3rd Seminar

Processes; Signals

What can happen?

```
#!/bin/bash
F=$1
N=0
while [ $N -lt 200 ]; do
        K=`cat $F`
        K=`expr $K + 1`
        echo $K > $F
        N=`expr $N + 1`
done
```

```
#!/bin/bash
echo 0 > a.txt
# The script on the left is inc.sh
./inc.sh a.txt &
./inc.sh a.txt &
./inc.sh a.txt &
```

What can happen?

```
#include <everything.h>
int main(int argc, char** argv)
{
   int f, k, i;
   f = open(argv[1], O_RDWR);
   if(argc > 2 && strcmp(argv[2], "reset") == 0)
   {
          k = 0;
          write(f, &k, sizeof(int));
          close(f);
          return 0;
    }
          for(i=0; i<255*255; i++)
          {
                 lseek(f, 0, SEEK_SET);
                 read(f, &k, sizeof(int));
                 k++;
                 lseek(f, 0, SEEK_SET);
                 write(f, &k, sizeof(int));
          }
          close(f);
          return 0;
}
```

Execution

```
/* Script for simultaneous execution

#!/bin/bash

./inc b.dat reset

./inc b.dat &

./inc b.dat &

./inc b.dat &

*/
```

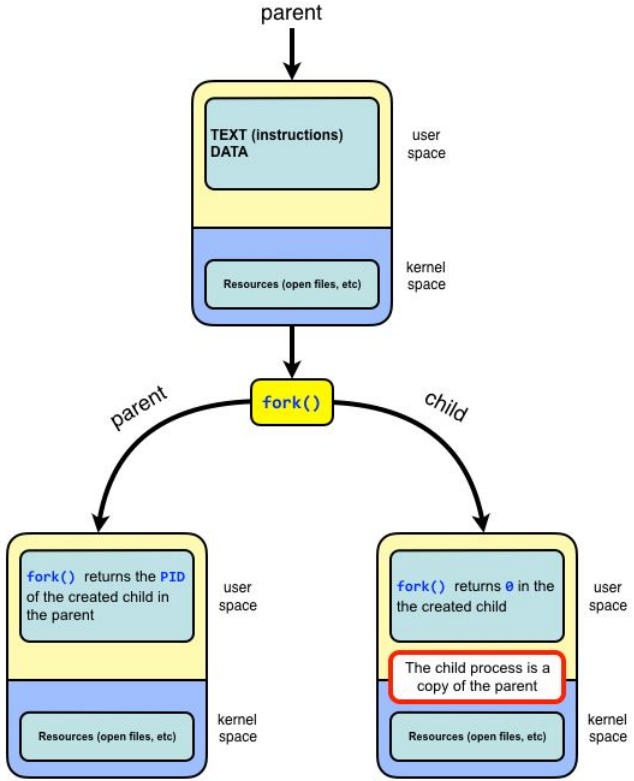
Processes

The process invoking fork is called the **parent**. The new process created as the result of a fork is the **child** of the parent.

After a successful fork, the child process is a copy of the parent. The parent and child processes executes the same program but in separate processes

On success **fork returns twice**: once in the parent and once in the child.

On success, the PID of the **child process is returned in the parent,** and **0 is returned in the child**. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.



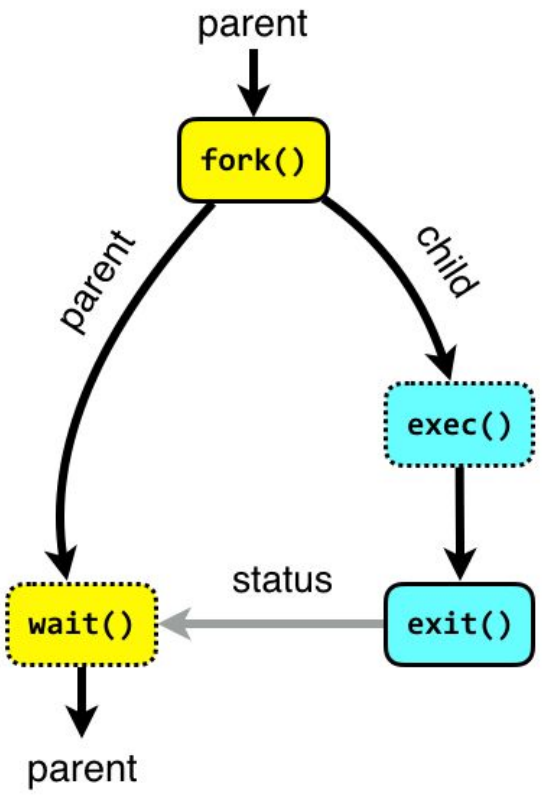
Processes

fork: a parent process creates a child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

exec: replaces the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different program than the parent.

exit: terminates the process with an exit status.

wait: the parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child.

