Processes in UNIX

Mohiuddin Khan

CIT, IUT

Objectives

- Learn how to create processes
- Experiment with **fork** and **exec**
- Explore the implications of process inheritance
- Use **wait** for process cleanup
- Understand the UNIX process model

Process

- A program is a prepared sequence of instructions to accomplish a defined task
- A process is an instance of a program that is executing
- A process has an address space (memory it can access) and at least one flow of control called a thread.

Process identification

- UNIX identifies processes by a unique integral value called the process ID.
- Each process also has a parent process ID, which is initially the process ID of the process that created it.
- The getpid and getppid functions return the process ID and the parent process ID, respectively. pid_t is an unsigned integer type that represents a process ID.

Getting process ids

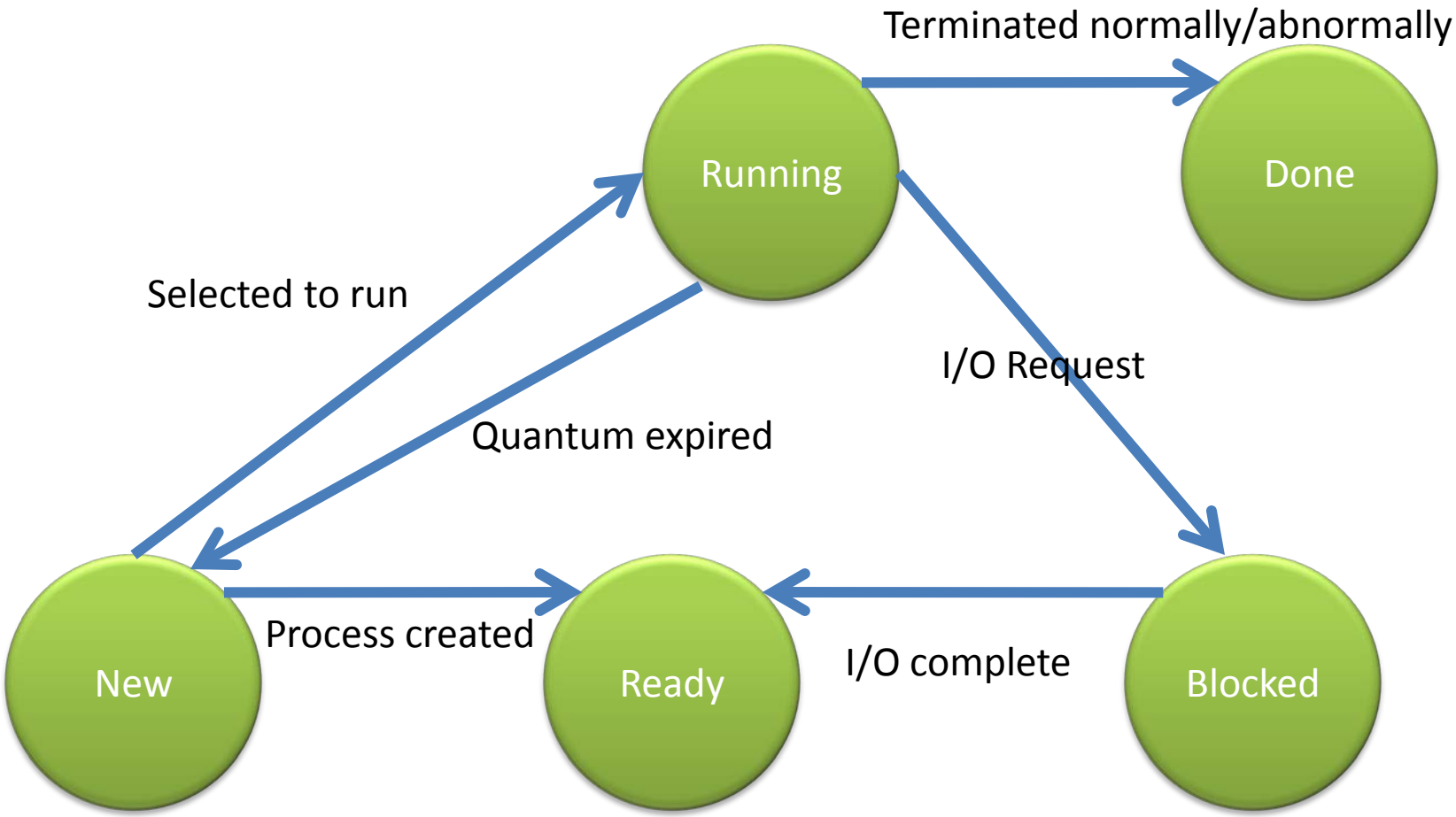
```
#include <stdio.h>
#include <unistd.h>
int main (void) {
        printf("I am process %ld\n",   (long)getpid());
        printf("My parent is %ld\n", (long)getppid());
        return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
        printf("My real user ID is %5ld\n", (long)getuid());
        printf("My effective user ID is %5ld\n", (long)geteuid());
        printf("My real group ID is %5ld\n", (long)getgid());
        printf("My effective group ID is %5ld\n", (long)getegid());
        return 0;
}
```

