CS344 Assignment - 4

Process management in Linux

190101061 - P SAI ASHRRITTH

Presentation Link:- youtu.be/SGtBPo8y9dA

Terminating a foreground process from the shell, Ctrl+C key:

In Linux, when a foreground process is running, we can use the Ctrl+C key to terminate that process. Now, when this key is pressed, the shell sends a SIGINT signal to the foreground process. Now, this signal is processed by the parent shell and it is passed down to the children with a default action to terminate. There are nearly 30 signals in Linux operating system. The default action for handling each signal is defined in the kernel, and usually, it terminates the process that received the signal. A similar thing happens when several processes run in the foreground via pipelines.

Now, to adopt this feature to xv6, we won't be creating signals (as xv6 doesn't signal-based Inter-process communication) but rather work with what does original xv6 offers. We will discuss the implementation of a signal-based IPC just like Linux in the coming sections.
To implement this feature, we would do the following steps:

- First, we will create a system call named fgkill(). We will follow the subsequent steps we follow generally while creating a system call.
- Now we will define this sys_fgkill() kill in sysproc.c where it will call another function declared in def.h and defined in proc.c named fgkill(). (note: both are different functions)
- The fgkill() in proc.c is a local function as we cannot access ptable in sysproc.c we implemented this function here. This function will iterate through the ptable and kills the parent foreground process. We also have to make sure that the inti and sh process are not killed by this.
- Now, to call the system call when we get ctrl + c input we have to add some lines in console.c. We should add a case in consoleintr(), and if that case is true, then we will call the fgkill system call.

Now, this approach has some flaws but works on the original xv6, as it doesn't have any extra user background processes running. But if we take the xv6 from assignment-3 (where we implemented swapping out and in), we have several background kernel processes running, this approach may kill such background processes.
To ensure that doesn't happen we can try another approach.

- When a command is entered in Shell, it forks and creates a child that takes the role of running the given command. Now, the shell goes into a sleeping state while holding the PID of the child i.e the running process itself (uses exec to invoke the given process).
- Now, In proc structure, add another parameter let's say named fg, that marks, if the process is foreground process or not. Now, this is done via a system call from the shell, before the execution of the process starts. We can use a semaphore to ensure this order.
- This system call, let's say named fgid(), takes PID from the parent shell after the creation of the process and calls a function defined proc.c which changes the proc->fg value to true.
- Now, we also have to modify fork() so, that when a foreground process creates a child, its child also inherits the proc->fg value.
- Now, as said in precious implementation fgkill() will kill all the functions with proc-<fg value equals to true. In this way, we could kill the foreground process effectively in any environment.

Now, Using signals to do these operations would be a more effective way to kill a process or send it to sleep. We will see the implementation of signals in the coming sections.

Displaying all the running processes:

In Linux, we have a “ps” command which lists out all the processes a user is running in the background and foreground in the corresponding window. Now, its working is pretty straightforward, it just displays all the information of process in foreground group and background group maintained by its terminal window. Now as the name suggests, the foreground group consists of the foreground process and the background processes consist of background processes. It is used by users mainly to see which background processes they are running.

Now, in further implementations, we are going to add a feature for processes to run in the background, so having this command will be helpful.

Implementation of this command follows as:

- First, we will create a file jobs.c which calls a system call name printProc().
- We will implement this system call printProc() by following the steps to add a system a call. Now in sysproc.c() it will call a function defined in proc.c also named printProc().
- Now, in this printProc() present in proc.c would have access to the ptable and will loop through all the processes while displaying the information of the processes which are in either running, sleeping or runnable state.
- Add this jobs.c as a user program to the makefile.

Maintaining state lists instead of a ptable like Linux:

One important point we have to is that Linux doesn’t store all processes in the same array(ptable) as xv6. Rather it has separate lists to store processes of a particular state. RUNNING & RUNNABLE, INTERRRUPTABLE_SLEEP, UNINTERRRUPTABLE_SLEEP, STOPPED, ZOMBIE, are the states in Linux, a bit different from the states in xv6. Usage of lists is done to increase the efficiency of runtime in state transitions.

Now, we will implement this kind of approach to store process states. (Note, we are not changing the process states, the states will remain the same as running, runnable, embryo, unused, zombie. Changing them won’t have any impact on the performance as they are sufficient to maintain all process functions throughout the OS). The main aim of this implementation is to divide and store the process according to their process states.

