Week 10 Lecture 2

NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

- Process management system calls

Recap: Types and examples of system calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

- Unix and Linux both use POSIX standard
- POSIX: Portable Operating System Interface

C and C++

Process management system calls

- `fork()`
- `exec()`

} Defined in `unistd.h`

- `wait()` } Defined in `sys/wait.h`
- `exit()` } Defined in `stdlib.h`

Process creation with `fork()`

- A process calling **`fork()`** spawns a child process.
- After a successful **`fork()`** call, two copies of the original code will be running
  - Parent process: *return value* of **`fork()`** → *child PID*.
  - **New** child process: *return value* of **`fork()`** → 0.

Process creation with `fork()`

- **`fork()`** is called once, but returns twice!
- After **`fork()`** both the parent and the child are executing the same program
- On error, **`fork()`** returns -1.

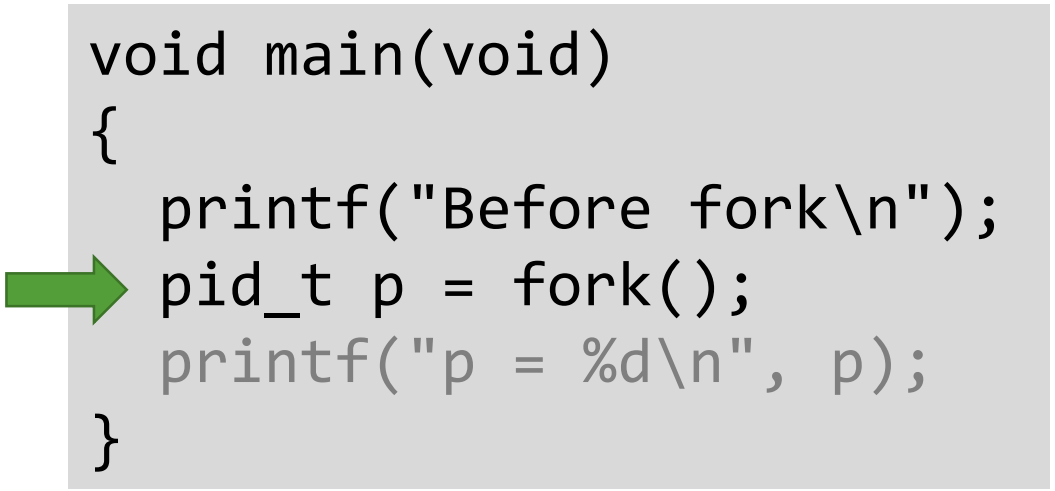
Illustration

Prior to `fork()` system call:

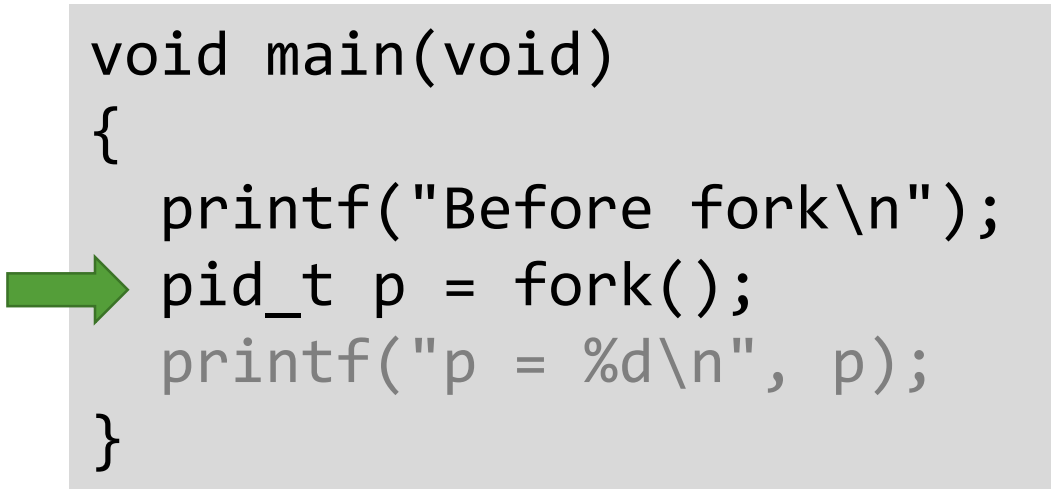
```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

Illustration

After `fork()` system call:



