System Programming

4. Concurrent Processes

Seung-Ho Lim
Dept. of Computer & Electronic Systems Eng.

Concurrent Processes (1)

- A running program-system consists of several dynamic concurrent processes
- A set of collaborating processes with IPC
- Concurrent process programming tools
  - Concurrent languages (Ada, Java, PathPascal, Modula II)
  - C, C++,.. with system calls for concurrent programming (Linux, UNIX, Win32 Environment)

Concurrent Processes (2)

- Applications (Objectives)
  - Networked (or distributed) applications
    - Network App : client processes, server processes
    - Distributed App : chess program, cloud computing
  - Real-time applications
    - Aircraft control : cockpit display, Flight management,
    - Missile system, Automobile : many sensors, controllers, and servo motors(actuators) are handled by multiple processes
    - Usually, a process for each sensor
  - Parallel applications: 3D graphic, many CPUs, many parallel processes
    - To utilize more CPUs, GPUs
  - CPU/IO overlap in a program level
    - When a process does computing, another process does I/O in a program.
  - Asynchronous event handling
    - Handling of multiple input sources: don't know which device makes input first.
    - In this case, assign a process to each input device.

Process & PID

- Every process has a unique process ID (PID, a non-negative integer)
  - Although they are unique, the IDs are reused after termination
  - PID 0 is usually the scheduler known as the *swapper*
  - PID 1 is usually the init process (invoked at the end of the boostrap procedure, located in `/etc/init` or `/sbin/init`)
  - PID 2 is *`pagedaemon`* responsible for supporting the paging of the virtual memory system
- Each process has its own PCB(process control block)
  - contains all the information about a process
    - PID, UID, GID, current working directory, terminal, priority, state, CPU usage, memory map, open files and locks, ...

System calls for PIDs

```
#include <unistd.h>

pid_t getpid(void);
// Returns: process ID of calling process

pid_t getppid(void);
// Returns: parent process ID of calling process

uid_t getuid(void);
// Returns: real user ID of calling process

uid_t geteuid(void);
// Returns: effective user ID of calling process

gid_t getgid(void);
// Returns: real group ID of calling process

gid_t getegid(void);
// Returns: effective group ID of calling process
```

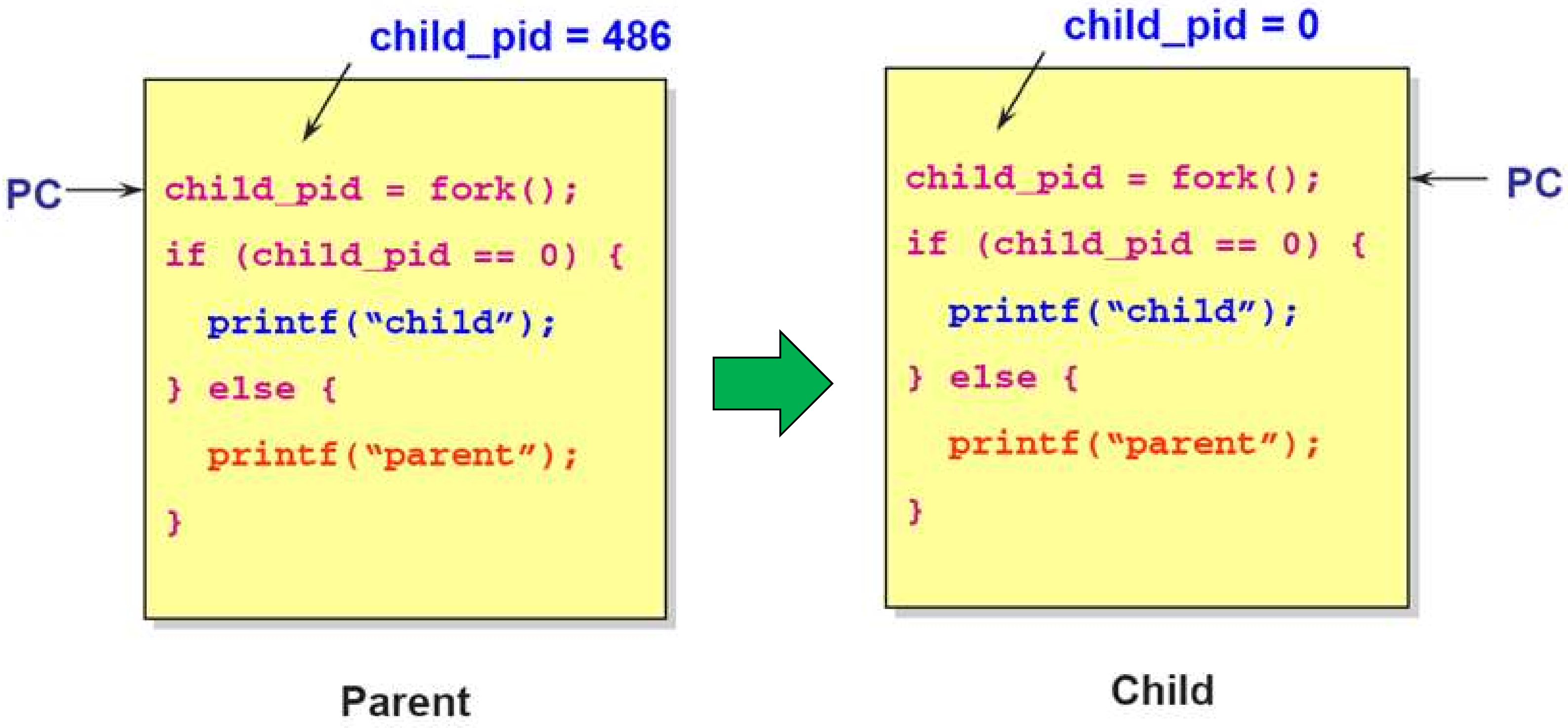
Process creation in Linux

```
#include <unistd.h>

pid_t fork(void);
```

