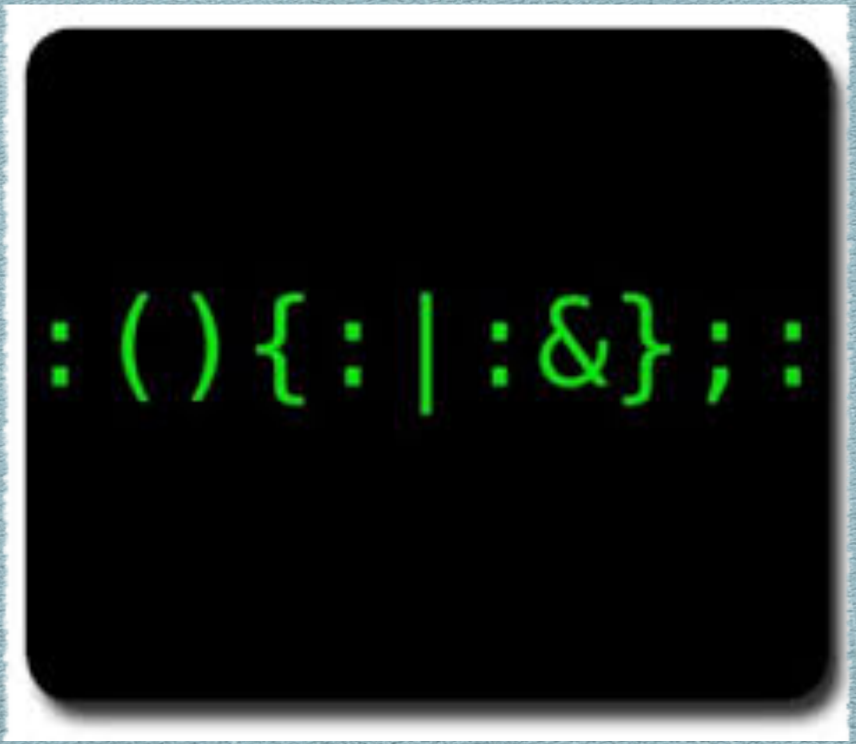


```
:(){:|:&};:
```

CSCI3150 Week 4 Tutorial

Calvin, Kam Ho Chuen (hckam @ cse)

2016

Outline

- Recap: What's C? 😬
- Knowing about process
- Let's fork()!
- Process Execution - exec()
- Wait! - wait() & waitpid()

Recap: What's C?

- Hope you still remember how to code in C..

```
#include <stdio.h> Header

int main(int argc,char* argv[])
{
        printf("Hello World!\n");
        return 0; Main Function
}
```

Knowing about process

- In order to identify a process, we need to find its process pid (pid).
- By using `ps` and `top`, we can check the status of processes
- To terminate (send signals) to them, use kill().
- Check out previous labs/man page for details.

