Lecture 07: Process Termination and Inter Process Communication

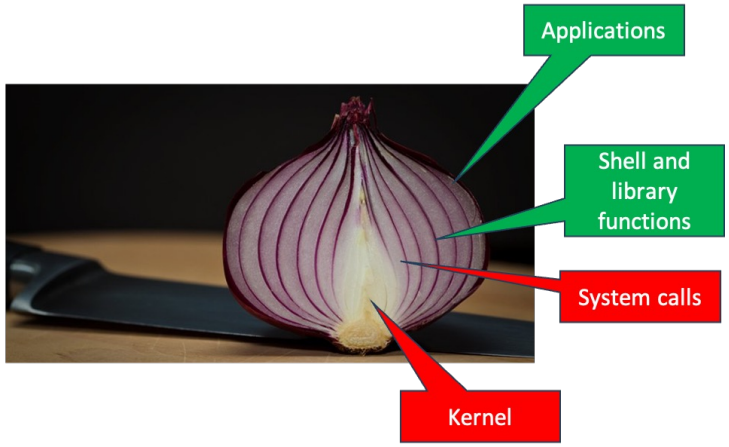
Vivek Kumar

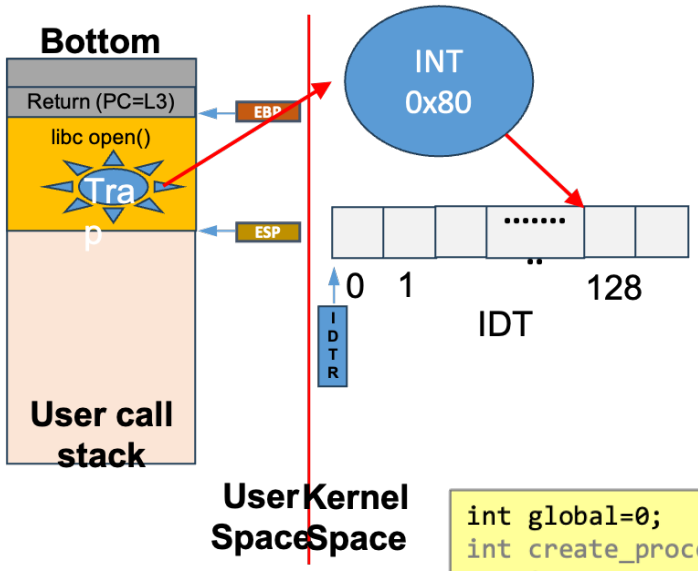
Computer Science and Engineering

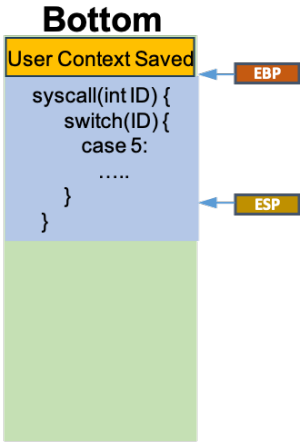
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture







- Protection rings (kernel and User mode) in Unix-like OS
- Interrupts and system call
- Process creation
  - Copy on write when forked

```
int global=0;
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        global++;
    } else {
        printf("I am the parent Shell\n");

    }
    printf("Global value = %d\n",global);
    ....
    return 0;
}
```

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

A Process's Life Lessons (contd.)

1. Processes can have children
2. Children should be obedient to their parent
3. Parent must follow the steps for good parenting
4. Children should not run their family business

The Obedient Child

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        exit(0);
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- **exit** syscall allows to send a specific termination code (exit status) from a child process to the parent upon termination
  - "signal" is sent to the parent (inter-process communication)
  - In case of abnormal termination of child, the exit status is generated and send by the kernel
- exit carries out process cleanup – reclaiming memory, flushes buffers, closing fds, etc.
- But how the parent can get the exit status (next slide)?
  - Remember child is not going to return to the parent just like a callee method returns to a caller method

The Act of Good Parenting

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child (%d)\n",getpid());
    } else {
        int ret;
        int pid = wait(&ret);
        if(WIFEXITED(ret)) {
          printf("%d Exit =%d\n",pid,WEXITSTATUS(ret));
        } else {
          printf("Abnormal termination of %d\n",pid);
        }
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