- creates a new process called *child process*
  - the child is a clone (copy) same as the parent
  - the child starts by returning from this fork() call (Why?)
- returns
  - PID of the new born child process in parent process
  - 0 in the child process
  - -1 on error

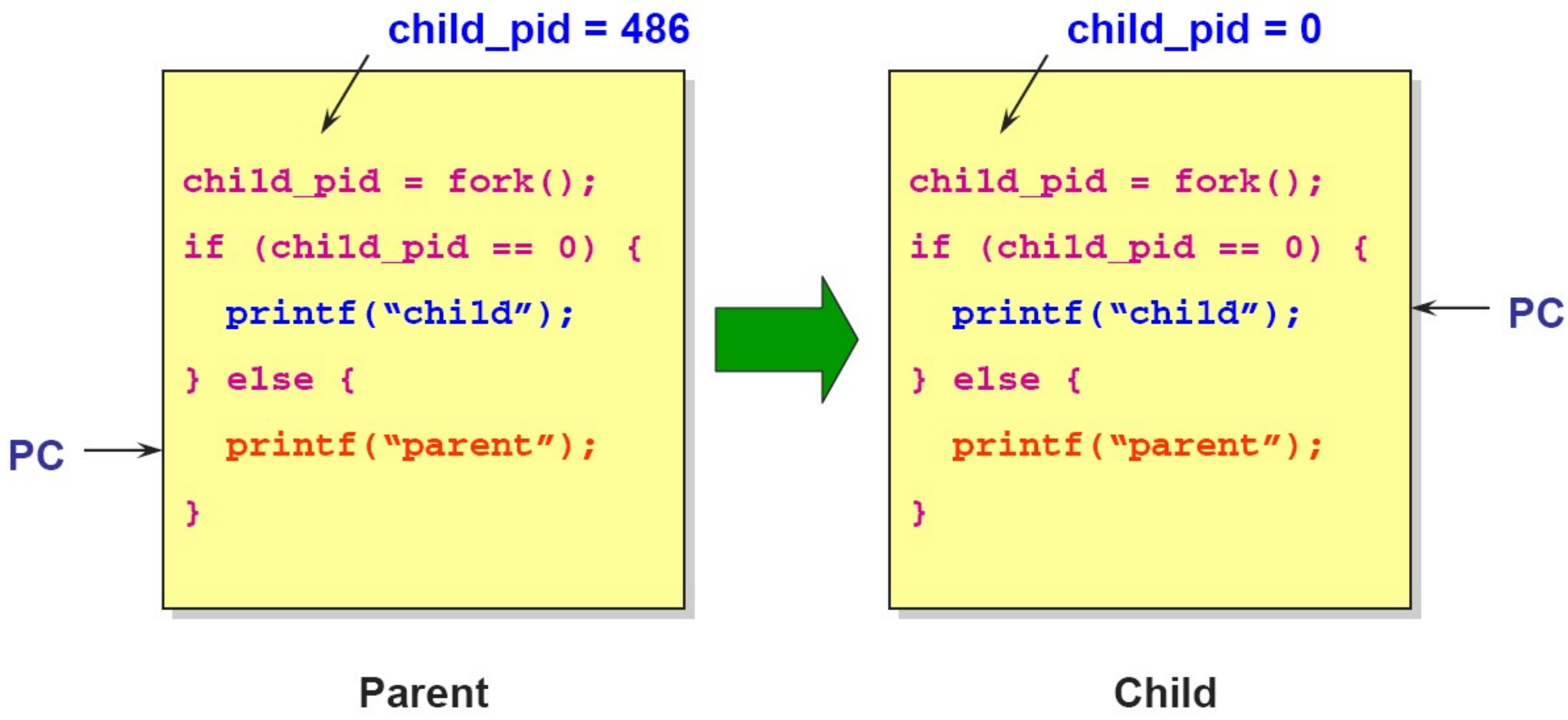
fork() example (1)

