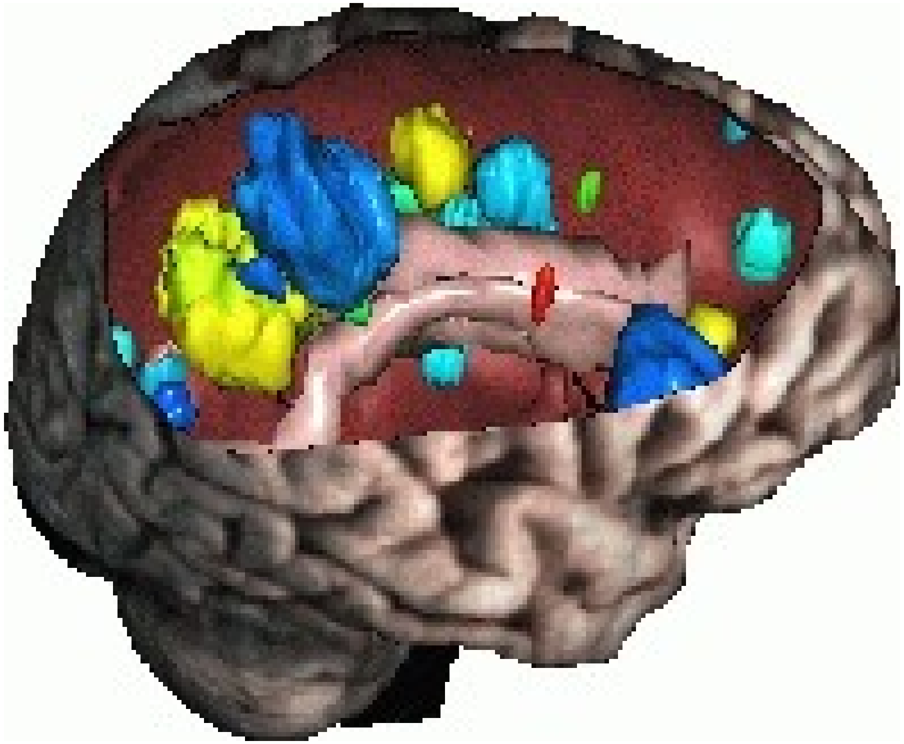
THE UNIVERSITY OF SYDNEY

Processes



"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. The other way is to make it so complicated that there are no obvious deficiencies."

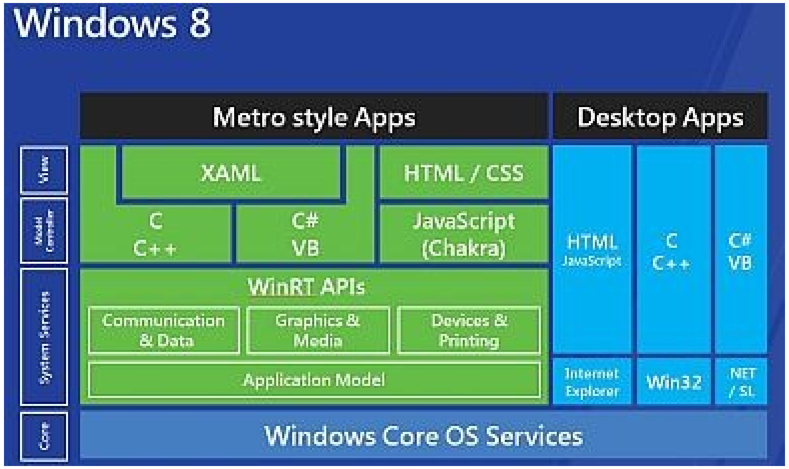
C.A.R.Hoare

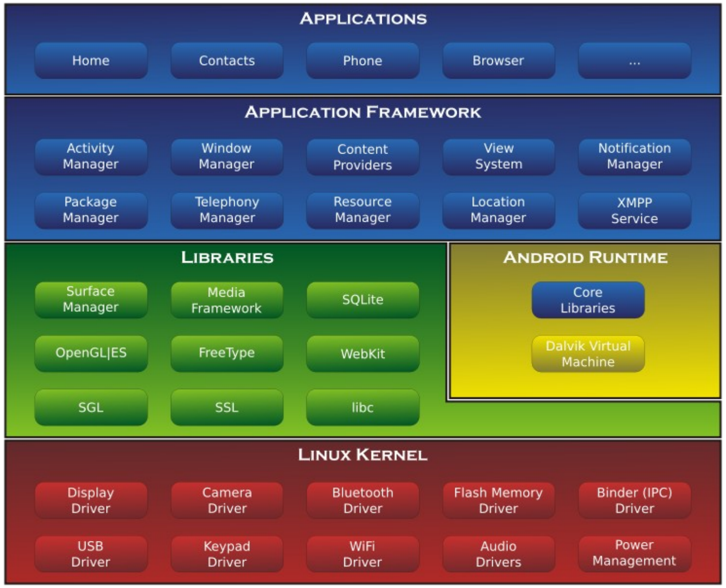
Processes and Memory

- Operating System (OS) only purpose is make the software run on the hardware

- OS is an overhead cost. Everything that the OS does requires resources from the hardware: memory, computation

