# Process Management

On Linux

Prayas Mohanty (Red Hat Certified Instructor)

Red Hat Certification ID: 100-005-594

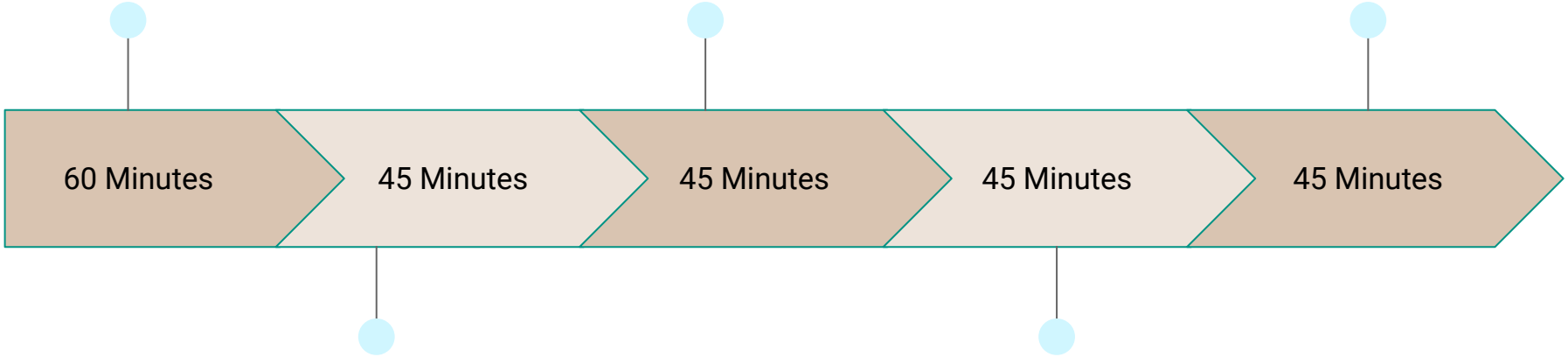
# Objective

- What is the concepts of process management.
  - Process creation
  - Parent – Child Relation in Process
  - Process Termination & the Role of  wait.
  - What are zombie and orphan process.
- What Linux system calls – fork(), wait()…. do
- How to write programs using system call to:
  - create processes
  - extract exit codes using macros
  - handle zombie processes etc.
- What are process specific coding guidelines.

Securing Apache using SSL module

Optimizing Apache using MPM Modules

Automate Web Server Deployment using Ansible.




Apache Security and SELINUX

Apps (tomcat/wsgi/cgi) Integration with Apache.

# Prerequisite of Participants

- Having Knowledge on Linux Platform
- Having familiarity with Linux Command line
- Basic Knowledge on vi Editor
- Proper Knowledge on C programming
- Basic Knowledge on Processes

# What is Process Management

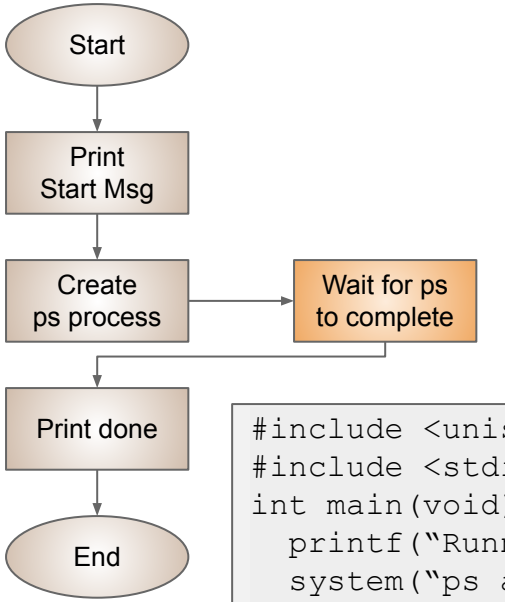
- Process management involves various tasks like creation, scheduling, termination of processes.
-

# Process Creation

- Everytime you execute a command, Linux create a process for your command.
  - Commands are executables stored in a directory specified by PATH variable.
  - Executing a command loads the instructions from the executable file to the text part of the address space after creating the necessary data structure for a Process with a PID.
- You can cause a program to run from inside another program and thereby create a new process by using the system library function.

```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  printf("Running ps with system\n");
  system("ps ax");
  printf("Done.\n");
  exit(0);
}
```