- Duplicating the address spaces



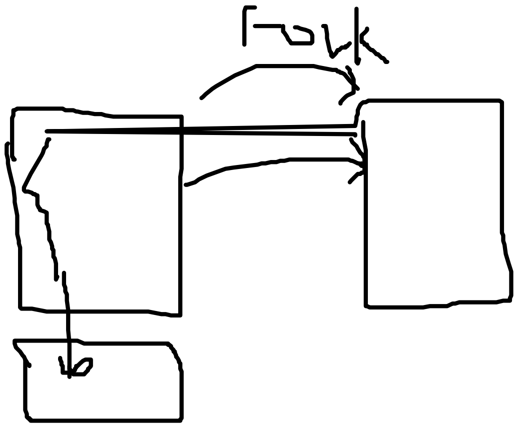
fork() example (2)

- Divergence of their execution flow



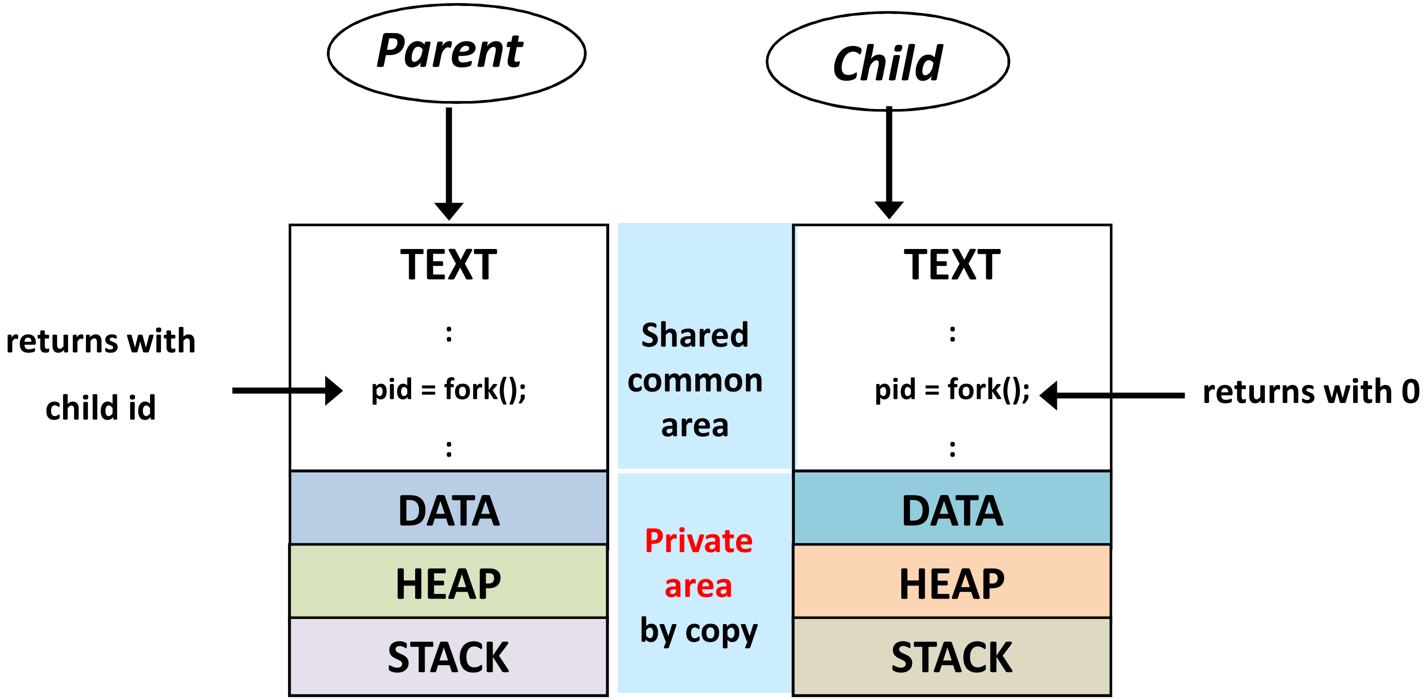
Parent and Child (1)

