Unix System Calls

PART-1

Process Management Model

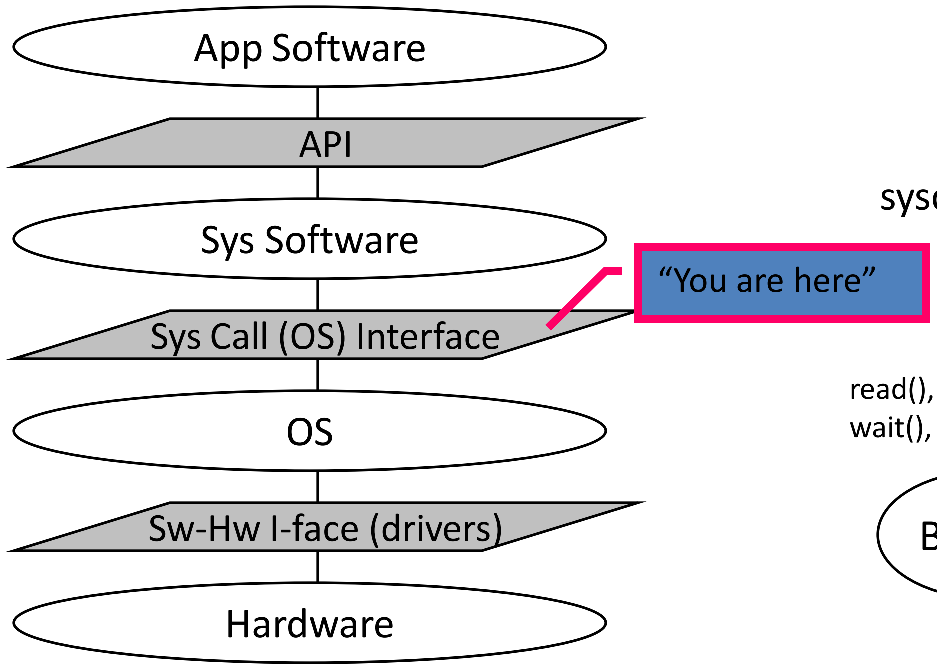
The Unix process management model is split into two distinct operations：

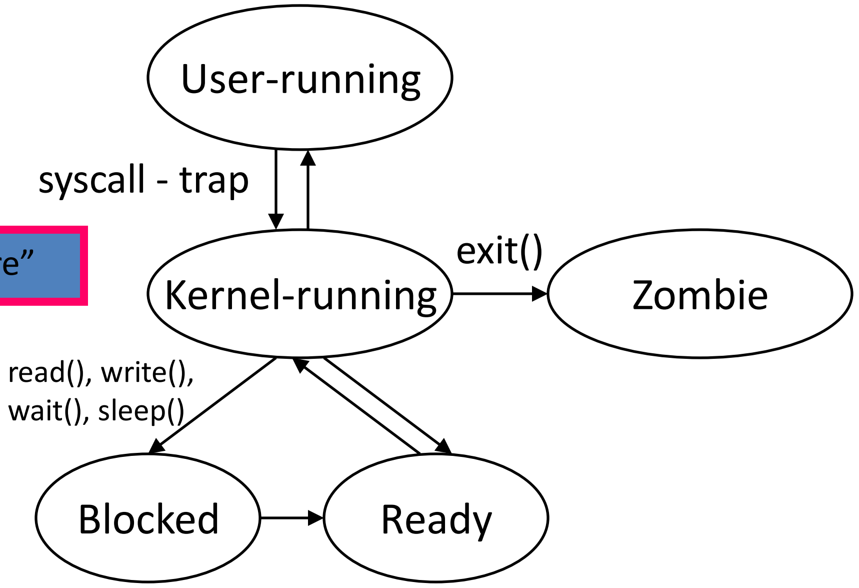
1. The creation of a process.
2. The running of a new program.