Process states

- New => being created
- Running => instructions are being executed
- Blocked => waiting for an event such as I/O
- Ready => waiting to be assigned to a processor
- Done => finished

Process states



UNIX Process creation

- A process can create a new process by calling fork.
- The calling process becomes the parent, and the created process is called the child.
- The fork function copies the parent's memory image so that the new process receives a copy of the address space of the parent.
- Both processes continue at the instruction after the fork statement
- The fork function returns 0 to the child and returns the child's process ID to the parent. When fork fails, it returns –1 and sets the errno.

Example: getpid( )

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
        int x;
        x = 0;
        fork();
        x = 1;
        printf("I am process %ld and my x is %d\n", (long)getpid(), x);
        return 0;
}
```

I am process 6128 and my x is 1
I am process 4624 and my x is 1

Example: fork ()

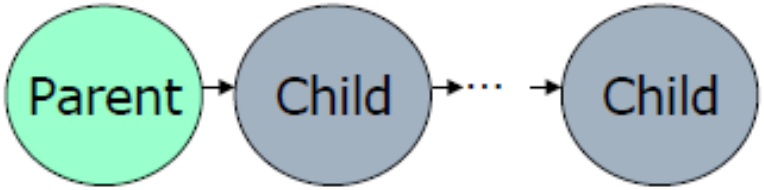
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
        pid_t childpid;
        childpid = fork();
        if (childpid == -1) {
                perror("Failed to fork");
                return 1;
         }
        if (childpid == 0) /* child code */
                printf("I am child %ld\n", (long)getpid());
        else /* parent code */
                printf("I am parent %ld\n", (long)getpid());
        return 0;
}
```

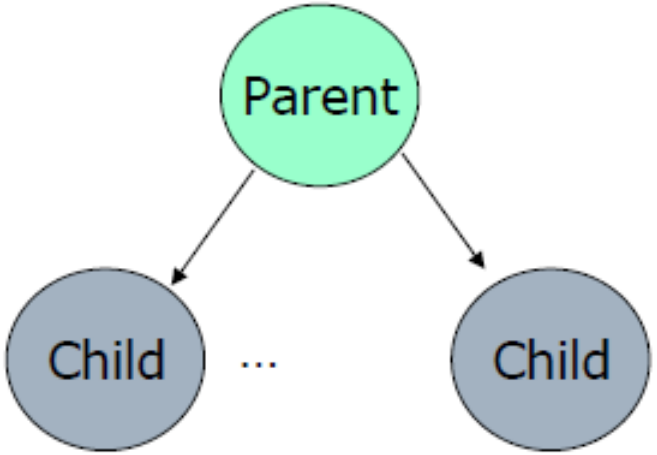
I am parent 3976
I am child 2940

fork: chain and fan

```
pid_t childpid = 0;
for (i=1;i<n;i++)
    if (childpid = fork())
            break;
```

```
pid_t childpid = 0;
for (i=1;i<n;i++)
    if ((childpid = fork()) <=0)
            break;
```





Exercise

- What happens when you replace the test
- (childpid = fork()) <= 0 with
- (childpid = fork()) == -1

Parent and child

- The child inherits parent attributes such as environment and privileges. The child also inherits some of the parent's resources such as open files and devices.
- Not every parent attribute or resource is inherited by the child. The child has a new process ID and of course a different parent ID.
- The child's times for CPU usage are reset to 0

