

System calls for process management

Mythili Vutukuru
CSE, IIT Bombay

API for process management

- What API does OS provide to user programs to manage processes?
 - How to create, run, terminate processes?
- API = Application Programming Interface
 - = functions available to write user programs
- API provided by OS is a set of “system calls”
 - System call is a function call into OS code that runs at higher CPU privilege level
 - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
- Some “**blocking**” system calls cause the process to be blocked and context switched out (e.g., `read()` from disk), while others (e.g., `getpid()` to get PID) can return immediately

Portability of code across OS

- **POSIX API**: standard set of system calls (and some C library functions) available to user programs, defined for portability
 - Programs written using POSIX API can run on any POSIX compliant OS
 - Most modern OSes are POSIX compliant
 - Program may still need to be recompiled for different architectures
- Program language libraries hide the details of invoking system calls
 - The `printf` function in `libc` calls the `write` system call to write to screen
 - User programs usually do not need to worry about invoking system calls
- **ABI (application binary interface)** is the interface between machine code and underlying hardware: ISA, calling convention, ...

Process related system calls (in Unix)

- `fork()` creates a new child process
 - All processes are created by forking from a parent
 - OS starts `init` process after boot up, which forks other processes
 - The `init` process is ancestor of all processes, including shell/terminal
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants of the above system calls exist in language libraries with different arguments

Process creation: fork

- Parent process calls “fork” system call to create (spawn) a new process
- New child process created with new PID
- Memory image of parent is copied into that of child
- Parent and child run different copies of same code

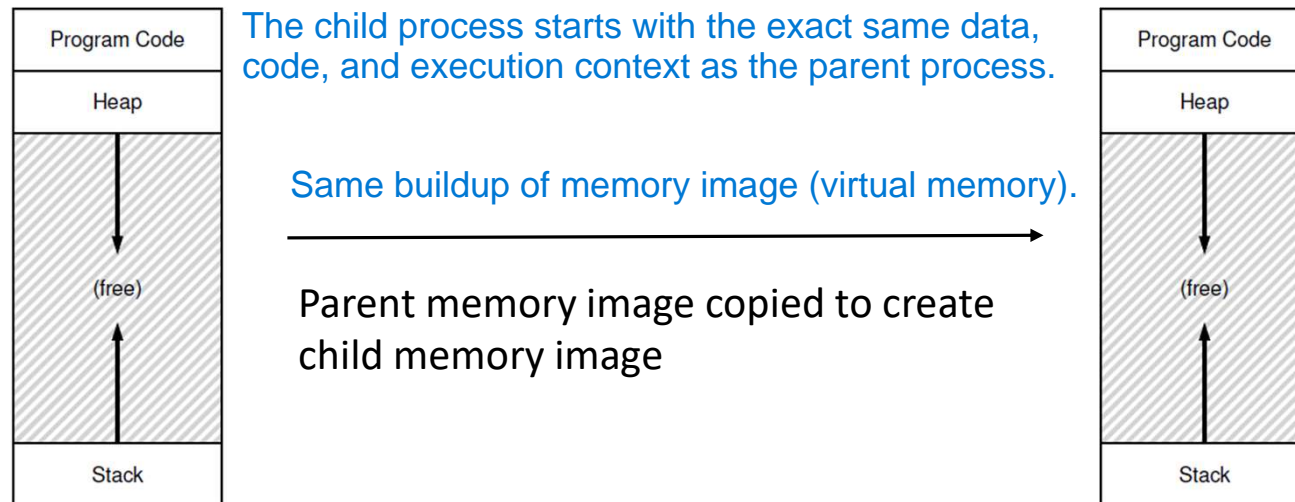


Image credit: OSTEP

When fork happens parent and child have different virtual pages [similar to each other] but they point to same physical page (address) and it is separated only when actual write is required in one of them.

What happens after fork?