Overview

- System Call Interface
- Process Management with C
  - `fork()`
  - `exec()`
  - `wait()`
  - `exit()`

System Call Interface





The `fork()` System Call (1)

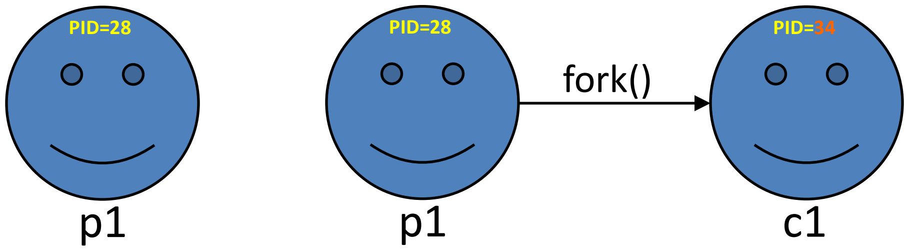
- A process calling `fork()` spawns a child process.
- The child is almost an identical *clone* of the parent:
  - – Program Text (segment .text)
  - – Stack (ss)
  - – PCB (eg. registers)
  - – Data (segment .data)

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

f you only need the OS types, say for the prototypes of your functions, then just `#include <sys/types.h>`. However if you need the function definitions, then you `#include <unistd.h>` or any of the other system headers, as needed.

The `fork()` System Call (2)

- The `fork()` is one of the those system calls, which is called once, but returns twice!
- After `fork()` both the parent and the child are executing the same program.
- On error, `fork()` returns `-1`



Consider a piece of program

```
...
pid_t pid = fork();
printf("PID: %d\n", pid);
...
```

The parent will print:

```
PID: 34
```

And the child will **always** print:

```
PID: 0
```

The **pid_t** data type represents process IDs. You can get the process ID of a process by calling getpid . The function getppid returns the process ID of the parent of the current process (this is also known as the parent process ID)

The following is a simple example of fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
  printf("Hello \n");
  fork();
  printf("bye\n");
  return 0;
}
```

Hello –is printed once by parent process

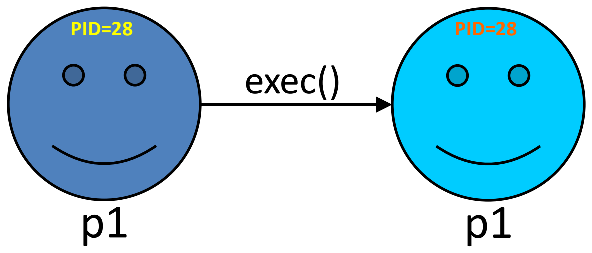
bye –is printed twice, once by the parent and once by the child

The `exec()` System Call (1)

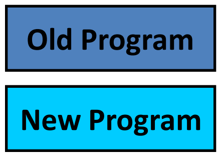
- The `exec()` call replaces a current process' image with a new one (i.e. loads a new program within current process).
- The new image is either regular executable binary file or a shell script.
- There's no a syscall under the name `exec()`. By `exec()` we usually refer to a family of calls:
  - `int execl(char *path, char *arg, ...);`
  - `int execv(char *path, char *argv[]);`
  - `int execle(char *path, char *arg, ..., char *envp[]);`
  - `int execve(char *path, char *argv[], char *envp[]);`
  - `int execlp(char *file, char *arg, ...);`
  - `int execvp(char *file, char *argv[]);`
- Here's what *l*, *v*, *e*, and *p* mean:
  - ***l*** means an argument list,
  - ***v*** means an argument vector,
  - ***e*** means an environment vector, and
  - ***p*** means a search path.

The `exec()` System Call (2)

- Upon success, `exec()` **never** returns to the caller. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via `exec()` appear in the `argv[]` of the `main()` function.



Legend:

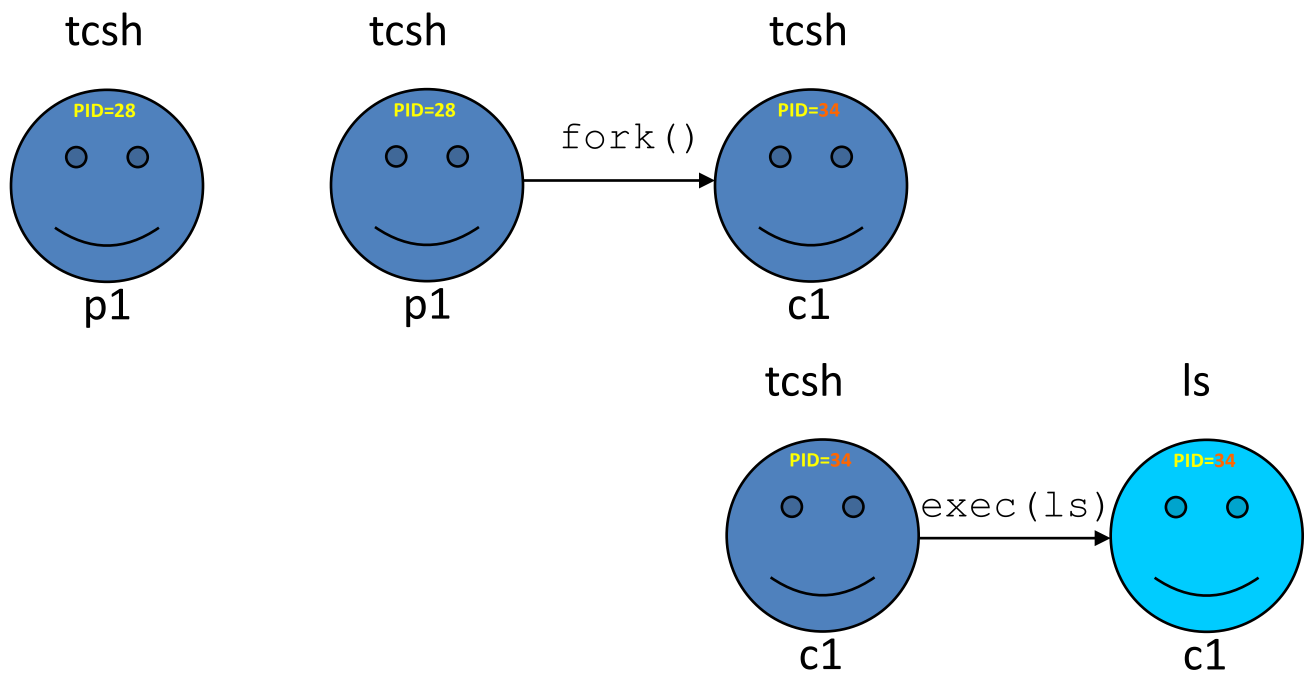


Environment

- The *e*-exec calls use the environment when attempt to invoke a new program.
- Name = Value
  - HOME
  - PATH
  - SHELL
  - USER
  - LOGNAME
  - …
- `set` or `env` - will display current environment, which you can modify with:
  - the `export` command in a shell or a shell script (`bash`);
  - the `setenv` for `tcsh`
  - the `getenv()`, `setenv()`, `putenv()`, etc. in C

`fork()` and `exec()` Combined

- Often after doing `fork()` we want to load a new program into the child. *E.g.*: a shell.



The System `wait()` Call

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate* is more appropriate terminology, but a bit less common).

The System `wait()` Call (2)

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The `wait()` causes the parent to wait for any child process.
- The `waitpid()` waits for the child with specific PID.
- The status, if not NULL, stores exit information of the child, which can be analyzed by the parent using the `W*()` macros.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

The `exit()` System Call