- OS is a necessary abstraction for program writers so they don’t need to know hardware details





Processes and Memory

- OS manages all the memory for processes (execution state of a program), devices and communication (interrupts).

- OS will computationally solve many problems (search tasks) that programmer doesn’t have to worry about.

- To do this, OS also needs to have additional memory for each program and for itself.

- Memory contains both ***data*** and ***instructions*** (binary code).

Processes and Memory

- System memory is divided into two spaces
- Kernel
  - only processes with certain privileges can r/w/x
  - OS functions and data live here e.g. I/O, processes, devices
  - protect the hardware by only accessing through this layer

- User
  - all user created processes privilege depends on who created
  - These programs are treated as rogue/untrusted that can run and die
  - processes in this space are independent

- User space processes use *system calls* to access Kernel space *

Processes and Memory

- User creating a new process
  - OS creates a new image of memory that will be used by the process by cloning an existing process.
  - This has permissions associated with r/w/x of user/group

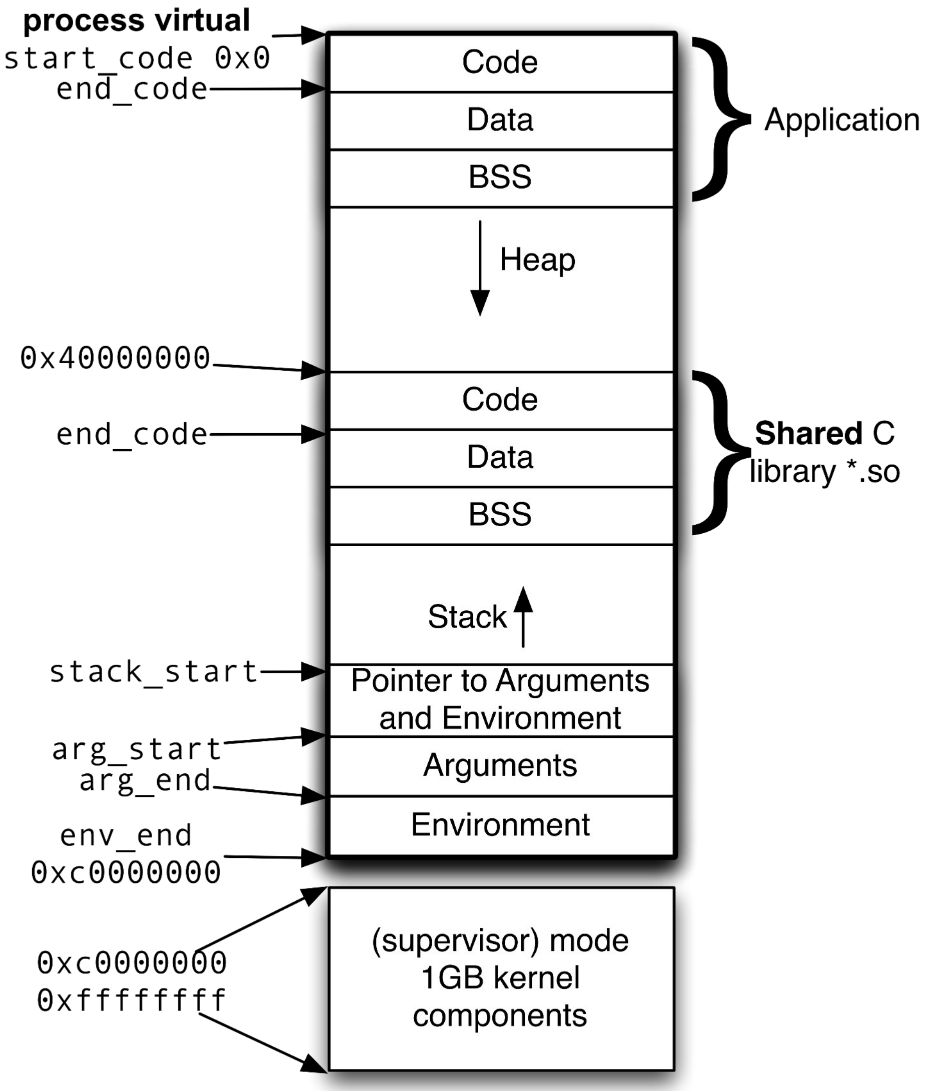
- The memory inside the process is assigned a virtual address range
  - Pieces of virtual memory get mapped to physical memory when they are needed during execution