- Parent and child resume execution in their copies of the code
- Child starts executing with a return value of 0 from fork
- Parent resumes executing with a return value equal to child PID
- Parent and child run independently
- Any changes in parent's data after fork does not impact child

```
int ret = fork()
if(ret == 0) {
    print "I am child"
}
else if(ret > 0) {
    print "I am parent"
}
```

Child resumes
here

Parent resumes
here

```
int ret = fork()
if(ret == 0) {
    print "I am child"
}
else if(ret > 0) {
    print "I am parent"
}
```

Parent gets the return value of child's PID, and child gets the return value 0.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {          // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {               // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18     }
19     return 0;
20 }

```

Can get the parent PID as well using `getppid()`.

Figure 5.1: Calling `fork ()` (`p1.c`)

Example code with fork

- Parent and child run independently and print to screen
- Order of execution of parent and child can vary

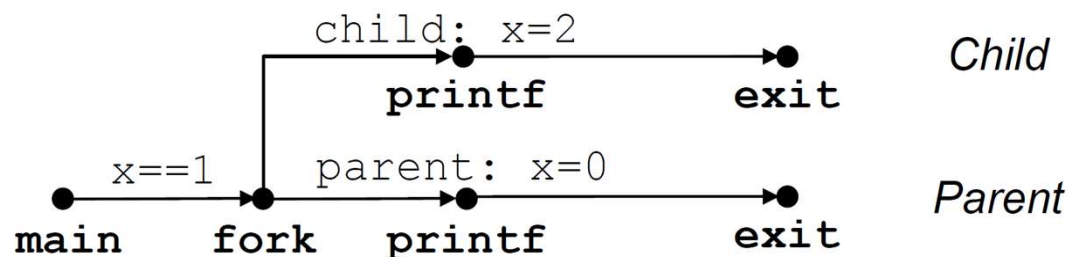
When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>

prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```


Example code with fork

- What values of x are printed?
- Parent and child both start with their own independent copies of variable x in their memory images
- Child increments its copy of x, prints 2
- Parent decrements its copy of x, prints 0

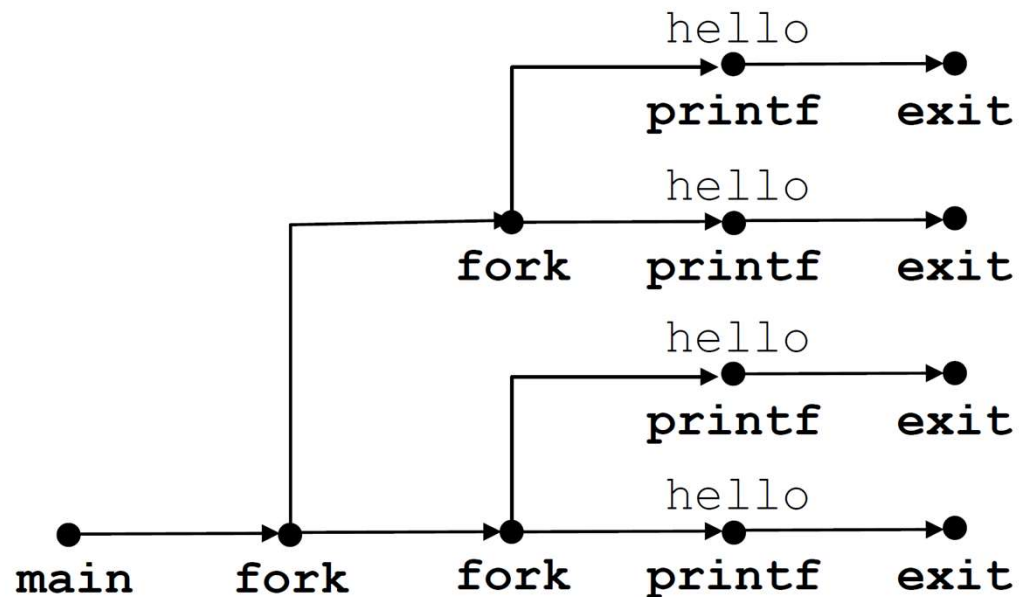


```
int ret = fork()
int x = 1
if(ret == 0) {
    print "I am child"
    x = x+1
    print x
}
else if(ret > 0) {
    print "I am parent"
    x = x -1
    print x
}
```

Example code with nested fork

- Total 4 processes (1 parent + 3 child)
- Hello printed 4 times

```
fork()  
fork()  
print hello  
exit
```



Exit system call

- When a process finishes execution, it calls `exit` system call to terminate
 - OS switches the process out and never runs it again
 - Exit is automatically called at end of main
- Exiting process cannot clean up its memory, and memory must be freed up by someone else (why? More on this later.)
- Terminated process exists in a `zombie` state
- How are zombies cleaned up?

Wait system call

- Parent calls `wait` system call to `reap` (clean up memory of) a zombie child
- `Wait` cleans up memory of **one** terminated child and returns in parent process One wait cleans one child!!
- If child still running, `wait` system call blocks parent until child exits
 - If child terminated already, `wait` reaps child and returns immediately
 - If parent with no child calls `wait`, it returns immediately without reaping anything
NO error!!!