```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```



```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

Illustration

After `fork()` system call:

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

In parent, `fork()` will return PID of child

Illustration

After fork( ) system call:

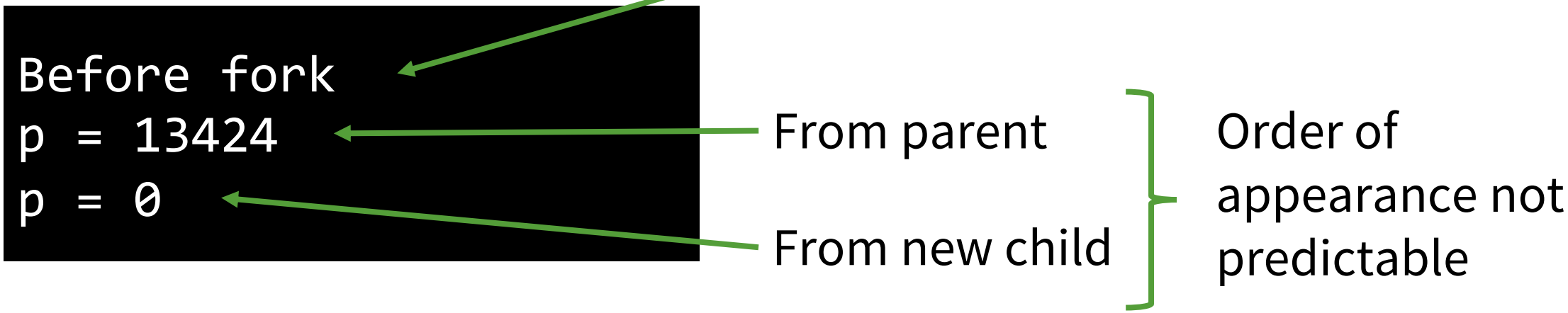
```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

In parent, fork( ) will return PID of child
In child, fork( ) will return 0

Output (if fork() is successful)

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```



Using the return value

Can use return value to determine what to do in parent and child

```
void main(void)
{
    printf("Before fork\n");
    pid_t p = fork();
    if(p < 0) {
        /* Failed to fork */
    } else if(p == 0) {
        /* Child process will execute this part */
    } else if(p > 0){
        /* Parent process will execute this part */
    }
}
```

Process ID

- To obtain the process ID of a process:

pid_t getpid(void);

```
void main(void)
{
    pid_t p = fork();
    if(p == 0) { /* Child */
        printf("My PID: %d\n", getpid());
    } else if(p > 0) { /* Parent */
        printf("My PID: %d, child PID: %d\n", getpid(), p);
    }
}
```

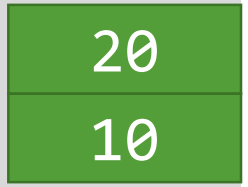
Variables

- After a successful **`fork()`** call, two copies of the original code will be running
- Parent and child will have their **own** copies of variables
- Variable changes in one process will not affect the variables in the other process

Illustration

Prior to fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```



Illustration

After fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 10

Illustration

After fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 21
a 10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 11

Illustration

After fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 21
a 10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 11

Illustration

After fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) {
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 21
a 10

```
10 21
11 20
```

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */

                == 0) { /* Child */

  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 11

Illustration

After fork( ) system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) {
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 21
a 10

```
11 20
10 21
```

