whether the parent exists or not. If it exists, we acquire the parent's lock and get its pid to store in ppid. Else, we store ppid = -1.
Finally, we print these values as asked in the problem statement.

---

7. **pinfo()**

We handle the case *pid=-1* separately. A new header file *procstat.h* is created for this purpose, in which we define all the parameters required to be returned. Then, in the *pinfo()* function, the corresponding parameters of the *struct procstat (pstat)* are set. The state and command parameters are set using the *safestrcpy.* Now, we need to copy the data in kernel memory to the user memory, which is done using the *copyout* function. We also maintain a *found* variable to check if the pointers and pids are valid or not.