# How that program works



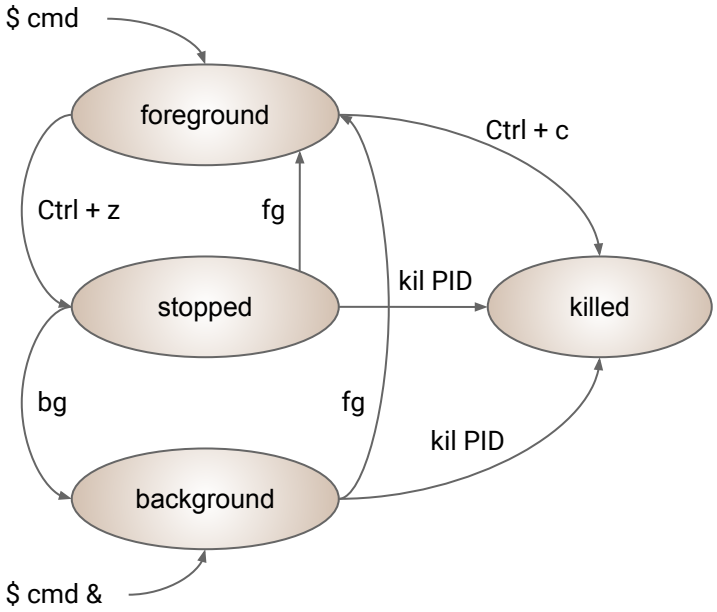

```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  printf("Running ps with system\n");
  system("ps ax");
  printf("Done.\n");
  exit(0);
}
```

- The program calls system function with the string “ps ax” , which executes the ps program.
  - In this case the running instance of the program calling the system function is call the **Parent** process.
  - The running instance of the ps program is called the **Child** process.
- The program returns from the call to system when the ps command has finished.
  - As a default, Parents wait for Child process to gracefully complete to start execution of it’s next instruction(wait to ready).

# How to Control the Process

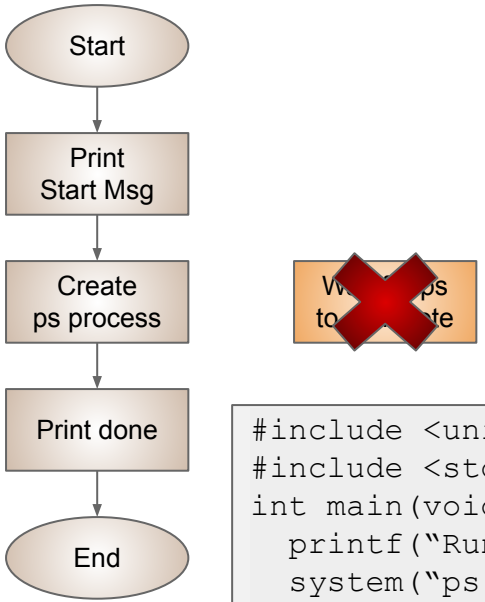
- When you run a command / create a process, Your process may need some input and produce some output / errors.
- By default, all the inputs are fetched from the user through keyboard and the output, and error are written to the Monitor.
- This combination of Monitor and keyboard are technically referred as terminal (tty) which is sometime physical and most often virtual.
- All these processes which are being attached to a terminal can also be called jobs for that terminal and mostly running in foreground.
- The foreground jobs are attached to the terminal session’s lifecycle.
    - If the terminal is closed, the respective job gets terminated.
- Most of the server programs do not tolerate this terminal dependency, so they prefer to run on background.

# Create a process to run in background




- The job in the background continues to run irrespective of the terminal from which it is being initiated.
- You either run a job in the background or move an existing running foreground job in the to the background.
- All the inputs and outputs are supposed to be read and written to a file.
    - if you put a job in the background which prints back to the terminal you might still get to see the output of the job in the terminal.
- This kind of background job can be terminated using kill command & process id.

# Create a process to run in background




```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  printf("Running ps with system\n");
  system("ps ax &");
  printf("Done.\n");
  exit(0);
}
```

- The program call to system returns as soon as the ps program is started as it was to start ps command in background.
- The program then prints Done, and exits before the ps command has had a chance to finish all of its output.
- This kind of process behavior can be quite confusing for users.
- To make good use of processes, you need finer control over their actions.

# Food for thought

Write program to start 10 background count program for 10 minutes and show the result?