First, we will create a struct StructLists to store the header of a respective state processes list. This will be added to the ptable struct instead of a proc array.

```
struct StateLists {
        struct proc* ready;
        struct proc* free; // or also unused
        struct proc* sleep;
        struct proc* zombie;
        struct proc* running;
        struct proc* embryo;
};
```

In this struct, we will store the heads of the process state list only. Not the whole array.

Now, we will add a struct proc * next parameter was added to the proc structure to support linking the processes together in the lists. This will help in finding the next process to run in the scheduler, and this is the logic we use to iterate through a list.

After this, we will create a few generic functions that help in the transition of processes from one state list to another. They claim the locks and transfer a process from one state to another, by iterating through the list using proc->next, and removing them from the list or adding them in the list.

As u can see, we are using a Linked list as a data structure to store the processes. So, a lot of care and synchronization needs to be taken while writing the code of the generic function, otherwise, some processes may be lost in lists or intertwining of list and circling also could happen, which could be fatal for os and several measures would be needed to prevent this.

All state transitions follow the procedure of obtaining the ptable lock, removing the process from its list through a generic function, adding the process to the new list through a generic function, then releasing the ptable lock.

All generic functions accessing a list assert that the lock is held and panic if it is not - this eliminates race conditions within the xv6 state lists and supports atomicity of state transitions which in part helps to ensure our list invariant is maintained. (Note: The functions which directly access the stateList to change process state are mentioned as Generic functions in this paper.)

Now, lets apply and use this structure for state changes throughout xv6.

<u>Transition to free/unused:</u>
If the kernel fails to allocate memory for the process in allocproc() it is removed from the embryo list and placed on the free list
If copying a process into p fails in fork() it is removed from the embryo list and placed on the free list.
In wait() a zombie process may transition from the zombie list and be placed on the free list.

<u>Transition to Embryo:</u>
If allocproc() is successful in finding an UNUSED process, it is removed from the free list and added to the embryo list

<u>Transition to Ready/Runnable:</u>
In userinit() initproc is removed from the embryo list and added to the ready list.
In fork() if a process is created it is removed from the embryo list and added to the ready list.
In yield(), which is invoked by a process or through an interrupt, a process is moved from the running list and added to the ready list.
In wakeup1() each process sleeping on the channel will be removed from the sleep list and moved to the ready list.
In kill() the process being killed, if it is sleeping, will be removed from the sleep list and moved to the ready list.

<u>Transition to Running:</u>
If the scheduler is successful in finding a process at the head of the ready list, it is removed and added to the front of the running list.
One important feature in this list is that FIFO should be maintained.

<u>Transition to Sleep:</u>
In sleep() the process calling it is running so it is removed from the running list and added to the sleep list.

Transition to Zombie:
In exit() the process exiting is running so it is removed from the running list and added to the zombie list.

Now, we still have some functions to change which depend on ptable for process information.

Wait():
Now in wait, we will use a method hasChildren(struct proc * p) to search the embryo, ready, sleep, and running lists for children of the parent. Let's say, hasChildren calls a function findChild(struct proc* stateList, struct proc* parent) to identify whether a list contains a child of the process in a respective stateList. The function will return as soon as the child is found. The number of processes inspected is reduced by the size of the free list. The zombie list is treated differently since if a child is found it is reaped by removing it from the zombie list and adding it to the free list.

Kill():
Now in Kill, we will use the generic function findProcess(struct proc** p, int pid) to obtain the process with the given PID. findProcess just calls getProcesson the embryo, ready, running, sleep, and zombie lists, which looks for the given process in the given list. kill obtains the process through the pointer argument and proceeds to handle killing it appropriately. Since kill no longer uses the proc table, and instead uses the lists its efficiency has increased to O(NP ROC − |free|).

wakeup1():
Now in wakeup1, we only need to use the sleep list to search for processes to wake up and transition them from the sleep list to the runnable list. The search is done in-line. This efficiency is now O(|sleep|).

exit():
Now in exit(), we will abandon its children by calling a function on the embryo, ready, running, sleep, and zombie lists. This function takes parent PID and stateList as an argument and searches through the given list and abandons all the parent's children, by changing their PPID, to initproc. If the child is in the ZOMBIE state, initproc is woken. There is a slight efficiency performance in this modification as exit no longer has to look through the entire process table - the number of processes inspected is now reduced by the size of the free list NP ROC − |free|.

Now, these are some of the functions which are replaced. Now there are other functions that need to be modified to use the new list data structure.

