Video Lecture # 19
Process Management
Part - III

Course: SYSTEM PROGRAMMING

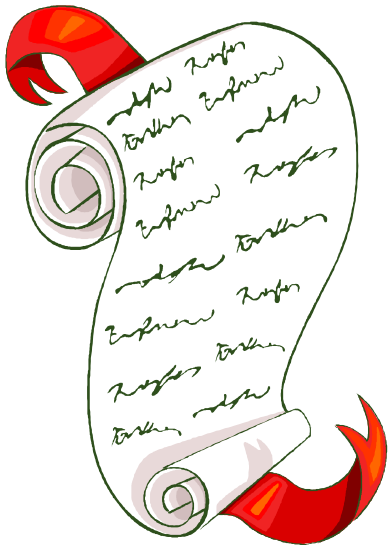
Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Agenda

- Overwriting process address space using `exec()`
- Effect of fork and exec on process attributes
- Writing your own `system()` library function
- Job Control
- Process groups
- Process sessions
- Terminals



Program Execution
`exec()` Family

exec() Functions

- A process may overwrite itself with another executable image. When a process calls one of the six **exec()** functions, it is completely replaced by the new program, and the new program starts executing its `main` function
- There are five library functions of `exec` family and all are layered on top of the **execve()** system call. Each of these functions provides a different interface to the same functionality
- There is no return after a successful `exec` call. The `exec()` functions return only if an error has occurred. The return value is -1, and `errno` is set to indicate the error

exec() Functions (cont...)

```
int execl (const char *pathname, const char* arg0,...,(char*)0);
int execlp (const char *filename, const char* arg0,...,(char*)0);
int execle (const char* pathname, const char* arg0,...,(char*)0,
                    char* const envp[]);
```

- The first argument to this family of `exec()` calls, is the name of the executable, which on success will overwrite the address space of the calling process with a new program from the secondary storage
- The **l** after the `exec` means that command line arguments to the new program will be passed as a comma separated list of strings with a `'\0'` character at the end
- The **p** stands for path. It means that the program specified as the first argument should be searched in all directories listed in the PATH variable. However, using absolute path to program is more secure than relying on PATH variable, which can be more easily altered by malicious users
- The **e** stands for environment. It means that after the command line arguments, the program should pass an array of pointers to null terminated strings, specifying the new environment of the program to be executed. Otherwise, the caller environment will be used

exec() Functions (cont...)

```
int execv (const char *pathname, char *const argv[]);
int execvp (const char *filename, char* const argv[]);
int execve (const char* pathname,char* const argv[],
                char* const envp[]);
```

- The first argument to this family of `exec()` calls, is the name of the executable, which on success will overwrite the address space of the calling program with a new program
- The **v** after the `exec` means that command line arguments to these functions will be passed as an array of pointers to null terminated strings
- The **p** stands for path. It means that the program specified as the first argument should be searched in all directories listed in the PATH variable. However, using absolute path to program is more secure than relying on PATH variable, which can be more easily altered by malicious users
- The **e** stands for environment. It means that after the command line arguments, the program should pass an array of pointers to null terminated strings, specifying the new environment of the program to be executed. Otherwise, the caller environment will be used

exec Functions (cont...)

- All successful **`exec()`** functions never return. In case it returns, it always return -1 , but we need not to compare this value. The fact that it returned informs us that an error occurred. We can use **`errno`** to determine the cause of failure

- **Reasons of failure can be:**

EACCES	The specified program is not a regular file, or doesn't have execute permissions enabled or one of the directory components of pathname is not search able
ENOENT	The specified program does not exist
ENOEXEC	The specified program is not in a recognizable executable format
ETXTBSY	The specified program is open for writing by another process
E2BIG	The total space required by the argument list & environment list exceeds the allowed maximum

Use of `exec()` Functions
Proof of concept
`exec1.c` to `exec4.c`

Process Attributes Inherited/Preserved after `fork()` / `exec()`

Attributes Inherited after `fork()` & `exec()`

Process IDs	`fork()`	`exec()`	Description
PID	No	Preserved	
PPID	No	Preserved	
PGID	Inherited	Preserved	
SID	Inherited	Preserved	
Real IDs	Inherited	Preserved	
Effective and Saved SUIDs	Inherited	Preserved	Can be changed
Supplementary Group IDs	Inherited	Preserved	