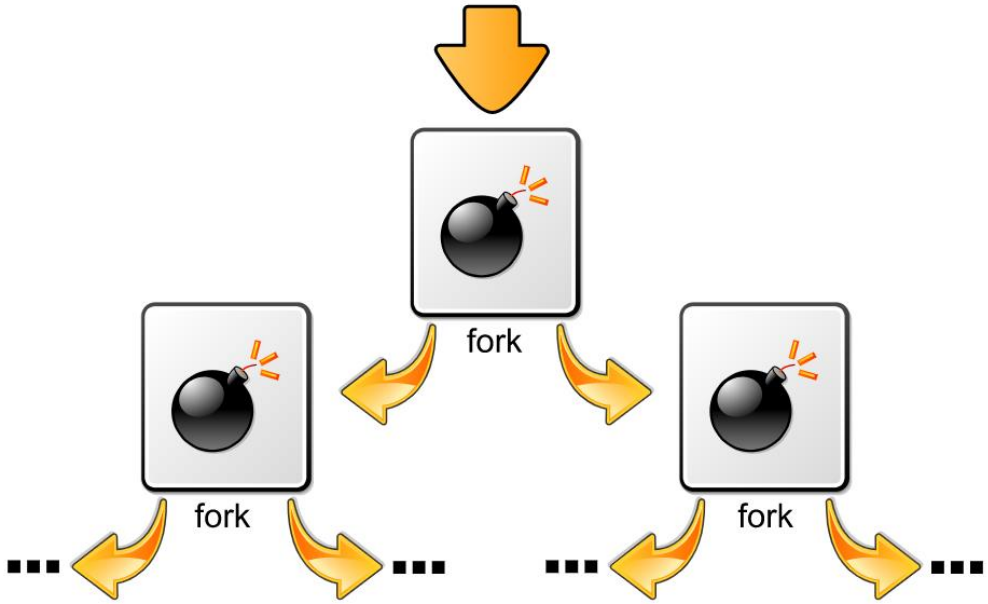
```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */

                == 0) { /* Child */

  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

b 20
a 11

Fork bomb

What will happen in this code?

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



Fork bomb (aka *wabbit* or *rabbit virus*): a form of denial of service attack to Linux based systems

Process creation with `exec()`

- **`exec()`** call replaces a current process' image with a new one (i.e. loads a new program within current process)

- Upon success, **`exec()` never** returns to the caller
  - If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.

- Arguments passed via **`exec()`** appear in the **`argv[]`** of the **`main()`** function.

Process creation with `exec()`

- There is no system call specifically by the name **`exec()`**
- By **`exec()`** we usually refer to a family of calls:
  - `int execl(char *path, char *arg, ...);`
  - `int execv(char *path, char *argv[]);`
  - `int execle(char *path, char *arg, ..., char *envp[]);`
  - `int execve(char *path, char *argv[], char *envp[]);`
  - `int execlp(char *file, char *arg, ...);`
  - `int execvp(char *file, char *argv[]);`
- The various options *l*, *v*, *e*, and *p* mean:
  - *l* : an argument list,
  - *v*: an argument vector,
  - *e*: an environment vector, and
  - *p*: a search path.

Illustration

Prior to exec( ) system call:

```
void main(void)
{
  printf("Before exec\n");
  int r = execl("/bin/ls", "ls", NULL);
  printf("r = %d\n", r);
}
```

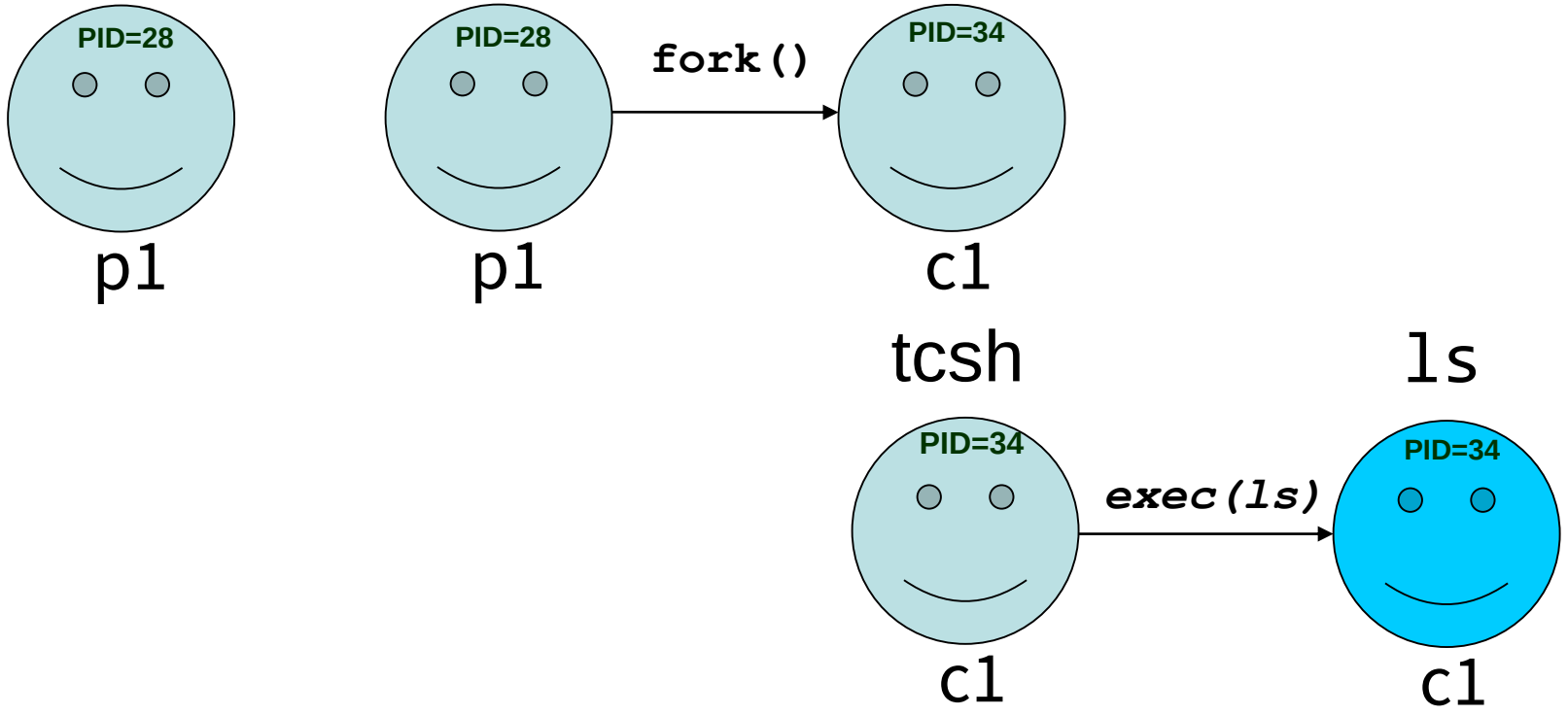
Illustration

After `exec()` system call:

Image will be replaced with `/bin/ls`

`fork()` and `exec()` together

- Often after doing **`fork()`** we want to load a new program into the child.
- Most common e.g. a shell programs



Example of forking separate processes

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

`wait()` Call System

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate* is more appropriate terminology, but a bit less common).

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

- The status, if not NULL, stores exit information of the child, which can be analyzed by the parent.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

Example

```
void main(void)
{
    printf("Before fork\n");
    int p = fork();
    if(p < 0) { /* Failed to fork */
        exit(0);
    } else if(p == 0) { /* Child process */
        printf("p = %d\n", p);
    } else { /* Parent process */
        wait(NULL);
    }
}
```

`exit()` System Call

- Gracefully terminates process execution, meaning it does clean up and release of resources, and puts the process into the ***zombie*** state

```
void exit(int status);
```

- By calling **`wait()`**, the parent cleans up all its zombie children.

- **`exit()`** specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.

Example of wait() and exit()

```
#include <stdio.h>
#include <stdlib.h>

main()
{
  int pid; int rv;

  pid=fork();
  switch(pid){
    case -1:
      printf("Error -- Something went wrong with fork()\n");
      exit(1); // parent exits
    case 0:
      printf("CHILD: This is the child process!\n");
      printf("CHILD: My PID is %d\n", getpid());
      printf("CHILD: Enter my exit status: ");
      scanf(" %d", &rv);
      printf("CHILD: I'm outta here!\n");
      exit(rv);
    default:
      printf("PARENT: This is the parent process!\n");
      printf("PARENT: My child's PID is %d\n", pid);
      printf("PARENT: I'm now waiting for my child to exit()...\n");
      wait(&rv);
      printf("PARENT: I'm outta here!\n");
  }
}
```

More about `wait()` and `exit()`

- Should not interpret the status value of system call **`wait(&status)`** literally. If **`&status`** is not NULL, **`wait()`** stores status information in the **`int`** to which it points.
- Value returned by **`exit(&status)`** is moved to 2nd byte and 1st (lowest) byte is used to store the status information.
- In previous example:

```
    scanf(" %d", &rv); // if value of x is entered
    ...
    exit(rv);
...
    wait(&rv); // the rv contents will be x left shift
               // by 8 bits and additional status
               // written into lowest 8 bit
```

Next lecture

- Process management in the operating system