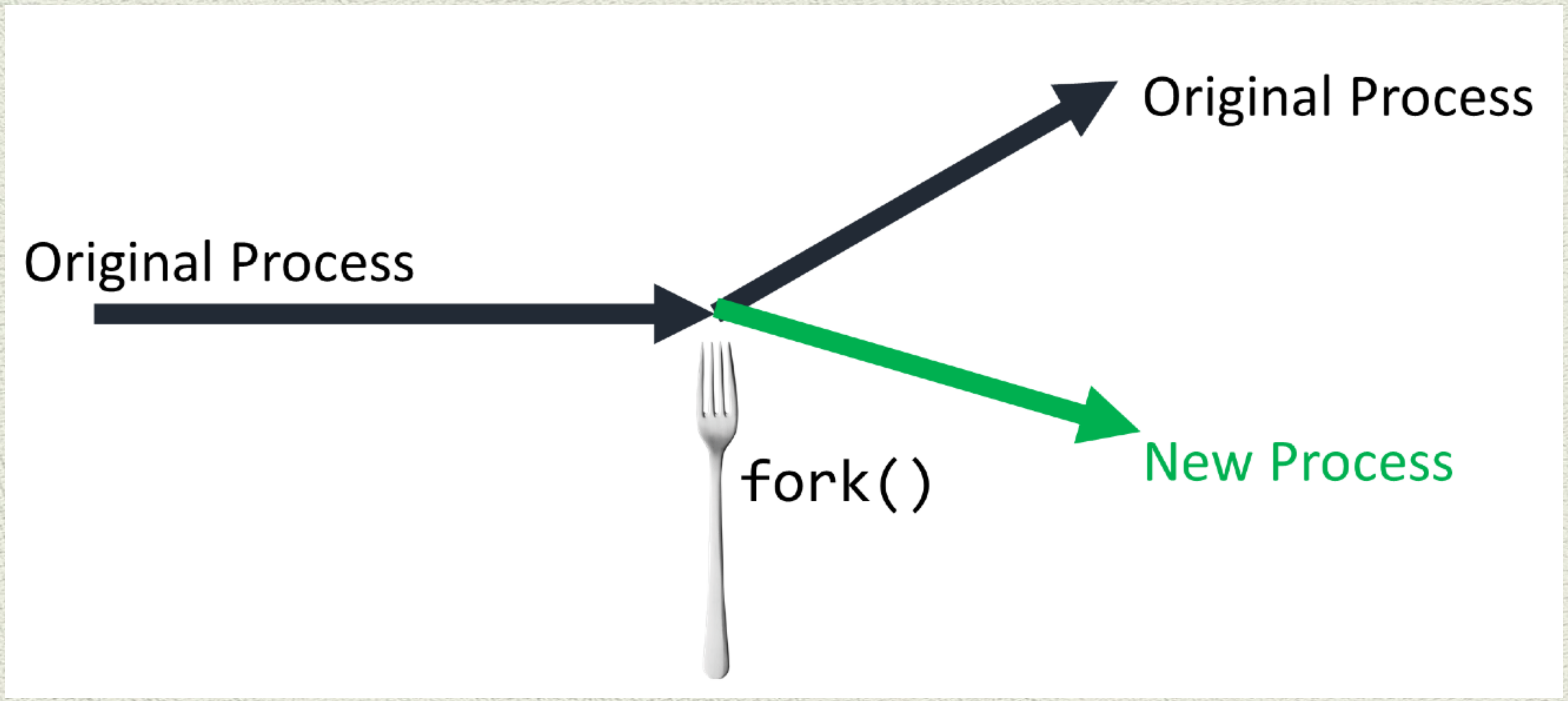
Knowing about process - PID

- We can use a function `getpid()` to get the PID of the current process.

```c
/* LetsFork/fork1.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
  printf("Before Fork, My PID: [%d]\n",getpid());
 fork();
 printf("After Fork, My PID: [%d]\n",getpid());
 return 0;
}
```

Let's fork!

- We create a process by a system call `fork()`.
- After calling, it will split into two: original process (parent) and new process (child).



Let's fork! First fork()

- Take a note of the following code:

```
/* LetsFork/fork1.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
 printf("Before Fork, My PID: [%d]\n",getpid());
 fork();
 printf("After Fork, My PID: [%d]\n",getpid());
 return 0;
}
```

Let's fork! - Behaviour of fork()

1. The child process will be spawned after fork.
2. The child will continue executing the code after fork() is returned, not from the beginning.
3. The parent still continues executing the same program.

Let's fork! Parents & Childs

- Then how can we know about which is parent and which is child?

```c
/* LetsFork/fork2.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc,char* argv[]) {
    int res;
    printf("Before fork():PID[%d]\n",getpid());
    res = fork();
    printf("The value of res is %d\n",res);
    if(res == 0) {
        printf("I am child! PID: [%d]\n",getpid());
    }
    else {
        printf("I am parent! PID: [%d]\n",getpid());
    }
    printf("Program Terminated!\n");
    return 0;
}
```

Let's fork! Parents & Childs

- Remember that:
- For the parent, the return value of fork() is the *PID of the new **child***.
- For the child, the return value of fork() is zero.

Let's fork! Let's try this

- For the following code, what are the printouts respectively?

```
/* LetsFork/fork_ex.c */
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A\n");
    fork();
    printf("B\n");
    fork();
    printf("C\n");
    return 0;
}
```

Let's fork! Let's try this

- Which one is more powerful (should be disastrous)?

```
while(fork());
```

vs

```
while(1) { fork(); }
```

Process Execution

- We can invoke external programs by using exec*().
- It has a series of family members.
- Let’s try!

Process Execution - First exec example

```c
/* Exec/execl.c */
#include <stdio.h>
#include <unistd.h>

int main(int argc,char* argv[])
{
    printf("Using *execl* to exec ls -l...\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("Program Terminated\n");
    return 0;
}
```

“Once you go exec(), you never go back”

It will not return to the original program, codes are changed to the new one.