```
#include <stdlib.h>

void exit(int status);
```

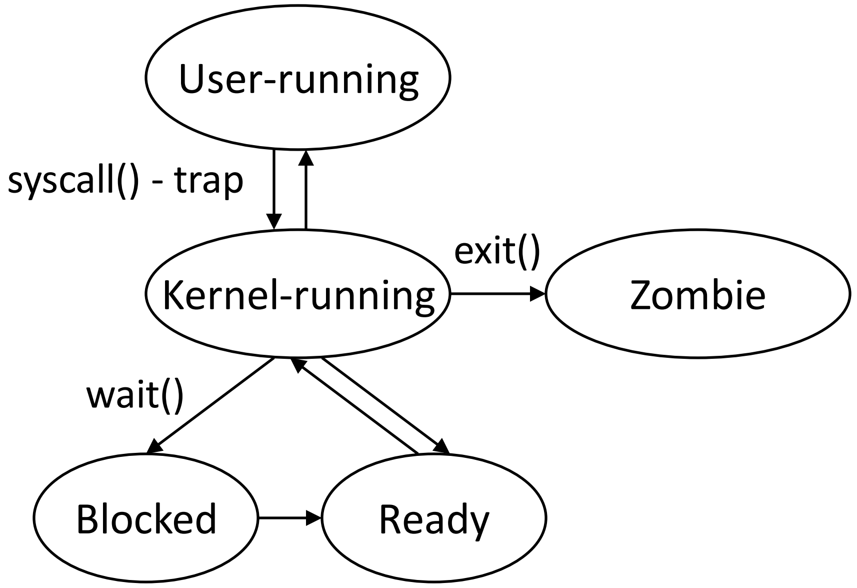
- This call gracefully terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the zombie state.
- By calling `wait()`, the parent cleans up all its zombie children.
- `exit()` specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.
- `_exit()` call is another possibility of quick death without cleanup.

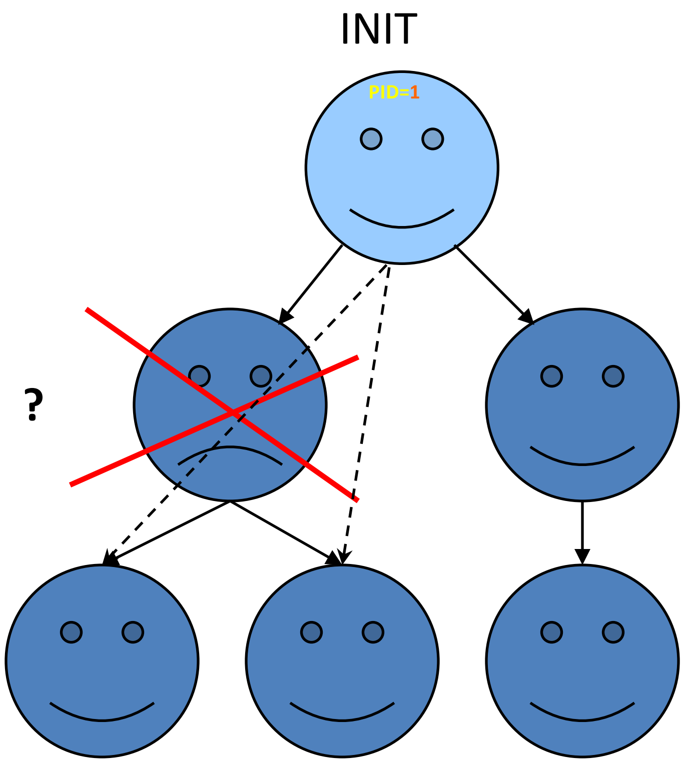
_exit()

- The exit operation typically performs clean-up operations within the process space before returning control back to the operating system. Some systems and programming languages allow user subroutines to be registered so that they are invoked at program termination before the process actually terminates for good. As the final step of