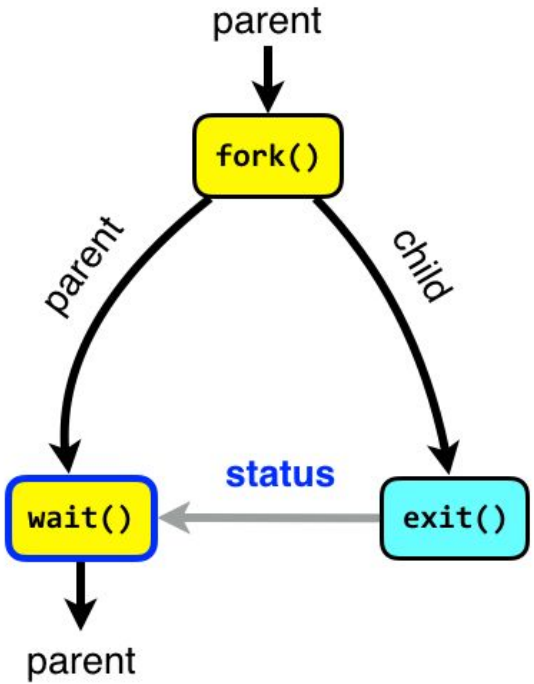


Processes

The wait system call blocks the caller until one of its child process terminates. If the caller doesn't have any child processes, wait returns immediately without blocking the caller. Using wait the parent can obtain the exit status of the terminated child.

```
pid_t wait(int *status);
```

status: If status is not NULL, wait store the exit status of the terminated child in the int to which status points. Can be inspected using the WIFEXITED and WEXITSTATUS macros.
Return val: On success, wait returns the PID of the terminated child. On failure (no child), wait returns -1.



Processes

```
WIFEXITED(status);
```

status: The integer status value set by the wait system call.

Return val: Returns true if the child terminated normally, that is, by calling exit or by returning from main.

```
int WEXITSTATUS(status);
```

status: The integer status value set by the wait system call.

Return val: The exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit or as the argument for a return statement in main. This macro should be employed only if WIFEXITED returned true.

Processes

A process whose parent process has terminated is said to be **orphan**. Any orphaned process will be immediately adopted by the special init system process with PID 1.
A terminated process is said to be a **zombie** or defunct until the parent calls wait on the child.
When a process terminates all of the memory and resources associated with it are deallocated so they can be used by other processes.
However, the exit status is maintained in the PCB until the parent picks up the exit status using wait and deletes the PCB (Process Control Block, structure in kernel).
A child process always first becomes a zombie.
In most cases, under normal system operation zombies are immediately waited on by their parents.
Processes that stay zombies for a long time are generally an error and cause a resource leak.

Processes

The correct way to create a child process must involve calls to fork, exit and wait.

One process:

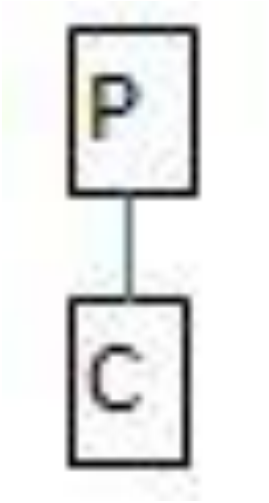
```
pid = fork();
if(pid == 0) {
        // do some stuff
        exit(0);
}
// do some other stuff
wait(0);
```

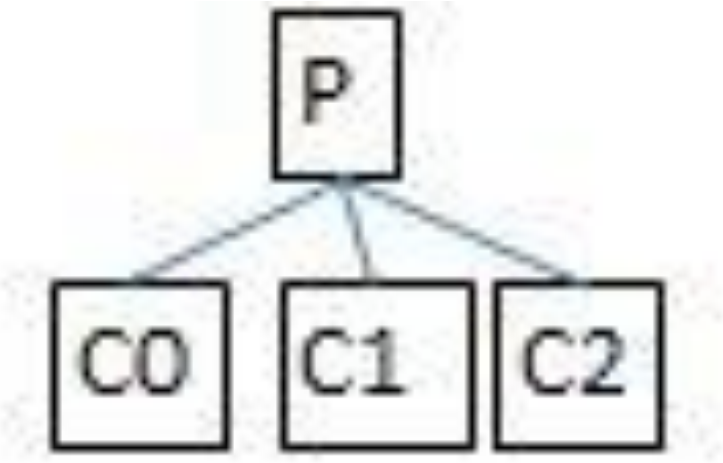
Several processes:

```
for(i=0; i<3; i++) {
        pid = fork();
        if(pid == 0) {
        // do some stuff
        exit(0);
        }
}
// do some other stuff
for(i=0; i<3; i++) {
        wait(0);
}
```

Processes

What are the hierarchies for the 2 examples above?