- Child is a copy (or clone) of the parent process
- Child inherits most resources of the parent at the fork() call
  - share the same text (program code and constant data)
  - real/effective user-id, group-id,
  - current working, root directory,
  - open files before fork including tty (stdin, stdout, stderr),
  - share r/w offsets of the files which the parent opened before fork.



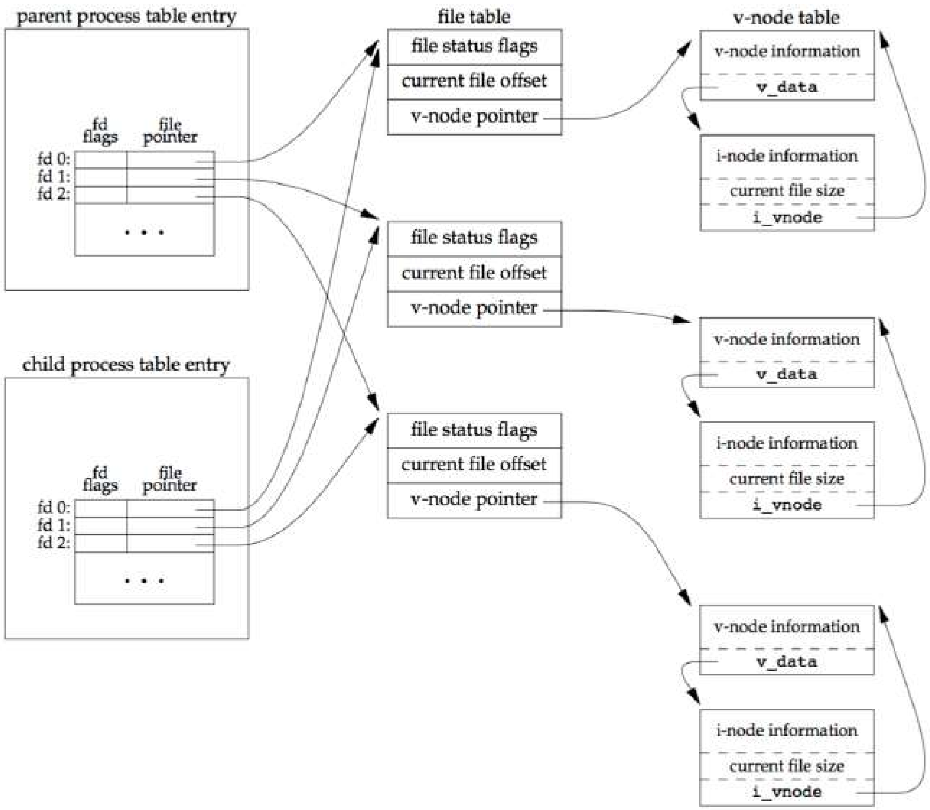
Parent and Child (2)

- But, in reality, those are different processes
  - different PID, different PCB,
  - scheduled independently (concurrently) by the kernel
  - each has some private resources
  - different data/heap/stack (copied when forked)



Parent and Child (3)

- the fork() may fail
  - if too many processes are already in the system
  - if the total number of processes for this real user ID exceeds the system's limit (CHILD_MAX)



Terminating a process (1)

```
#include <stdlib.h>

void exit(int status);
```

- `exit()` function sends status to its parent
  - the status informs how the child process has been terminated
  - its parent gets the status value via `wait()` call (see later!)
- A process can terminate normally in five ways
  - executing a return from the main function.
  - calling the `exit()` function.
  - calling the `pthread_exit` function from the last thread in the process.