Process Creation - Environment Variable

- In the shell the environment variables are a set of strings that stores the settings.
- E.g. $PATH, $HTTP_PROXY.
- You use them extensively in assignment 1 :P

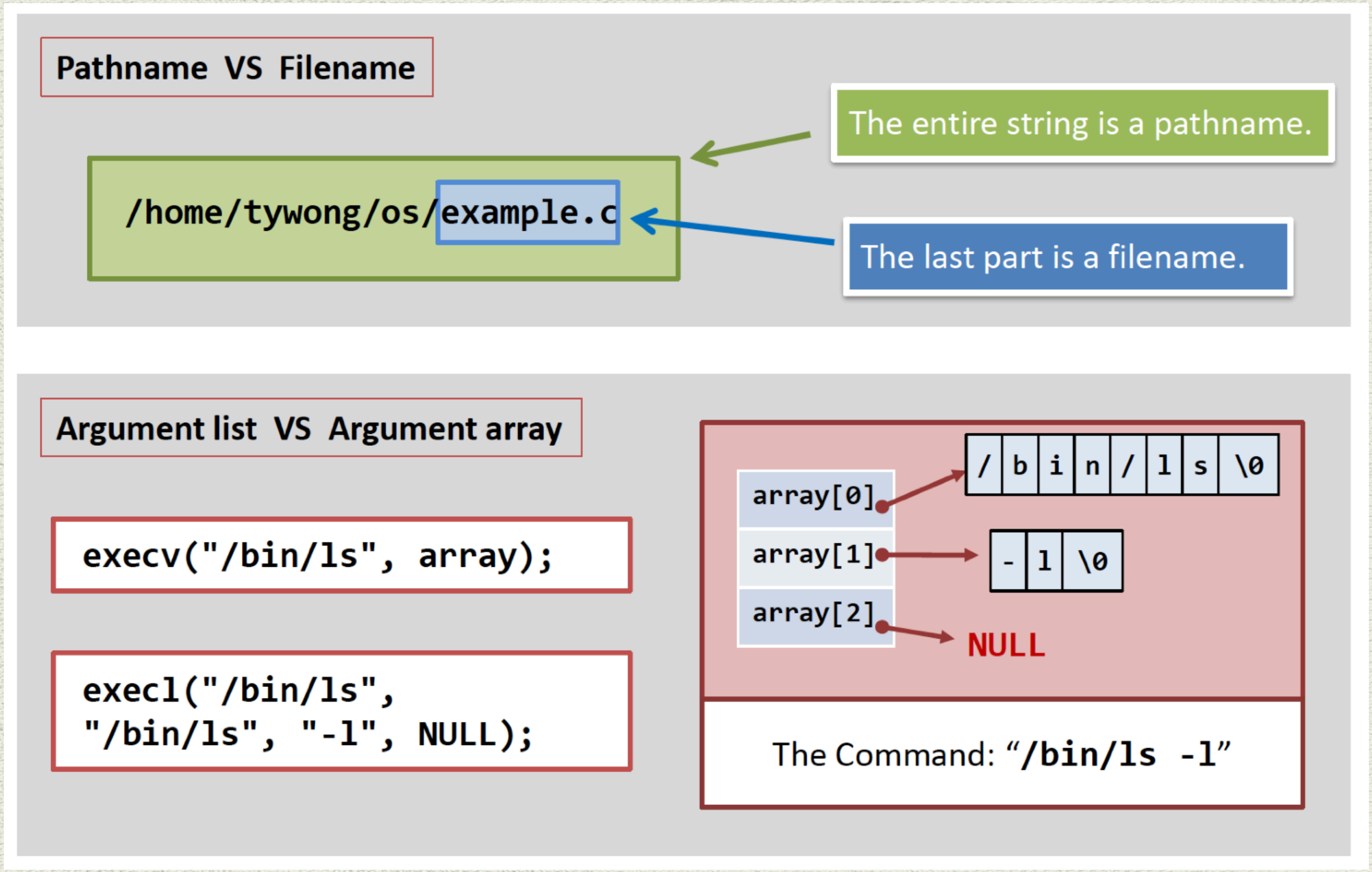
Process Creation - Environment Variable

```
/* Exec/env.c */
#include <stdio.h>
int main(int argc,char *argv[], char* envp[])
{
    int i;
    for (i = 0; envp[i];i++)
    {
        printf("[%d]: %s\n",i,envp[i]);
    }
    return 0;
}
```