termination, a primitive system exit call is invoked, informing the operating system that the process has terminated and allows it to reclaim the resources used by the process.

- It is sometimes possible to bypass the usual cleanup; C99 offers the _exit() function which terminates the current process without any extra program clean-up. This may be used, for example, in a fork-exec routine when the exec call fails to replace the child process.

Process Overview





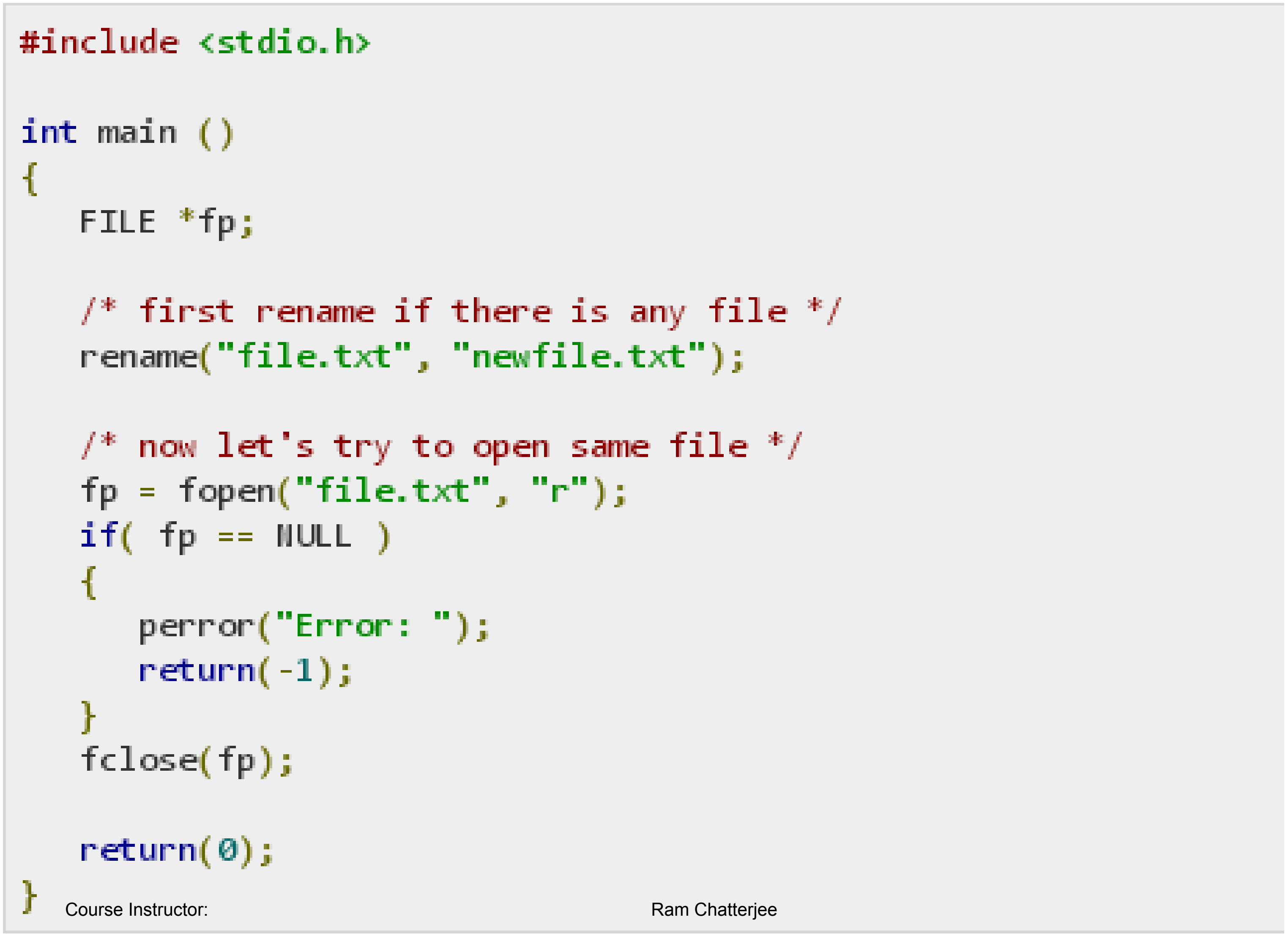
```
/*****************************************************************
 *
 *     Example: to demonstrate fork() and execl() and system calls
 *
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char *argv[], char *env[] )
{
   pid_t my_pid, parent_pid, child_pid;
   int status;