The wait function

- When a process creates a child, both parent and child proceed with execution from the point of the fork.
- The parent can execute wait or waitpid to block until the child finishes.
- The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.

wait

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

```
pid_t childpid;
childpid = wait(NULL);
if (childpid != -1)
        printf("Waited for child with pid %ld\n", childpid);
```

waitpid

- allows a parent to wait for a particular child. This function also allows a parent to check whether a child has terminated without blocking
- takes three parameters: a pid, a pointer to a location for returning the status and a flag specifying options.
- If pid is –1, waitpid waits for any child.
- If pid is greater than 0, waitpid waits for the specific child whose process ID is pid.

waitpid

- If wait or waitpid returns because the status of a child is reported, these functions return the process ID of that child. (normal case)
- If an error occurs, these functions return –1 and set errno.
- If called with the WNOHANG option, waitpid returns 0 to report that there are possible unwaited-for children but their status is not available.

waitpid example

```
pid_t childpid;
while (childpid = waitpid(-1, NULL, WNOHANG))
        if ((childpid == -1) && (errno != EINTR))
                break;
```

- Waits for all children that have finished. But it avoids blocking if there are no children whose status is available.
- It restarts waitpid if that function is interrupted by a signal or if it successfully waited for a child.

Zombieland

- What happens when a process terminates, but its parent does not wait for it?
- It becomes a zombie in UNIX terminology. Zombies stay in the system until they are waited for.
- If a parent terminates without waiting for a child, the child becomes an orphan and is adopted by a special system process that periodically waits for children

Possible outputs for this?

```
if (childpid == 0)
        fprintf(stderr, "I am child %ld\n", (long)getpid());
else if (wait(NULL) != childpid)
        fprintf(stderr, "A signal must have interrupted the wait!\n");
else
        fprintf(stderr, "I am parent %ld with child %ld\n", (long)getpid(),
(long)childpid);
        return 0;
```

- Output will depend on the timing of a possible signal and execution of child/parent

Status variables

- The stat_loc argument of wait or waitpid is a pointer to an integer variable.
- If it is not NULL, these functions store the return status of the child in this location.
- The child returns its status by calling exit, _exit, _Exit or return from main.
- A zero return value indicates EXIT_SUCCESS; any other value indicates EXIT_FAILURE.

Checking status variables

- #include <sys/wait.h>
- WIFEXITED(int stat_val)
- WEXITSTATUS(int stat_val)
- WIFSIGNALED(int stat_val)
- WTERMSIG(int stat_val)
- WIFSTOPPED(int stat_val)
- WSTOPSIG(int stat_val)

Checking status variables

- These macros are designed to be used in pairs
- The WIFEXITED evaluates to a nonzero value when the child terminates normally.
- If WIFEXITED evaluates to a nonzero value, then WEXITSTATUS evaluates the value returned by the child through _exit(), exit() or return from main
- And similarly for the other two pairs...

The exec function

- Unlike fork, many applications require the child process to execute code that is different from that of the parent.
- The job of exec is to replace the current process with a new process
- The traditional way to use the fork–exec combination is for the child to execute the new program while the parent continues to execute the original code