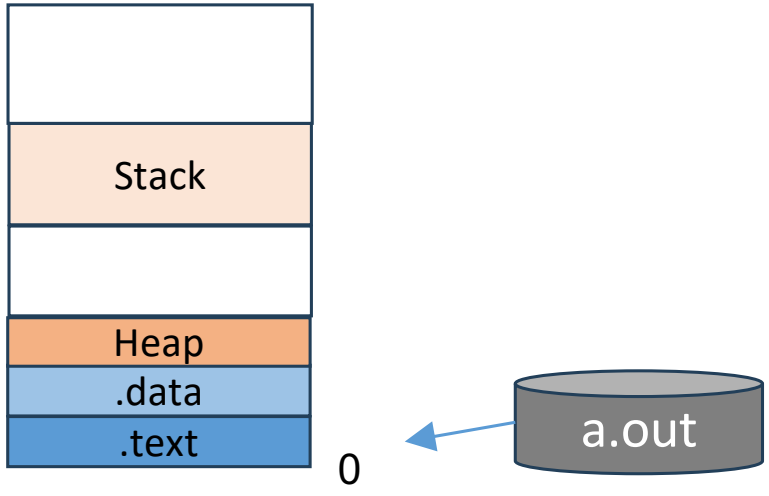
- **wait** and **waitpid** allows the parent process to block until the child process terminates
  - **wait** will block only for the first child, whereas **waitpid** can be used for a specific child
  - Returns the child's PID
  - Used for retrieving exit status from child
- Will there be deterministic execution of printfs from parent and child processes (notice there is no sleep)?
- Good parents avoid making their child as **Zombies or Orphan**
  - Child is zombie when it has terminated but has its exit code remaining in the process table as it is waiting for the parent to read the status
  - **Orphaned** children outliving their parent's lifetime are adopted by the mother-of-all-processes (**init**)

Child Should Not Run Family Business

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- Main goal for creating a child process is to let it live its own free life without depending on its parent
  - The child won't let go off the parent's property (code path) until its forced to call exec
- An exec calls the OS loader internally that loads the ELF file with its command line argument as specified in the argument list
- There are seven different versions of exec which are collectively referred as exec function

exec Behind the Curtain

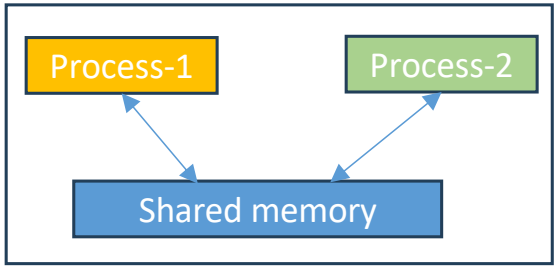


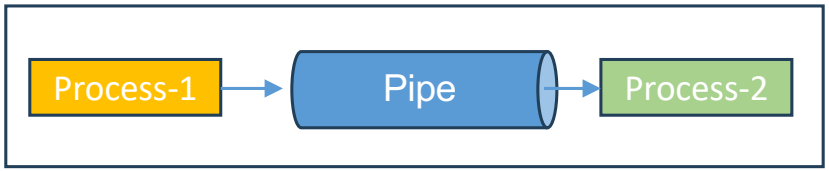
- It's only job is to construct the process's address space
  - Unload current process address space (segments)
  - Read ELF file from the disk
  - Create the user part of the address space lazily
    - E.g., space for .data will be allocated only after some global variable is accessed during program execution from the .text segment
- Note that PID remains the same after the process calls exec

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

Inter-Process Communication





- Recall, forked process calls an exec after which it has its own address space that is not shared with the parent
- IPC mechanisms to share information between processes