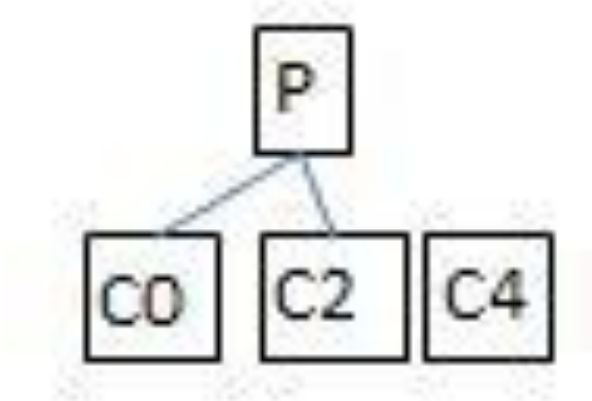




Processes

Draw the process hierarchy of the code below

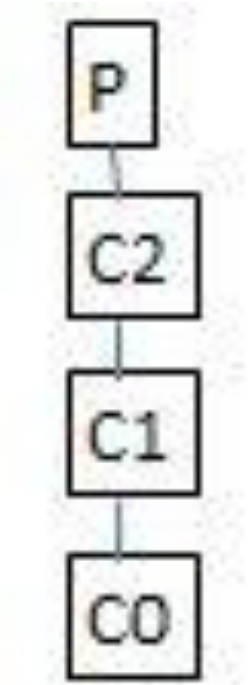
```
for(i=0; i<6; i++) {
        if(i % 2 == 0) {
                pid = fork();
                if(pid == 0) {
                        exit(0);
                }
        }
}
```



Processes

Draw the process hierarchy when calling f(3), f being the function defined below.

```
void f(int n) {
        if(n > 0) {
                if(fork() == 0) {
                        f(n-1);
                }
                wait(0);
        }
}
```



Signals

Signals are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems.

A signal is used to notify a process of an synchronous or asynchronous event.

When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal.

If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.

Each signal is represented by an integer value. Instead of using the numeric values directly, the named constants defined in signals.h should be used.

Signals

A program can install a signal handler using the signal function.

```
signal(sig, handler);
```

sig: The signal you want to specify a signal handler for.
handler: The function you want to use for handling the signal.

```
void hello(int signum){
  printf("Hello World!\n");
}
int main(){
  signal(SIGINT, hello);
  while(1);
}
```

To send a signal from terminal to a specific process, the **kill** command can be used: `$ kill -s INT <PID>`
To send a signal from terminal to all processes named the same, the **killall** command can be used: `$ killall cat`

Signals

To send a signal to a particular process, we use the kill() system call.

```
int kill(pid_t pid, int signum);
```

kill() takes a process identifier and a signal, in this case the signal value as an int, but the value is #defined so you can use the name.

```
void hello(){
  printf("Hello World!\n");
}
int main(){
  pid_t cpid, ppid;
  signal(SIGUSR1, hello);
  if ( (cpid = fork()) == 0){
        ppid = getppid();
        kill(ppid, SIGUSR1);
        exit(0);
   }else{
        wait(NULL);
   }
}
```

Signals

signal.h defines a set of actions that can be used in place of the handler:
  SIG_IGN : Ignore the signal
  SIG_DFL : Replace the current signal handler with the default handler