# Replacing the Process Image

- In general, using system is a far from ideal way to start other processes.
  - A much better way of invoking programs is to use system call instated.
- You can use exec functions to “hand off” execution of your program to another.
- An exec function replaces the current process with a new process specified by the path & file argument.
- There is a whole family of related functions grouped under the exec heading.
  - int execl(const char *path, const char *arg0, ..., (char *)0);
  - int execlp(const char *file, const char *arg0, ..., (char *)0);
  - int execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);
  - int execv(const char *path, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - int execve(const char *path, char *const argv[], char *const envp[]);

# Exec Function Family

- Execl Family
  - Take all argument Independently
- Execv Family
  - Take all argument as a single array of string
- Both Family
  - P take the path parameter
  - E need to define own (Customize) environment variable.

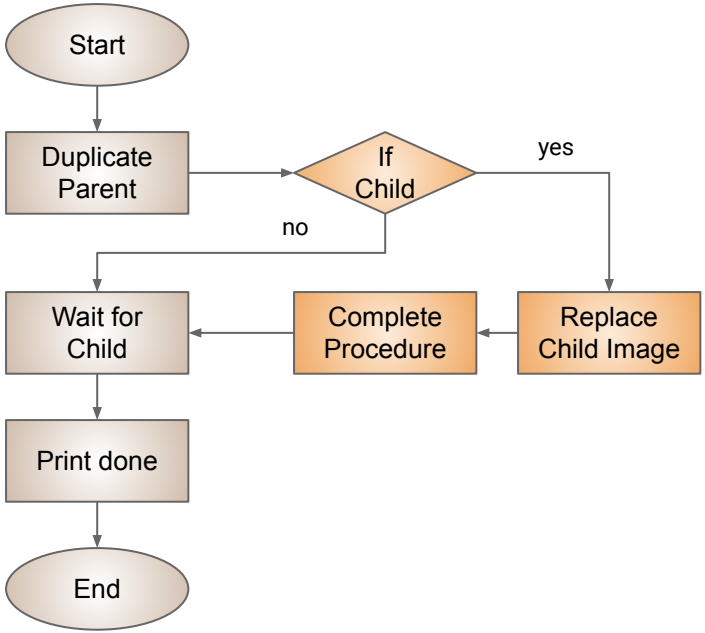
```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  printf(“Running ps with exelp\n”);
  execlp(“ps”, “ps”, “ax”, 0);
  printf(“Done.\n”);
  exit(0);
}
```

```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  char *const ps_argv[] = {"ps", "ax",  0};
  char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console",
0};
  printf("Running ps with execlve\n");
  execve("/bin/ps", ps_argv, ps_envp);
  printf("Done.\n");
  exit(0);
}
```

# Two major difference between system & exec

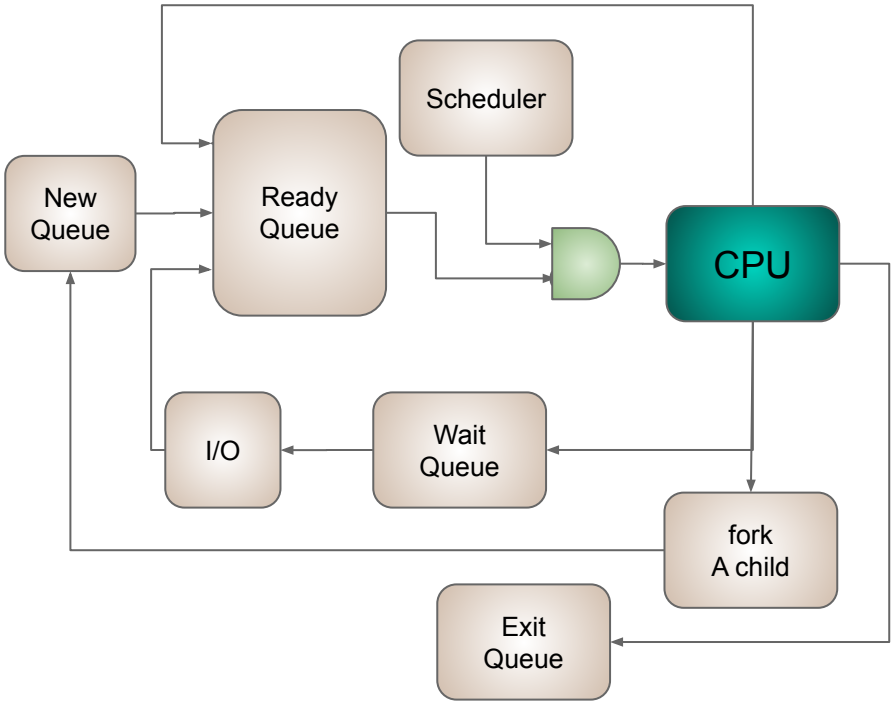
- **There is no reference to a current process in the ps output.**
  - The program prints its first message and then calls exec function family , which replace the current process image with  ps and start execute new instruction from start, which is now ps.
  - The PID of the new process is the same as the original, along with the parent PID and nice value.
  - In effect, all that has happened is that the running program has started to execute new code from a new executable file specified in the call to exec .
- **When you run this program, you get the usual ps output, but no Done.**
  - Because of the image replacement, when ps finishes, it return back to it's parent (shell) that provide you the shell prompt.
  - As the control doesn't return to the original program , so the second message doesn't get printed.
- This kind of process behavior can be quite confusing for users.