```
...  
int ret = fork()  
if(ret == 0) {  
    print "I am child"  
    exit()  
}  
else if(ret > 0) {  
    print "I am parent"  
    wait()  
}  
...
```

More on wait

- Wait system call variant `waitpid` reaps a specific child with a given PID, while regular `wait` reaps any terminated child
 - Read man pages for more details on arguments to `waitpid` and `wait`
- Wait system call “reaps” one dead child at a time (in any order)
 - Every fork must be followed by call to wait at some point in parent
- What if parent has exited while child is still running?
 - Child will continue to run, becomes orphan
 - Orphans adopted by init process, reaped by init when they terminate
- If parent forks children, but does not bother calling wait for long time, system memory fills up with zombies
 - Common programming error, exhausts system memory

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      printf("hello world (pid:%d)\n", (int) getpid());
10     int rc = fork();
11     if (rc < 0) {          // fork failed; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) { // child (new process)
15         printf("hello, I am child (pid:%d)\n", (int) getpid());
16     } else {              // parent goes down this path (main)
17         int wc = wait(NULL);
18         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19               rc, wc, (int) getpid());
20     }
21     return 0;
22 }

```

Figure 5.2: Calling `fork()` And `wait()` (p2.c)

Example code with fork and wait

- Order of printing of child and parent is deterministic now
- Why? Parent waits until child prints and exits, then prints
because of `wait()` system call which waits for child to finish executing and then reaps out its memory.

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

Exec system call

- Isn't it impractical to run the same code in all processes?
 - Sometimes parent creates child to do similar work..
 - .. but other times, child may want to run different code
- Child process uses “exec” system call to get a new “memory image”
 - Allows a process to switch to running different code
 - Exec system call takes another executable as argument
 - Memory image is reinitialized with new executable, new code, data, stack, heap, ...
exec system call gives everything new with the new executable.

```
...  
int ret = fork();  
if(ret == 0) {  
    exec("some_executable")  
}  
else if(ret > 0) {  
    print "I am parent"  
}  
...
```



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {          // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc"); // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;          // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else {                // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26               rc, wc, (int) getpid());
27     }
28     return 0;
29 }

```

l : in the form of strings
v : in the form of null terminated array.

Figure 5.3: Calling `fork()`, `wait()`, And `exec()` (p3.c)

Example code with exec

- Many variants of exec system call (execvp used in example), which differ in the arguments provided (read more in man pages)
- If exec successful, child gets new memory image, never comes back to the code in old memory image after exec
 - Print statement after exec doesn't run if exec successful
- If exec unsuccessful, reverts back to original memory image

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29      107      1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

Shell / Terminal

- After bootup, the `init` process is first process created
- The `init` process spawns a shell like `bash`
- All future processes are created by forking from existing processes like `init` or shell
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls`, `echo`, `cat` are all readily available executables that are simply exec-ed by the shell

Example shell code

- How does the shell run a user command?
- Read input from user
- Shell process **forks** a child process
- Child process runs **exec** with “echo” program executable as argument, calls **exit** when done
- Parent shell calls **wait**, blocks till child terminates, reaps it, goes back for next input

```
$echo hello  
hello  
$
```

```
do forever {  
    input(command)  
  
    int ret = fork()  
  
    if(ret == 0) {  
        exec(command)  
    }  
    else {  
        wait()  
    }  
}
```

More on shell and commands

- Some commands already exist as programs written by OS developers and compiled into executables
 - Shell runs such command by simply calling `exec` in child process
- Some commands are implemented directly in shell code itself
- Think: why doesn't shell `exec` command directly? Why fork a child?
 - Do we want the shell program code to be rewritten fully?
- For “`cd`” command, “`chdir`” system call used to change directory of parent process itself, no child process is forked. Why?
 - Every process has a current working directory
 - Do we want to change directory of some child process or shell itself?