Terminating a process (1)

- Kernel code executes these for a terminated process
  - the code closes all the open descriptors for the process
  - releases the memory allocated

cf. the terminated process remains in the system until its parent returns from `wait()` system call

- called zombie process in Linux

Waiting a child process (1)

```
#include <sys/wait.h>
pid_t wait(int *status);
```

- caller (usually parent) will be blocked until any child terminates
  - when resumes, removes the PCB of the child.(a funeral)
- return
  - terminated PID if Ok
  - -1 on error
- parameters
  - int *status : variable address where the status will be strored

Waiting a child process (2)

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

- caller (usually parent) will be blocked until the designated process (pid) terminates
  - same as wait() except the pid designation
  - how to make `wait()` do as `waitpid()`?
- parameters
  - pid
    - if pid>0 waits for the child whose process ID equals pid
    - if pid == -1 waits for any child process, `waitpid` is equivalent to wait
  - options

    WNOHANG: for non-blocking call

Macros for exit status

- WEXISTATUS(status)
  - fetch the exit code which the child sends
- WIFEXITED(status)
  - true if the child terminated normally
- WIFSIGNALED(status)
  - true if status was child terminated abnormally by a signal
- WIFSTOPPED(status)
  - true if status was returned for a child that is currently stopped
- WTERMSIG(status)
  - fetch the signal number that caused the child to be terminated
- WCOREDUMP(status)
  - true if a core file or the terminated process was generated

wait() and exit() example

```
#include <wait.h>

main()
{
    int res, pid ;

    res = fork() ;

    if (res == 0) {
        // do something
        exit(77);   //  77 is an example exit status
    }
    else {
       pid = wait(&status) ; // waiting until the child terminates
       printf("the child %d is exited with %d \n",
                  pid, WEXITSTATUS(status)) ;
    }
}
```

```
#include <wait.h>

main()
{
    int res, pid ;

    res = fork() ;

    if (res == 0) {
        // do something
        exit(77);   //  77 is an example exit status
    }
    else {
       pid = wait(&status) ; // waiting until the child terminates
       printf("the child %d is exited with %d \n", pid,
                  pid, WEXITSTATUS(status)) ;
    }
}
```

parent process **child process**

exec() system call

- When a process calls one of the exec functions, that process is completely replaced by the new program
  - the new program starts executing at its main function
  - PID does not change across an `exec`
  - `exec` merely replaces the current process (text, data, heap, and stack segment)
- The program body is changed but the followings remain
  - priority
  - process id (same process), working directory
  - time left until an alarm
  - pending signals
  - open files and file locks

exec family (1)

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ... /* (char *)0, char *const
envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);

// All return: -1 on error, no return on success
```

exec family (2)

- Difference

Function	pathname	filename	fd	arg list	arvg[]	environ	envp[]
execl	•			•		•	
execlp		•			•	•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
letter		p	f	l	v		e

- letters
  - *v* in its name, argv's are a vector: const * char argv[]
  - *l* in its name, argv's are a list: const * char arg0, ...
  - *e* in its name, takes environment variables: char * const envp[]
  - *p* in its name, PATH environment variable to search for the file

exec() call example

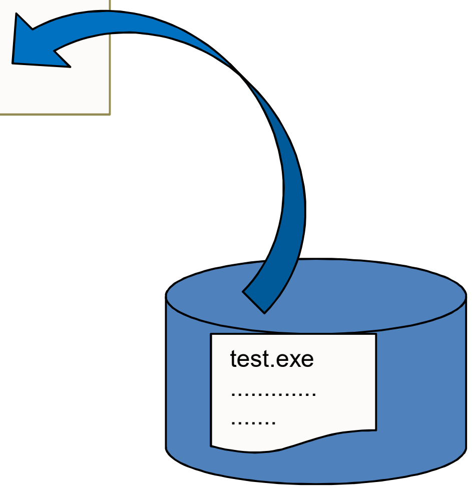
```
main()
{

   pid = fork() ;
   if (pid==0)
      exec("test.exe");