Although this implementation doesn't give a heavy advantage for xv6, as xv6 is limited to 64 functions. But in modern operating systems like Linux, should manage 1024 or more processes. So, usage of Lists would save a lot of time for process state shifting and CPU scheduling, killing a process, etc.

Shut-down:

In qemu, xv6 can be closed either pressing ctrl+a+x in the Linux terminal where it sends a terminate signal to qemu and it abruptly closes xv6. In other words, it powers off the xv6 system. This isn't the right way to close xv6.

In Linux, The shutdown command brings down the system in a secure way. All the logged-in users are notified about the system shutdown. Signal SIGTERM notifies all the processes that the system is going down so that processes can be saved and exited properly. Command shutdown signals the init process to change the runlevel.

Now, there is already a port in xv6, which tells the os system to terminate. So we will add a system call named shutdown(). In this system call, we send a signal to the port. The line we have to add is outb(0xf4, 0x00).

Now in the next step, we have to create a user program named shutdown. In this, we will call the system call shutdown, which will send a signal to the operating system saying to shutdown and then the user program will exit. Now add this user program to the makefile for compilation.

Inter-Process Communication (SIGNALS):

Signals are one of the oldest inter-process communication methods used by Unix systems. They are used to signal asynchronous events to one or more processes. A signal could be generated by a keyboard interrupt or an error condition such as the process attempting to access a non-existent location in its virtual memory. Signals are also used by the shells to signal job control commands to their child processes. There are nearly 30 IPC signals in Linux.

Linux implements signals using information stored in the task_struct for the process. The number of supported signals is limited to the word size of the processor. Processes with a word size of 32 bits can have 32 signals whereas 64-bit processors like the Alpha AXP may have up to 64 signals. The currently pending signals are kept in the signal field with a mask of blocked signals held in blocked. With the exception of SIGSTOP and SIGKILL, all signals can be blocked. If a blocked signal is generated, it remains pending until it is unblocked.

Linux also holds information about how each process handles every possible signal and this is held in an array of sigaction data structures pointed at by the task_struct for each process. Amongst other things, it contains either the address of a routine that will handle the signal or a flag that tells Linux that the process either wishes to ignore this signal or let the kernel handle the signal for it. The process modifies the default signal handling by making system calls and these calls alter the sigaction for the appropriate signal as well as the blocked mask.

Signals are not presented to the process immediately after they are generated., they must wait until the process is running again. Every time a process exits from a system call its signal and blocked fields are checked and, if there are any unblocked signals, they can now be delivered. This might seem a very unreliable method but every process in the system is making system calls, for example, to write a character to the terminal, all of the time. Processes can elect to wait for signals if they wish, they are suspended in a state Interruptible until a signal is presented.
The Linux signal processing code looks at the sigaction structure for each of the current unblocked signals.
Linux uses both SIGNALS and PIPES for inter-process communication.

Now, let’s implement an IPC for xv6 using signals.

First, we have to add some parameters to the proc struct to support signal handling.

– Signal Handlers: It stores function pointers to signal handlers for that process, i.e if another process sends this process a signal with sig num then the Signal Handler is implemented which is pointed by the stored function pointer. Pointer is set to 0, if no signal handler is set.

– Signal Queue: A queue to store the pending signals for that process. It stores sig num and sig arg for each pending signal.
The queue is implemented using an array with a constant size of 256.
(circular array implementation like we did in assignment 2A, to store history).

```
struct sig_queue {
        struct spinlock lock ;
        char sig_arg [ SIG_QUE_SIZE ][ MSG SIZE ];
        int sig_num_list [ SIG_QUE_SIZE ];
        int start; // works as a channel (for sleep and wakeup ) also
        int end ;
};
```

Now, in order to implement signals, we have to have a signal handler.

We have to define a function signature for the signal handler. We could define something like this for that, *typedef void (* sighandler_t) (void *).* This takes a void pointer as an argument and returns void.

Now, let's implement system calls for signal handling.

<u>sig_set():</u>
This should take function pointer and signal handler function pointer.
It will set the Signal handler for the signal number type. For now, we will implement 3 signals, so the signal number type will be between 0 and 3.
The system call sig set is called by the receiver to set the signal handler's function pointer. The system call fork copies the parent's signal handlers to the child's process. The system call exec resets the signal handlers to default value 0. The function allocproc resets the sig queue to an empty queue.