Process Address Space	`fork()`	`exec()`	Description
Text Segment	Shared	No	
Stack Segment	Inherited	No	
Data and Heap Segment	Inherited	No	
Environment Variables	Inherited	--	Depends on type of exec call
Memory Mappings	Inherited	No	
Memory Locks	No	No	

Attributes Inherited after `fork()` & `exec()`

Files and Directories	`fork()`	`exec()`	Description
PPFDT	Inherited	Preserved	PPFDT is inherited after exec unless close-on-exec flag is set
Close-on-exec Flag	Inherited	Preserved	
File offsets	Shared	Preserved	
Open file status flags	Shared	Preserved	
Directory streams	Inherited	No	
Present working directory	Inherited	Preserved	
File mode creation mask	Inherited	Preserved	

Scheduling, Resources	`fork()`	`exec()`	Description
Nice value	Inherited	Preserved	
Priority	Inherited	Preserved	
Scheduling policy	Inherited	Preserved	
Resource limits	Inherited	Preserved	
Resource usage	No	Preserved	
CPU times	No	Preserved	
Exit Handlers	Inherited	No	

Process Attributes Inheritance Proof of concept

`exit_fork.c` & `exit_exec.c`

Executing a shell command using system()

```
int system(const char* command);
```

- It executes a command specified in **cmd** by calling **/bin/bash -c** command and returns after the command has been completed
- Return -1 on error and the return status of the cmd other wise
- Main cost of **system()** is inefficiency. Executing a command using `system()` requires the creation of at least two processes
  - One for the shell
  - Other for the command(s) it executes

Executing shell command using `system()`

Proof of concept

`system1.c` to `system3.c`

Implementing `system()` using `exec()`

- The **`-c`** option to `/bin/bash` command provides an easy way to execute a string containing arbitrary shell command:

```
$ /bin/bash -c "ls"
```

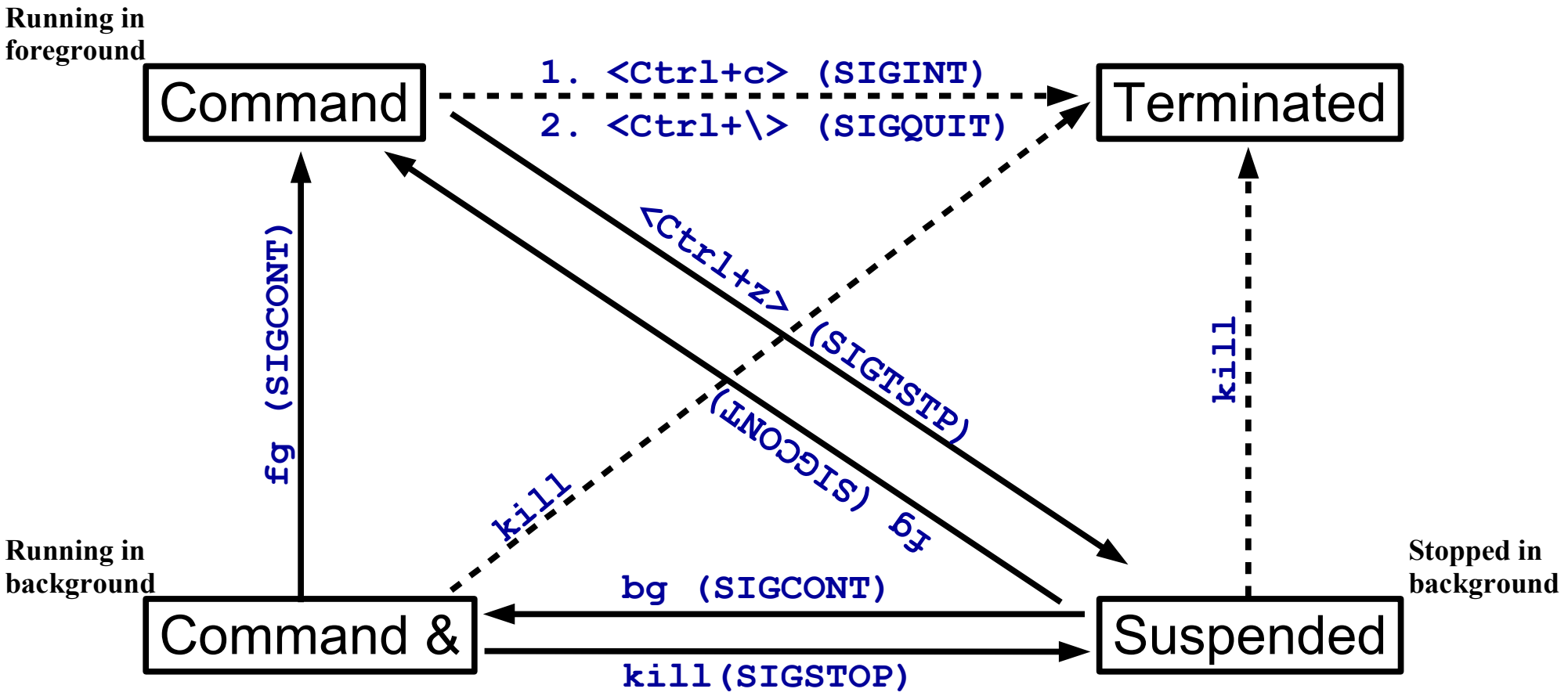
- If there are arguments after the string, they are assigned to the positional parameters, starting with `$0`
- Thus to implement `system()`, we need to use `fork()` to create a child that does an `execl()` to the `bash` program

```
execl("/bin/bash", "mybash", "-c", command, '\0');
```