Example: a child process to run ls -l

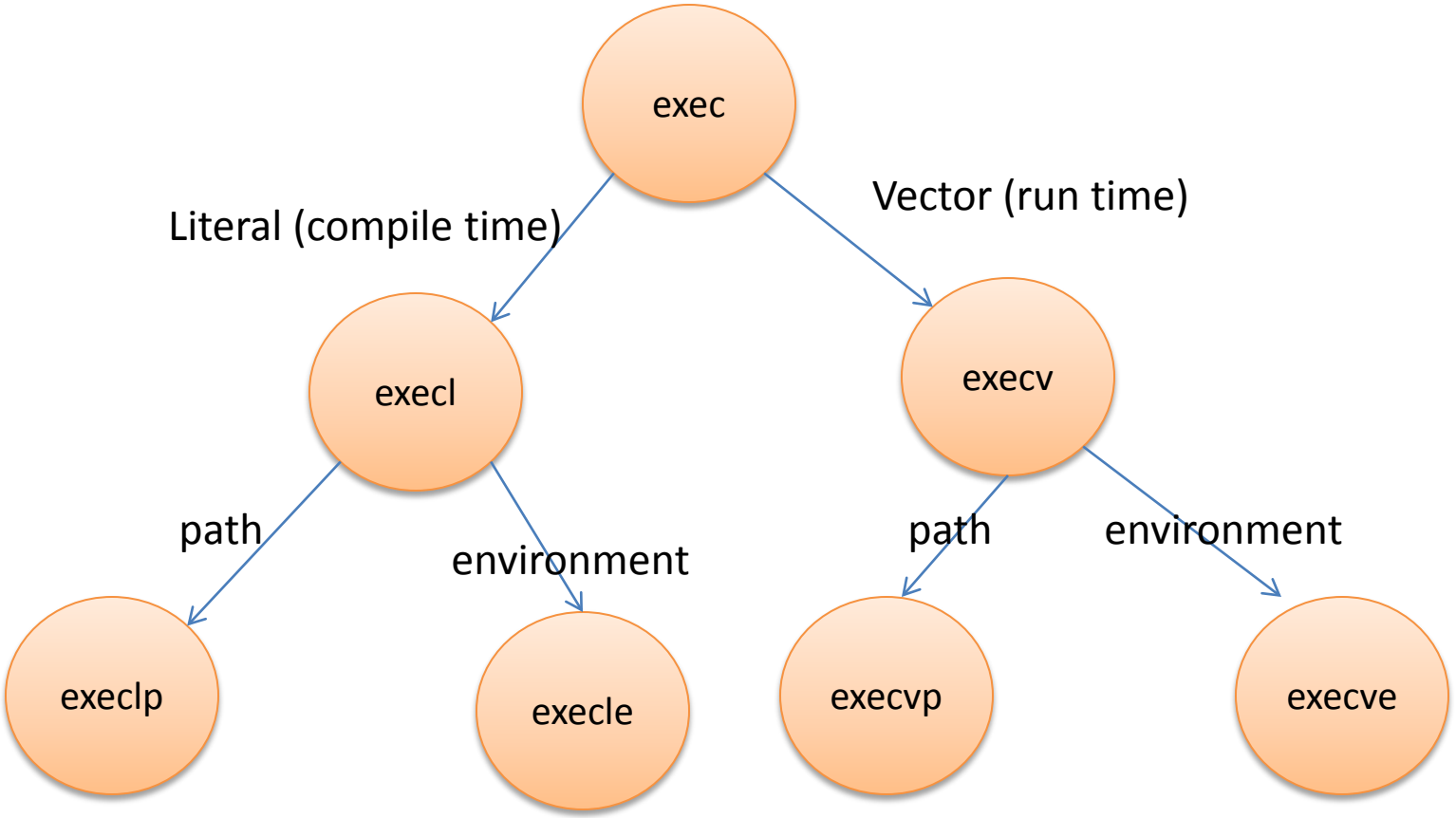
```
childpid = fork();
if (childpid == -1) {
        perror("Failed to fork");
        return 1;
 }
if (childpid == 0) {
        /* child code */
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Child failed to exec ls");
        return 1;
}
if (childpid != wait(NULL)) {
        /* parent code */
        perror("Parent failed to wait due to signal or error");
        return 1;
}
```

Six degrees of “exec”

- The six variations of the exec function differ in the way command-line arguments and the environment are passed.
- They also differ in whether a full pathname must be given for the executable.
- The execl (execl, execlp, execle) functions pass the command-line arguments in an explicit **list**
- The execv (execv, execvp, execve) functions pass the command-line arguments in an argument array

Exec variations

- Ex: execv(arg[0], arg)



Background processes and daemons

- Most shells interpret a line ending with & as a command that should be executed by a background process.
- Ctrl-C from the keyboard does not terminate a background process. (unlike normal processes)
- A daemon is a background process that normally runs indefinitely.
- UNIX relies on many daemon processes to perform routine (and not so routine) tasks.

Critical sections

- The problem: processes "simultaneously" attempt to access a shared resource
- Example: A shared printer
- Solutions: locks, semaphores, mutex

Reference

- Chapter 3, “Unix™ Systems Programming: Communication, Concurrency, and Threads” By Kay A. Robbins, Steven Robbins