/* get and print my pid and my parent's pid. */

   my_pid = getpid();    parent_pid = getppid();
   printf("\n Parent: my pid is %d\n\n", my_pid);
   printf("Parent: my parent's pid is %d\n\n", parent_pid);

/* print error message if fork() fails */
   if((child_pid = fork()) < 0 )
   {
      perror("fork failure");
      exit(1);
   }
/* fork() == 0 for child process */

   if(child_pid == 0)
   {  printf("\nChild: I am a new-born process!\n\n");
      my_pid = getpid();     parent_pid = getppid();
      printf("Child: my pid is: %d\n\n", my_pid);
      printf("Child: my parent's pid is: %d\n\n", parent_pid);
      printf("Child: I will sleep 3 seconds and then execute - date - command \n\n");

      sleep(3);
      printf("Child: Now, I woke up and am executing date command \n\n");
      execl("/bin/date", "date", 0, 0);
      perror("execl() failure!\n\n");

      printf("This print is after execl() and should not have been executed if execl were successful! \n\n");

      _exit(1);
   }
/*
 * parent process
 */
   else
   {
      printf("\nParent: I created a child process.\n\n");
      printf("Parent: my child's pid is: %d\n\n", child_pid);
      system("ps -acefl | grep ercal");  printf("\n \n");
      wait(&status); /* can use wait(NULL) since exit status
                        from child is not used. */
      printf("\n Parent: my child is dead. I am going to leave.\n \n ");
   }

   return 0;
}
```

C library function - perror()

- The C library function **void perror(const char *str)** prints a descriptive error message to stderr. First the string **str** is printed, followed by a colon then a space.
- Declaration
  - Following is the declaration for perror() function.
  - void perror(const char *str)Parameters
- **str** -- This is the C string containing a custom message to be printed before the error message itself.
  - Return Value
  - This function does not return any value.

Example

The following example shows the usage of perror( ) function.



```c
#include <stdio.h>

int main ()
{
   FILE *fp;

   /* first rename if there is any file */
   rename("file.txt", "newfile.txt");

   /* now let's try to open same file */
   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);

   return(0);
}
```