# Best Practice for multiprocessing application




- Best Practice for multiprocessing application is to create a new child process for all new task.
- The easiest way to create a child process is to use a system call fork.
  - fork duplicate the parent image into a new process treated as the child process.
- In child process use the exec family system call to replace the child image to parent image.
- As parent wait for child to complete and then start executing the remaining procedure.

# Creating Process with System Call (fork)




- This program runs as two processes creating a new process as Duplicate of parent.
  - The process calling fork() is called the parent.
  - The new process is called the child of the parent.
- Fork is usually a system call, implemented in the kernel.
- Fork is the primary (and historically, only) method of process creation on Unix-like operating systems.

```
#include<stdio.h>
#include<unistd.h>
int main()
{
  pid_t newpid;
  newpid = fork();
  printf("Hello, I have PID %d\n",(int) getpid());
  return 0;
}
```

# What about concurrency

```
include<unistd.h>
#include<stdio.h>
int main() {
  pid t pid; char *message; int n;
  puts("fork program starting\n");
  pid = fork();
  switch(pid) {
    case 0:
      message = "This is the child";
      n = 5;
      break;
    default:
      message = "This is the parent";
      n = 3;
      break;
  }
  for(; n > 0; n--) {
      puts(message);
      sleep(1);
  }
  return 0;
}
```

- The child and parent processes are executed concurrently.

# The wait system call

```
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
int main() {
  pid_t pid,childpid; char *message; int n,status;
  puts("fork program starting\n");
  pid = fork();
  switch(pid) {
    case 0:
      message = "This is the child";
      n = 5;
      break;
    default:
      message = "This is the parent";
      n = 3;
      childpid = wait(&status);
      break;
    }
  for(; n > 0; n--) {
      puts(message);
      sleep(1);
  }
  return 0;
}
```

- A call to wait() blocks the calling process until one of its child processes exits or a signal is received.
  - You must include <sys/wait.h>
  - It return the PID of the terminated child.
  - The status of the child can be captured as an integer by passing a pointer to the wait call.
- If you want to wait for a specific child use waitpid() instated.
  - Like wait it also return the PID of terminated child.
  - Along with status it take 2 more argument as option and the PID

# The exit system call

- When a process terminates it executes an exit() system call, either directly, or indirectly via library code.
- This leaves an exit status value (typically an integer) in the PCB of the terminated process for the parent process to read later.
- You must include <stdlib.h>
-

# Zombie and orphan process

```
#include<unistd.h>
#include<stdio.h>
int main() {
  pid_t pid; char *message; int n =2;
  puts("fork program starting\n");
  pid = fork();
  switch(pid)
    {
    case 0:
      message = "This is the child";
      break;
    default:
      message = "This is the parent";
      getchar();
      break;
    }
  for(; n > 0; n--) {
      puts(message);
      sleep(1);
  }
  return 0;
}
```

- When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait .
- Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait.
- This state of the process is known as defunct, or a **Zombie process**.
- If the parent terminates abnormally, the child process automatically gets the process with PID 1 as parent and known as **Orphan process.**
- A zombie orphan process is the last thing one expect to have in his system.