Points to Ponder

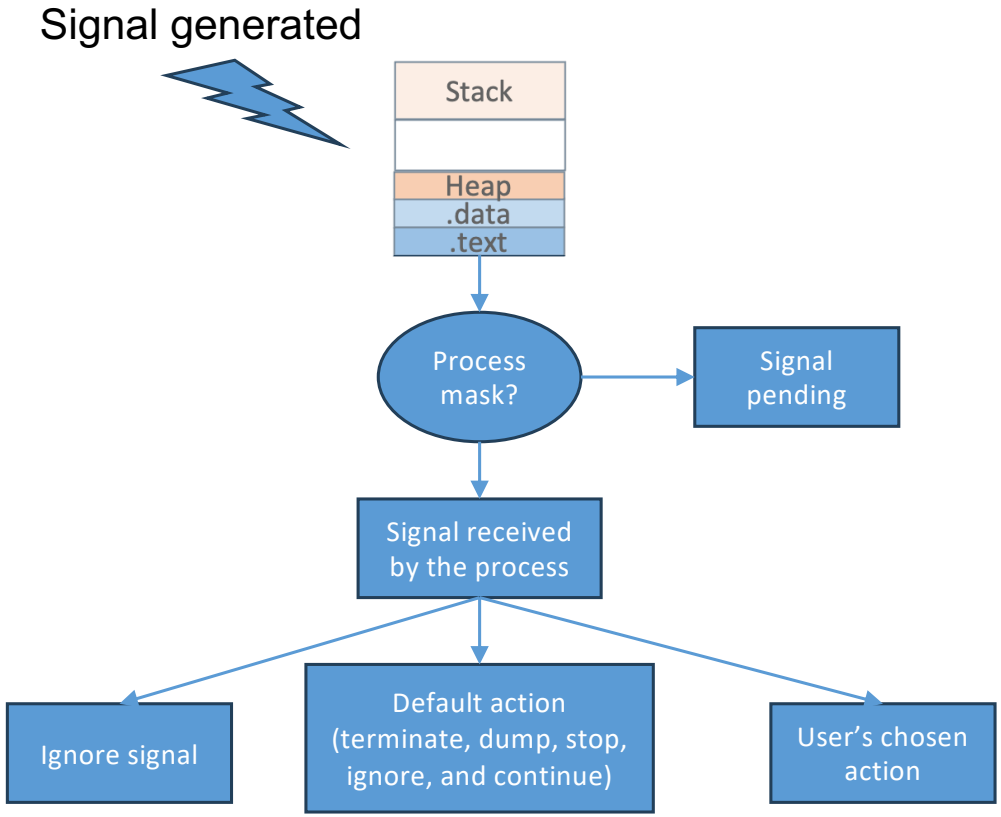
```
vivek@possum:~/os23$ vi infinite-fib.c
vivek@possum:~/os23$ gcc infinite-fib.c
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
^C
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
30
Fib(30) = 832040
Input i:
^Z
[1]+  Stopped                 ./a.out
vivek@possum:~/os23$ fg
./a.out
23
Fib(23) = 28657
Input i:
```

- What happens when we press the following key combination on a shell that is running some program
  - Ctrl-c
    - Process simply terminates even though it was executing some instruction inside the .text segment
  - Ctrl-z
    - Process moves to background (you can get it back to foreground using “fg”)
  - How come these actions are happening even though the program did not have any code to demonstrate the above mentioned behavior
- How does a child process that died inform about its fate to its parent?
- What happens when a program attempts to access an invalid memory address?

Signals – A Limited Form of IPC

- Signals (interrupts) are technique used by the OS to communicate with a process
- Every signal has a name that has a unique number assigned
  - Ctrl-c and Ctrl-z generates hardware interrupt from the keyboard that is handled by the OS by sending SIGINT (2) and SIGSTOP (20) signal, respectively, to the running process
- Each signal type has a default handler
  - A program can install its own handler for signals of any type
    - Except SIGKILL (default action is to exit the process) and SIGSTOP (default action is to suspend the process)
- States
  - Generated – due to some event
  - Delivered – received by the process
  - Blocked – process has blocked the signal
  - Pending – to be delivered
  - Caught – action associated with signal has been taken by the process

Signal Delivery and Handling



- Process can use sigprocmask function to make changes to its signal mask (member in task_struct) if it wants to temporarily block a set of signals (not all signals can be blocked, e.g., SIGKILL and SIGSTOP)
- User can create his own signal handler function (we will see soon)
- Signal name has prefix "SIG". Some of the signals that you can easily come across are:
  - SIGINT (ctrl-c), SIGQUIT (ctrl-\), SIGSTOP (ctrl-z), SIGTRAP (breakpoint), SIGSEV (segmentation fault), SIGTERM (process termination), SIGCHLD (child stopped/terminated), SIGFPE (floating point exception), etc.