Process Creation: exec*() family

Member name	Using pathname	Using filename	Argument List	Argument Array	Original ENV	Provided ENV
execl()	YES		YES		YES	
execlp()		YES	YES		YES	
execle()	YES		YES			YES
execv()	YES			YES	YES	
execvp()		YES		YES	YES	
execve()	YES			YES		YES
Alphabet used in name		**P**	**l**	**v**		**e**

Process Creation: exec*() family


Pathname VS Filename
/home/tywong/os/example.c
The entire string is a pathname.
The last part is a filename.
Argument list VS Argument array
array[0]
array[1]
array[2]
/ b i n / l s \0
- l \0
NULL

```
execv("/bin/ls", array);
```

```
execl("/bin/ls",
"/bin/ls", "-l", NULL);
```

The Command: “/bin/ls -l”

Process Creation: Error Handling

- If exec*() fails to invoke the program, it **WILL RETURN** to the original code and **CONTINUE** execution.
- Return Value: -1, errno is set.

Process Creation: Error Handling

```
/* Exec/exec_error.c */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int main(int argc,char *argv[])
{
    printf("Try to execute lss\n");
    execl("/bin/lls","lls",NULL);
    printf("execl returned! errno is [%d]\n",errno);
    perror("The error message is :");
    return 0;
}
```

Wait! Problem 1

- Let’s look at the following code:

```
/* Wait/problematic.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


int main(int argc,char *argv[])
{
    printf("Before fork...\n");
    if(fork() == 0)
    {
        printf("Hello World!\n");
        exit(0);
    }
 printf("After fork..\n");
 return 0;
}
```

Wait! Problem 1

- The order of execution is non-deterministic, i.e. it is random.
- We should have something to “pause” the parent and let the child run.

Wait! sleep()

- sleep() is a system call to make the process “sleeps” for a period of time.

Wait! sleep()

```c
/* Wait/sleep.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
    printf("Before fork...\n");
    if(fork() == 0)
    {
        printf("Hello World!\n");
        exit(0);
    }
    sleep(1);
    printf("After fork...\n");
    return 0;
}
```

Wait! Problem 2 Zombies!

Wait! Problem 2 Zombies!

```c
/* Wait/problematic2.c */
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]) {
    while(1) {
        printf("Press Enter to execute ls");
        while(getchar() != '\n');
        if(!fork()) {
            execl("/bin/ls","ls",NULL);
        }
        else {
            sleep(1);
        }
    }
    return 0;
}
```

Wait! Zombie

- If the parent is not aware of the terminated child, the child will still remains in the system. (ignoring SIGCHLD).
- It contains nothing but occupies one pid. (Think if there are lots of them…).
- Using wait() can solve the problem!

Wait! wait()

```
/* Wait/wait.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc,char *argv[]){
    while(1)  {
        printf("Press Enter to execute ls");
        while(getchar() != '\n');
        if(!fork()) {
            execl("/bin/ls","ls",NULL);
        }
        else  {
           wait(NULL);
        }
   }
    return 0;
}
```

Wait! waitpid()

- waitpid() is an advanced version of wait().
- It can tell you more about status of the children.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

PID that you wants to wait

Behaviour of waitpid()

Pointer that stores the status of the child

Wait! waitpid()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
    int pid;
    int status;
    if(!(pid = fork())) {
        printf("My PID: %d\n",getpid());
        exit(0);
    }
    waitpid(pid,&status,WUNTRACED);
    if(WIFEXITED(status))  {
        printf("Exit Normally\n");
        printf("Exit status: %d\n",WEXITSTATUS(status));
    }
    else  {
        printf("Exit NOT Normal\n");
    }
    return 0;}
```

Wait! waitpid()

- waitpid() has several **macros** for showing children's status.
- WIFEXITED(status) is to check whether the child exits normally (by exit() or return). (a.k.a. not exit by an interrupt).
- WEXITSTATUS(status) is to get the exit status (return value) of the child. Note that it only contains least significant 8 bits.

Exercise

- Try to write code to combine the uses of them to execute ls in a new process properly (without zombie!).

Summary

- Knowing about the process, getpid()
- Fork, exec, wait calls.

See you in next tutorial!

- Signals
- Remaining parts for assignment 2!
- Good Luck to your assignment 1 :P