# Error Handling

```
#include<unistd.h>
#include<stdio.h>
int main() {
  pid_t pid; char *message; int n =2;
  puts("fork program starting\n");
  pid = fork();
  switch(pid)
    {
    case 0:
      message = "This is the child";
      break;
    default:
      message = "This is the parent";
      getchar();
      break;
    }
  for(; n > 0; n--) {
      puts(message);
      sleep(1);
  }
  return 0;
}
```

- The status information collected from wait system call allows the parent process to determine the exit status of the child process.
- You can interpret the status information using macros defined in sys/wait.h , shown in the following table.

# Error Handling

```
#include<unistd.h>
#include<stdio.h>
int main() {
  pid_t pid,child;char *message;
  int n=4,exitcode=0,status;
  puts("fork program starting\n");
  pid = fork();
  switch(pid) {
    case -1:
      perror("fork failed");
      exit 1;
    case 0:
      message = "This is the child";
      exitcode = 37;
      break;
    default:
      message = "This is the parent";
      getchar();
      break;
    }
}
```

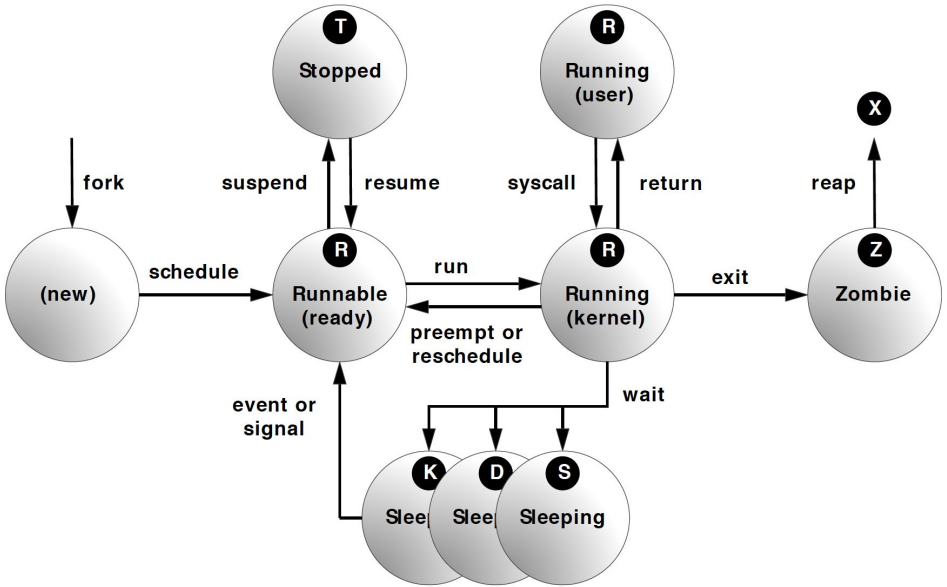
- The status information collected from wait system call allows the parent process to determine the exit status of the child process.
- You can interpret the status information using macros defined in sys/wait.h , shown in the

```
for(; n > 0; n--) {
    puts(message);
    sleep(1);
}
if (pid != 0) {
  child = wait(&status);
  printf("Child has finished:PID=%d\n",child);
  if(WIFEXITED(status))
    printf("Child exit code %d\n", WEXITSTATUS(status));
  else
    printf("Child terminated abnormally\n");
}
exit(exitcode);
```

# Listing / controlling Processes through ps & top

- View Process information with ps
  - Shows process from the current terminal by default
  - a includes process on all terminal
  - x includes process not attached to terminal
  - u  process owner information
- The Linux version of ps supports three option formats :
  - UNIX(POSIX) which start with a -
  - BSD options, where – is optional
  - GNU long options, which support --
- For example, ps -aux is not the same as ps aux.
- Use top command to interactively list and control the current process.

# Process status as per ps output




- R → TASK_RUNNING
- S → TASK_INTERRUPTIBLE
- D → TASK_UNINTERRUPTIBLE
- K → TASK_KILLABLE
- T → TASK_STOPPED
- Z → EXIT_ZOMBIE
- X → EXIT_DEAD

# What are process specific coding guidelines

- Handle errors after system call.
- Wait for child exit.
- Extract exit code using WEXITSTATUS().
- Avoid zombies.

# Useful Links

-

# Summary

- What is the concepts of process management.
  - Process creation
  - Parent – Child Relation in Process
  - Process Termination & the Role of  wait.
  - What are zombie and orphan process.
- What Linux system calls – fork(), wait()…. do
- How to write programs using system call to:
  - create processes
  - extract exit codes using macros
  - handle zombie processes etc.
- What are process specific coding guidelines.

## Any Questions ?

# Thank you!