Catching the Signal

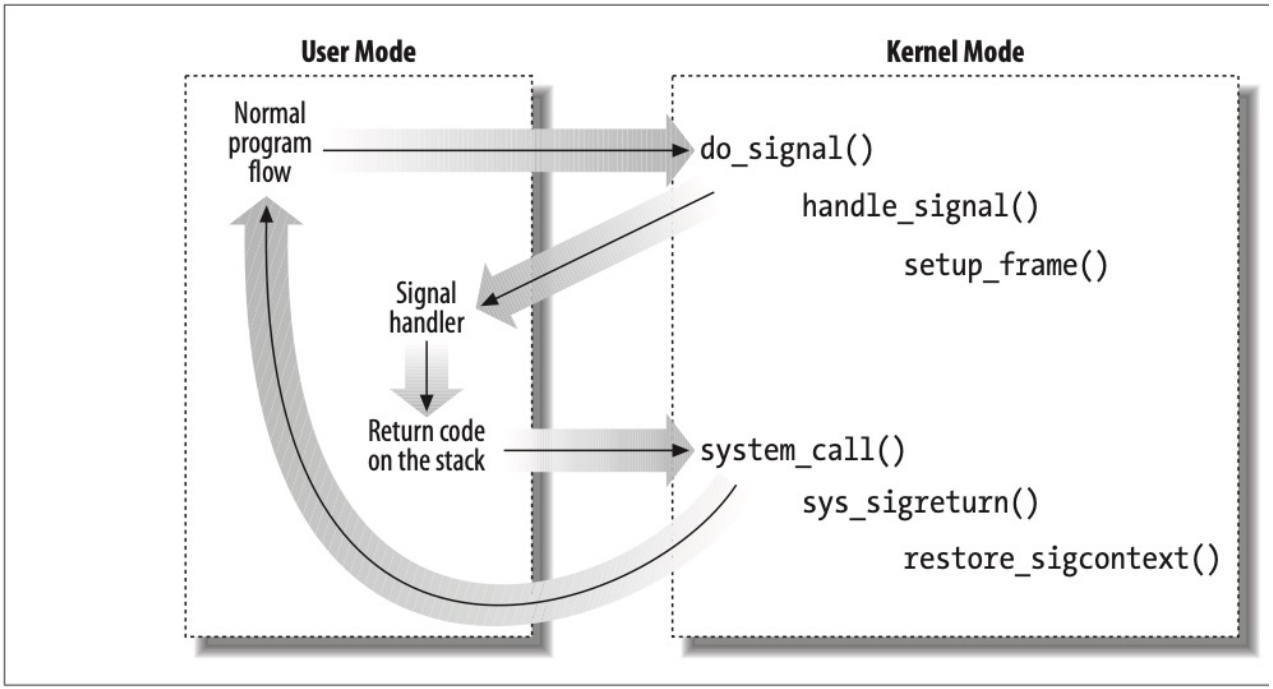


Figure 11-2. Catching a signal

- In the absence of user signal handler, signal is handled in kernel mode, otherwise it is handled in the user mode
- Process switches into kernel mode after receiving a non-blocking signal
- Kernel handle the signal by first setting up stack frame on user stack (save context)
- Control is passed to user stack and user signal handler is executed
- Control is returned back into kernel mode which restores the user stack to its original state to resume the execution of the program
  - **sigreturn** system call used to restore process state

Figure source: from the book "Understanding the Linux Kernel"