}
```

parent process

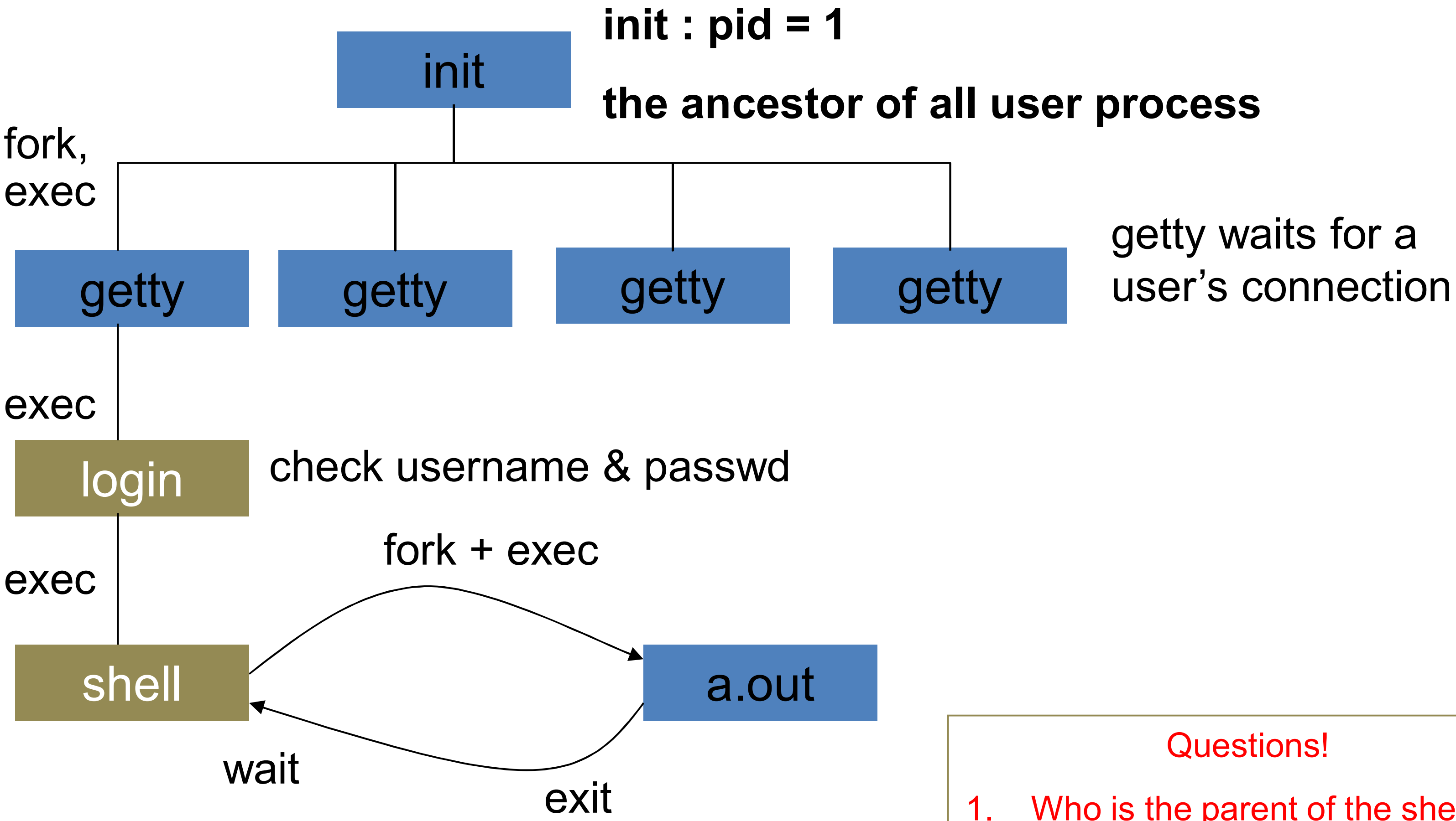
```
main()
{

}
```

```
// test.exe program
main()
{
 // this program replace all address
 // space of the forked process.
     printf("Hello, world! \n");
}
```

child process



User process tree



Questions!

1. Who is the parent of the shell?
2. Who is the parent of the a.out?

Shell & process tree

$: shell, the process of this user

$ ./a.out : shell and a.out : processes of this user

$ cat myfile : shell and cat : processes of this user

$ ./a.out & : a.out as a background process

$ rm myfile : shell, a.out and rm are running processes

$ ./a.out > output : stdout is changed to "output" before exec().
This is called "I/O redirection".

Simple *shell* example

```
while (not logout) {
      print prompt;
      get a line string;
      parse the line; // a.out, a.out&, ...
      if ((pid = fork()) == 0) { // child
            ...         // IO redirection if designated
            execv("parse name", argv);
      }
      if (foregroud) {
            while (pid != wait()); // wait for the death of the foreground process
          // if a background did exit(), it is removed here as a side effect.
      }  // if background, does not wait()
}
// You can make your own shell !
```