There are two signals that can never be ignored or handled: SIGKILL and SIGSTOP

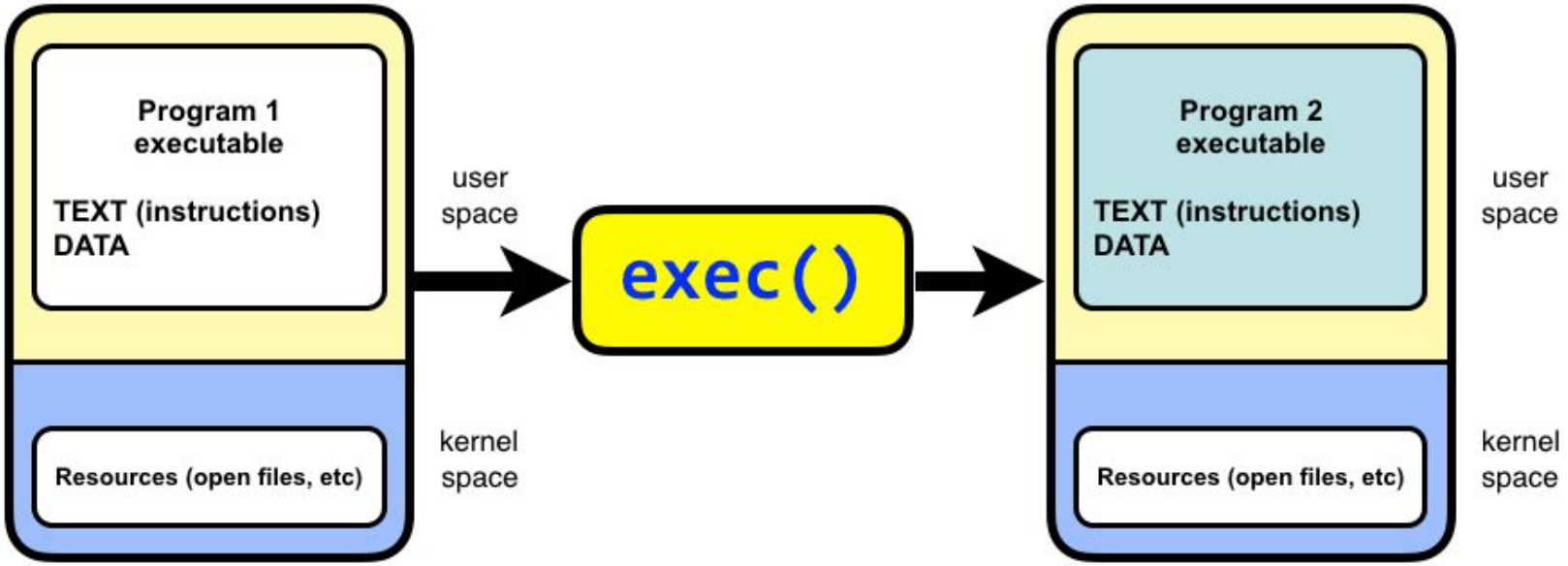
Signals - exercise

Implement a program that upon receiving SIGINT (ctrl-C) asks the user if he/she is sure the program should stop, and if the answer is yes, stops, otherwise it continues.

Signals - solution

```
#include <everything.h>
void f(int sgn) {
	char s[32];
	printf("Are you sure you want me to stop [y/N]? ");
	scanf("%s", s);
	if(strcmp(s, "y") == 0) {
		exit(0);
	}
}
int main(int argc, char** argv) {
	signal(SIGINT, f);
	while(1);
	return 0;
}
```

Exec system call



Exec system call

The exec family of system calls replaces the program executed by a process. When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program. Although all data is replaced, all open file descriptors remains open after calling exec unless explicitly set to close-on-exec.

Exec family

Functions in the exec() family have different behaviours:

- l : arguments are passed as a list of strings to the main()
- v : arguments are passed as an array of strings to the main()
- p : path/s to search for the new running program
- e : the environment can be specified by the caller

You can mix them, therefore you have:

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

The initial argument is the name of a file that is to be executed.

Exec family

```
//EXEC.c
#include<stdio.h>
#include<unistd.h>
int main()
{
    int i;
    printf("I am EXEC.c called by execvp() ");
    printf("\n");
    return 0;
}
```

Exec family

```
//execDemo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    //A null terminated array of character
    //pointers
    char *args[]={"./EXEC",NULL};
    execvp(args[0],args);
```

Exec family

```
    /*All statements are ignored after execvp() as this whole
    process(execDemo.c) is replaced by another one (EXEC.c)
    */
    printf("Ending-----");

    return 0;
}
```

Exec family

Write a C program that measures the duration of another programs execution given as command line argument along with its own arguments (i.e. ./measure grep “/an1/gr214/” /etc/passwd)

Exec - solution

```
#include <everything.h>
int main(int argc, char** argv) {
	char** a;
	int i;
	struct timeval start, finish;
	float duration;
	if(argc < 2) {
		fprintf(stderr, "No command provided.\n");
		exit(1);
	}
	a = (char**)malloc(argc*sizeof(char*));
	for(i=1; i<argc; i++) {
		a[i-1] = argv[i];
	}
```

Exec - solution

```
    a[argc-1] = NULL;
    gettimeofday(&start, NULL);
    if(fork() == 0) {
        if(execvp(a[0], a) < 0) {
            fprintf(stderr, "Command execution failed.\n");
            exit(1);
        }
    }
    wait(0);
    gettimeofday(&finish, NULL);
    duration = (
    (finish.tv_sec - start.tv_sec)*1000.0f +  // convert seconds difference to milliseconds
    (finish.tv_usec - start.tv_usec)/1000.0f  // convert microseconds diff to milliseconds
    )/1000.0f;                                // convert duration from milliseconds to seconds
    printf("Duration: %f seconds\n", duration);
    return 0;
}
```