```
$sleep 10 &  
$
```

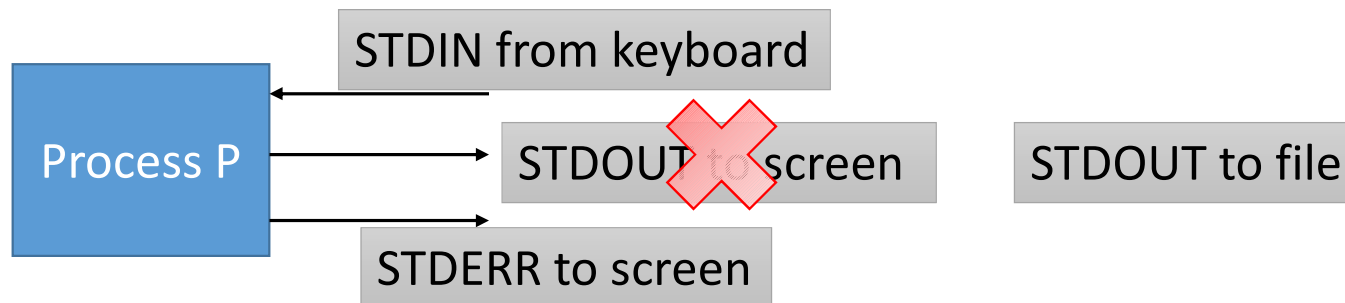
Foreground and background execution

- By default, user command runs in foreground, shell cannot accept next command until previous one finishes
- Background execution: when we type command followed by &
 - Shell starts child to run command, but does not wait for command to finish
- Background processes reaped at a later time by shell
 - When? Periodically? When next input is typed?
 - How? There is a way to invoke wait where parent is not blocked even if child has not exited (explore it on your own)
- It is also possible to run multiple commands in the foreground
 - One after the other serially (next command starts after previous finishes)
 - Or, all start at same time in parallel
 - Explore how such things can be done in the standard Linux shell

```
$ls > foo.txt  
$
```

I/O redirection

- Every process has some I/O channels (“files”) open, which can be accessed by file descriptors
 - STDIN, STDOUT, STDERR open by default for all processes
- Parent shell can manipulate these file descriptors of child before exec in order to do things like I/O redirection
- E.g., output redirection is done by closing the default STDOUT and opening a regular file in its place



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/wait.h>
7
8  int
9  main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) {          // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc"); // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL;          // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else {                  // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }

```

Here is the output of running the p4.c program:

```

prompt> ./p4
prompt> cat p4.output
      32      109      846 p4.c
prompt>

```

Open uses the first available file descriptor (STDOUT in this case)

Figure 5.4: All Of The Above With Redirection (p4.c)

Image credit: OSTEP

Signals

- Signal: a way to send notifications to processes
- Standard signals available in operating systems, each corresponding to a specific event, and with a specific signal number
 - Signal SIGINT sent to process by typing Ctrl+C, SIGSTP for Ctrl+D
 - Signal SIGCHLD sent to parent process when child terminates
 - SIGTERM and SIGKILL to terminate/kill processes
- System call `kill` can be used to send a signal from one process to other
 - Kill system call can send all signals, not just SIGKILL
 - Some restrictions on who can send to whom for isolation and security
- Signals can also be generated by OS for a process, e.g., when it handles interrupt due to Ctrl+C keyboard event
- Kill command to send signals, e.g., “kill -9 <pid>” sends SIGKILL (#9)

Process groups

- When we type Ctrl+C on keyboard, which processes get the signal?
- Processes are organized into process groups, every process belongs by default to process group of its parent
- When signal is sent to a process, it is delivered to all processes in its process group by default
- Example: when we hit Ctrl+C on keyboard, signal sent to all processes in the foreground process group
- System call `setpgid` can be used to change process group of signals, to control signal distribution

Signal handling

- Signals to a process are queued up by OS and delivered when process is running
- Default behavior defined by OS for a process receiving a signal
 - Ignore some signals (e.g., SIGCHLD)
 - Terminate when some signals are received (e.g., SIGINT)
- User processes can define their own signal handler functions to be executed when a signal is received
 - Override default behavior defined for a signal
 - Some signals (e.g., SIGKILL) cannot be overridden
- Process jumps to signal handler, executes it, resumes if still alive

Examples: sending and catching signals

- Parent sends SIGKILL to child using kill system call
- Child runs in infinite loop until killed by parent

```
int pid = fork()
if(pid == 0) {
    while(1); //infinite loop
    //terminates on SIGKILL
}
//parent
kill(pid, SIGKILL)
```

- Default SIGINT handler overridden
- Process prints message before terminating on SIGINT

```
void sigint_handler(int sig) {
    print "caught signal"
    exit()
}
int main() {
    signal(SIGINT, sigint_handler)
    ...
}
```