Programming Signals

```
static void my_handler(int signum) {
  static int counter = 0;
  if(signum == SIGINT) {
    char buff1[23] = "\nCaught SIGINT signal\n";
    write(STDOUT_FILENO, buff1, 23);
    if(counter++=1) {
      char buff2[20] = "Cannot handle more\n";
      write(STDOUT_FILENO, buff2, 20);
      exit(0);
    }
  } else if (signum == SIGCHLD) {
    char buff1[23] = "Caught SIGCHLD signal\n";
    write(STDOUT_FILENO, buff1, 23);
  }
}
```

```
int main() {
  struct sigaction sig;
  memset(&sig, 0, sizeof(sig));
  sig.sa_handler = my_handler;
  sigaction(SIGINT, &sig, NULL);
  sigaction(SIGCHLD, &sig, NULL);
  int n;
  while(1) {
    printf("Input i: \n");
    scanf("%d",&n);
    if(fork()==0) {
      printf("Fib(%d) = %d\n",n,fib(n));
      exit(0);
    }
    else wait(NULL);
  }
  return 0;
}
```

Why Fib(20) is calculated twice?

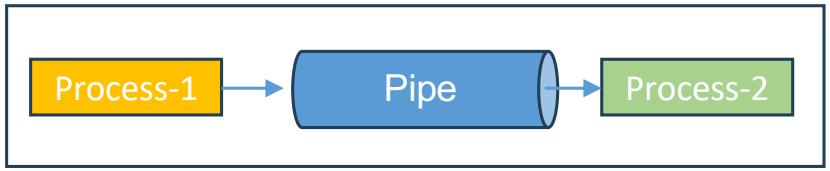
```
vivek@possum:~/os23$ ./a.out
Input i:
10
Fib(10) = 55
Caught SIGCHLD signal
Input i:
20
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Cannot handle more
```

- sigaction is used to change the default action associated with a signal
  - Except SIGKILL and SIGSTOP
- User can assign his own signal handler method that would get invoked when the assigned signal is received by the process
- Only asynchronous safe functions should be called inside signal handler
  https://man7.org/linux/man-pages/man7/signal-safety.7.html

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

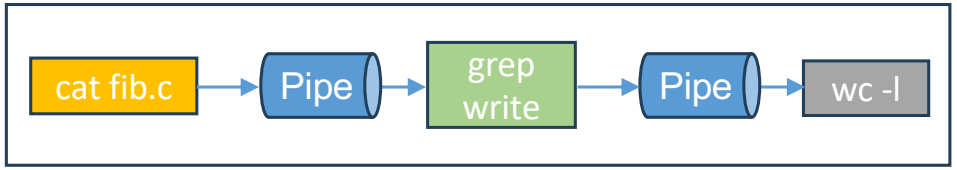
IPC Using Pipes



Processes **must** be running on the **SAME** machine

- Pipes (analogous to water pipe) is a unidirectional stream of data flowing from a source process to a destination process
  - Kernel buffer exposed to processes as a pair of file descriptors (readable end and writable end). Default buffer size is 16 pages (16x4096 bytes)
  - Data delivered in the same order as sent
  - Uses blocking IO and the writer process will block if the pipe is full
- We use it frequently on Shell
  - Better than using temporary files as pipes automatically clean up themselves unlike files that must be explicitly removed using the “rm” command on Shell

```
vivek@possum:~/os23$ cat fib.c | wc -c
1788
vivek@possum:~/os23$ cat fib.c | wc -l
77
vivek@possum:~/os23$ cat fib.c | grep write | wc -l
10
```



Programming With pipe

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        char buff[] = "Hello my dear good Parent";
        sleep(2);
        write(fd[1], buff, sizeof(buff));
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- pipe expects an int array of size two only for a readable and writable file descriptor
- Reader and writer processes must close the pipe end they are not going to use
  - It is very important as fork duplicates the open file descriptors in the parent process
- What would happen if the child sleeps before writing to the pipe, as the parent has already issued the read?

Programming With pipe and dup

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- dup2 can be used to duplicate a file descriptor
  - E.g., duplicate one of the end of the pipe as STDOUT or STDIN
    - Duplicating to STDOUT will cause printf to print to the pipe instead of the STDOUT
- Used by the Shell when we pipe the output of one command to another command

Next Lecture

- Inter-process communication (contd.)