<u>sig_send():</u>
This should take destination and source PID and an 8-byte message as arguments.
It will send a signal to dest_pid process ' sig_num Signal Handler with argument sig_arg which points to an 8 bytes message.
The sender calls syscall sig send for sending a signal to process with pid dest pid. The receiver process will be implemented it's signal handler indexed with sig num with argument sig arg, only if there is a signal handler set to sig num by the receiver already.
This function finds the pointer of the destination process and writes the signal message in its struct parameter.

<u>sig_pause():</u>
It blocks the process until a signal is received. It pushes it into a sleeping state.

<u>sig_ret():</u>
Syscall for returning from signal handling. This is called by a wrapping code on the stack.

Now, to send a signal;

First, we will set the signal handler function using set_sig() system call. Then we can send the signal using sig_send().

Now, let's see how a signal handler handles this signal:

First, whenever a process is about to return from kernel mode to user mode with function call trapret (implemented in trapasm.S) it checks the signal queue by calling a function to execute signal handler, let's say named exe_sh.

Function exe_sh does the following:

- If there is no signal available for this process, then it returns back to trapret.
- Setting up user stack: (for calling the signal handler)
    1. For saving the user mode's context, it saves the trapframe of this process from the kernel stack to the user-mode stack.

2. It changes the register %eip (stored in the trapframe) to the function pointer of the signal handler.
3. We can push the parameter sig arg on the stack, but this pointer points to an address in kernel space(which can't be accessed by the signal handler which is implemented in the user mode), so we first push 8 bytes of the message on the stack and then we push the pointer to the first byte pushed on to the stack as a pointer.
4. Returning back to kernel mode is necessary to retrieve back the user-mode context.
5. For returning back from the user mode to kernel mode, we have to make a syscall sig ret but the signal handler might not call this syscall due to other interrupts. To solve this problem we wrap up the system call code(written in x86 assembly) on the stack and call it implicitly by pushing the return address for the signal handler as the wrapped code's first instruction
6. Return back to trapret function to call user mode's signal handler.

Returning from Signal Handling:

- When the signal handler executes the wrapped code on the stack, it calls sig ret syscall.
- sig ret syscall just save the trapframe back from the user stack to the kernel stack which retrieves back the original user mode context.

Now, this is the rough diagram of how the signal handler works,

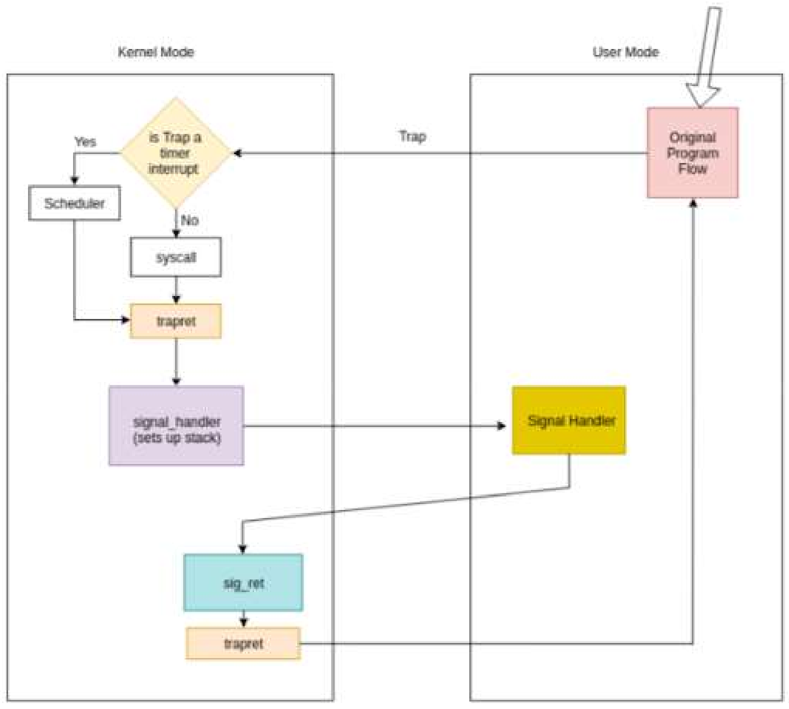

Figure 1: Flow Control of Receiver Processor on Signal Handling

Now, a process should wait if there is a signal available for it. So, it can use sig_pause() to sleep on a certain channel until the signal arrives at it. sig pause checks if there is any signal available for this process if there is not then it gets blocked by calling function sleep on a certain channel. Syscall sig send should wake up the receiver process which might be waiting for a signal.

In this way, two processes in xv6 can communicate with each other.