Writing your own `system()`
Proof of concept
`system4.c`

Process Groups, Sessions and Controlling Terminals

Illustration of Job Control States



Source: The Linux Programming Interface

Process Group, Session & Terminal

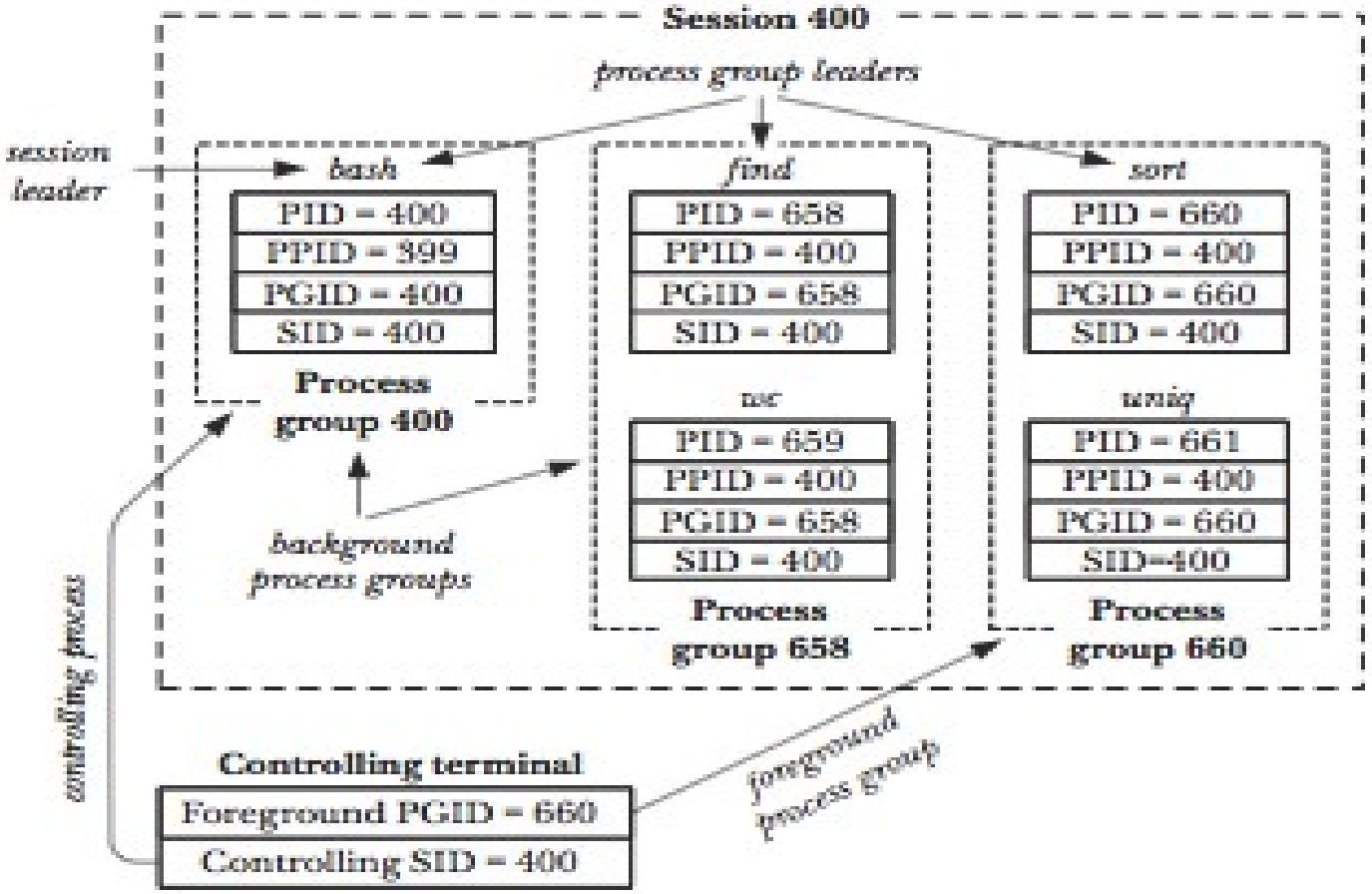
Process Group: Process group is a set of one or more processes sharing the same PGID. Every process group has a Process Group Leader, which is the process that creates the group and whose PID becomes the PGID of that group. A child process inherits its parent's PGID. Life time of a Process Group starts when the leader creates the group and ends when the last member process leave the group

Session: A session is a collection of one or more process groups. A process's session membership is determined by its SID. Every session has a session leader, which is the process that creates a new session and whose PID becomes the SID. At any point in time, one of the process groups in a session is the foreground process group & others are background process groups

Terminal: All processes in a session shares a single controlling terminal, which is established when a session leader first opens a terminal device. A session leader is the controlling process for the terminal. If a terminal disconnect occurs , the kernel sends the session leader SIGHUP signal

Relationships between PGs, Session, CT

```
echo $$
400
find / 2> /dev/null | wc -l &
[1] 659
sort < longlist | uniq -c
```



Source: The Linux Programming Interface

Retrieving and Changing Process Group

```
pid_t getpgid(pid_t pid);
int setpgid(pid_t pid, pid_t pgid);
```

- `getpgid()` returns the process group ID of the process specified by pid. If `pid` is zero, the process ID of the current process is used
- `setpgid()` sets the PGID of the process specified by `pid` to `pgid`. If 1st argument `pid` is zero, the PID of the calling process is used. If 2nd argument `pgid` is zero, the PID of the process specified by `pid` is used. If both arguments are zero, then the calling process is made the group leader
- If `setpgid()` is used to move a process from one process group to another, both process groups must be part of the same session. In this case, the `pgid` specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process

Retrieving and Changing Session

```
pid_t getsid(pid_t pid);
pid_t setsid();
```

- `getsid()` returns the session ID of the process, specified by `pid`. If `pid` is zero, the session ID of the calling process is returned
- `setsid()` creates a new session if the calling process is not a process group leader. The calling process is made the leader of the new session (i.e., its SID is made the same as its PID). The calling process also becomes the process group leader of a new process group in the session (i.e., its PGID is made the same as its PID). The calling process will be the only process in the new process group and in the new session

Things To Do

If you have problems visit me in counseling hours. . . .