Fore-ground and back-ground process execution in the shell:

Now, let’s see the basic definition of the foreground and background processes in Linux.

- Foreground Processes: They run on the screen and need input from the user. For example Office Programs
- Background Processes: They run in the background and usually do not need user input. For example Antivirus.

Now, if you are running a foreground process, we cannot access the shell till the foreground process is terminated or suspended.

Now, in Linux, we can add ‘&’ at the end of the command to make it run in the background making the shell accessible for the next commands. These background processes cannot access STD-IN.

Now, let’s implement this feature in xv6.

First, when a command is entered, the shell creates a child using a fork. Now, this child would be responsible for the executing of this process. Now, this child changes its program file into the process using exec() function. Meanwhile, the main shell calls wait() and go into a sleeping state till the process terminates.

Now, to implement a background process we need to stop the shell from going into sleep.

So, Shell forks a background process when given with an '&' at the end of the statement and returns immediately for the next input. Do not wait for background proc.

- Background procs will fork only more background procs
- Background procs will be terminated when they attempt to use STDIN
- Background proc will be reaped by a parent during waitpid, if not background proc will become a zombie.
  If the zombie's parent exits, it makes initproc the parent of all its children. Initproc sits in an infinite waitpid loop, reaping zombies

Now, when it comes to the foreground processes, we will do the same as it did in the original xv6.

- Shell forks a foreground process and waits for the foreground child to exit.
  During this wait, it may reap some zombie background child processes but will continue to wait for the foreground child.
- When a foreground process forks a child, the child will become the new foreground process.
  Only it is allowed to read STDIN. Any other process attempting to read STDIN will be killed.
  Once the foreground child dies its parent gets to be the foreground process.

Now, to implement this, we have to modify shell.c file

First, we will add a condition at the end before it forks, saying if the command has a ‘&’ sign in the end. If it is true, then we will call a function named runbg(), else we will run rung().

Now in runfg(), we will fork() and the child process will call the runcmd() function and gets the process started. And the parent will call wait() and keep waiting till there is a zombie process to be swept by wait() belonging to the shell.

While in runbg(), the shell will fork itself and the child will call runcmd function the same as the runfg() function. But in this, the shell won’t call wait instead it iterate into the next loop. So, the next line appears for a command to be entered.

The implementation is not yet finished.

Now, we will implement two system call just like we did in Ctrl+C section in the beginning of the report. And we will also add proc parameters named fg and bg which represent if a process is a foreground or background process. The two system call we will implement will be called by the shell, and one will set the fg parameter to true of a given PID, and another will set the bg parameter to true for a given PID. In this way, the os could recognize the process which are foreground and background processes. And in the fork, we will also have to implement such that the fg or bg information is passed on.

So, here shell calls these system calls and gives the PID of its child to them. Now, there is one problem that arises with this. That is if the child starts executing the process before the shell calls these system calls, some children of this process may not inherit the true fg and bg value from its parent i.e. shell child/called process. To prevent this, we could create a semaphore(a shared variable), which prevents the child shell from calling runcmd() function before the shell calls the system call.

Now, we have to stop background processes from accessing STDIN.

For this one approach is that we can add a condition in the scanf(), which is the basic function to access STDIN. So in this condition, we will check whether the process calling the function has bg true or not. If it has a true bg value then we will terminate it in the same way we did in Ctrl+C. If scanf() cannot directly access the myproc() function, then we can add a system call scanf(), for all standard output and input which will call this local scanf(), instead of directly calling scanf(). This system call will call a function in proc.c which will check if the process is background or foreground.

Another more feasible implementation is to close the STDIN pipeline for the background process or to its parent shell child. We can use **close(fd[0])** where fd should be initiated with pipe(fd). Now, the process which calls this has no access to read pipe. Similarly, fd[1], fd[2] represent the output and error pipe. So, this way would be a lot less complicated. Although we still have to implement the cases where it accesses the STDIN and the os has to terminate it because closing the pipe to a process doesn't kill it when it tries to access STDIN but it just ignores it. So, we have to follow the previous approach for this but don't have to do all of that.

In this way, we can prevent background processes from accessing STDIN.

Further improvements have to be done, like storing the output of the background process in a buffer and implementing a user program foreground, which will make a background process into a foreground process by calling a system call which changes the proc values and makes the shell wait (a bit complicated than that). The buffer containing the output of the background process can be shown in the terminal when it turns into a foreground process.

Note: *The presentation video of this assignment is just covering some details as the time was set to 10mins, but not the whole work.*

Thank You.