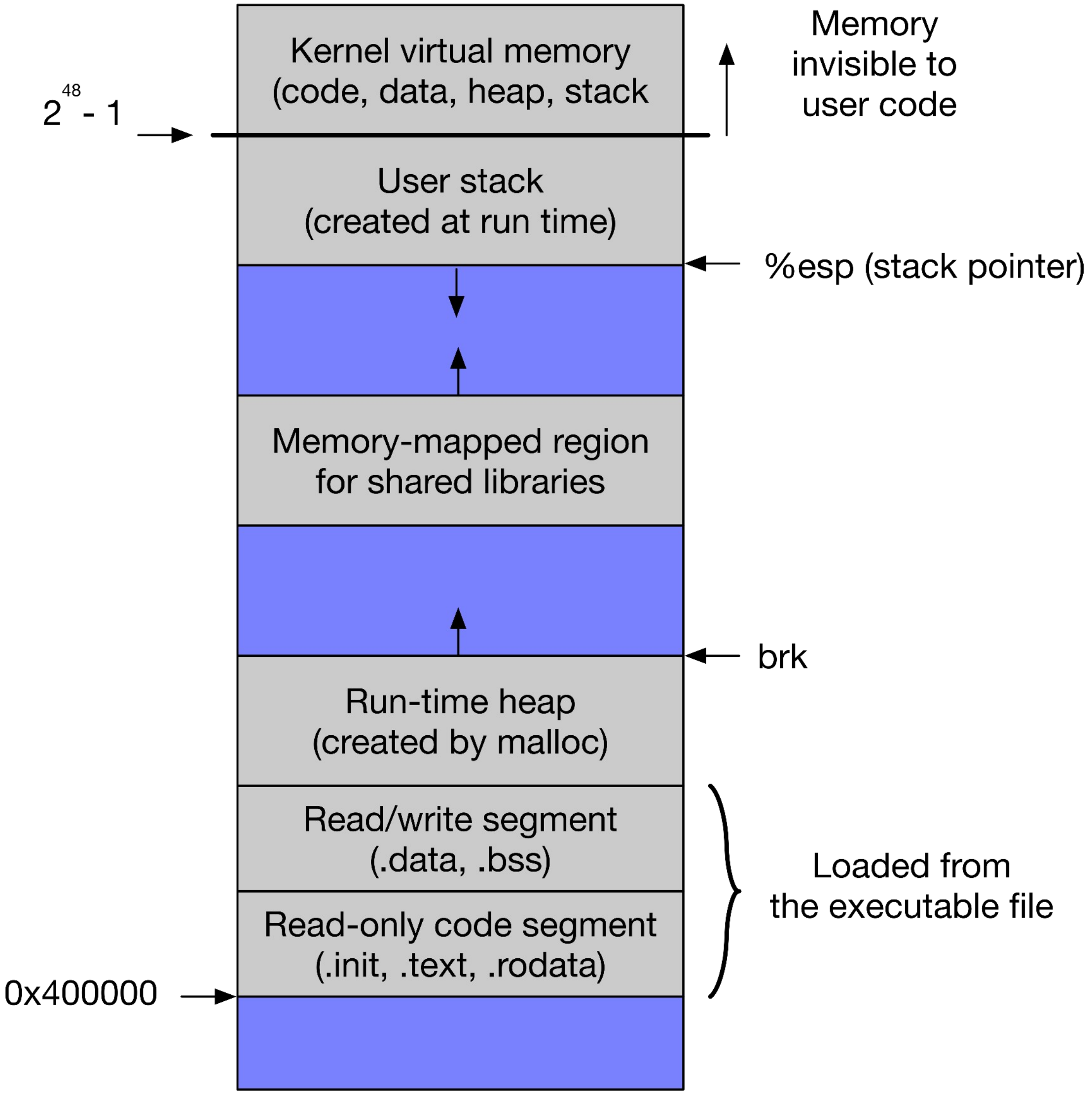
Processes and Memory

- Memory of a process is divided into several parts

- A process can potentially have more memory than system supports
  - Large virtual memory

- Size of a new process’ virtual memory address space varies with OS

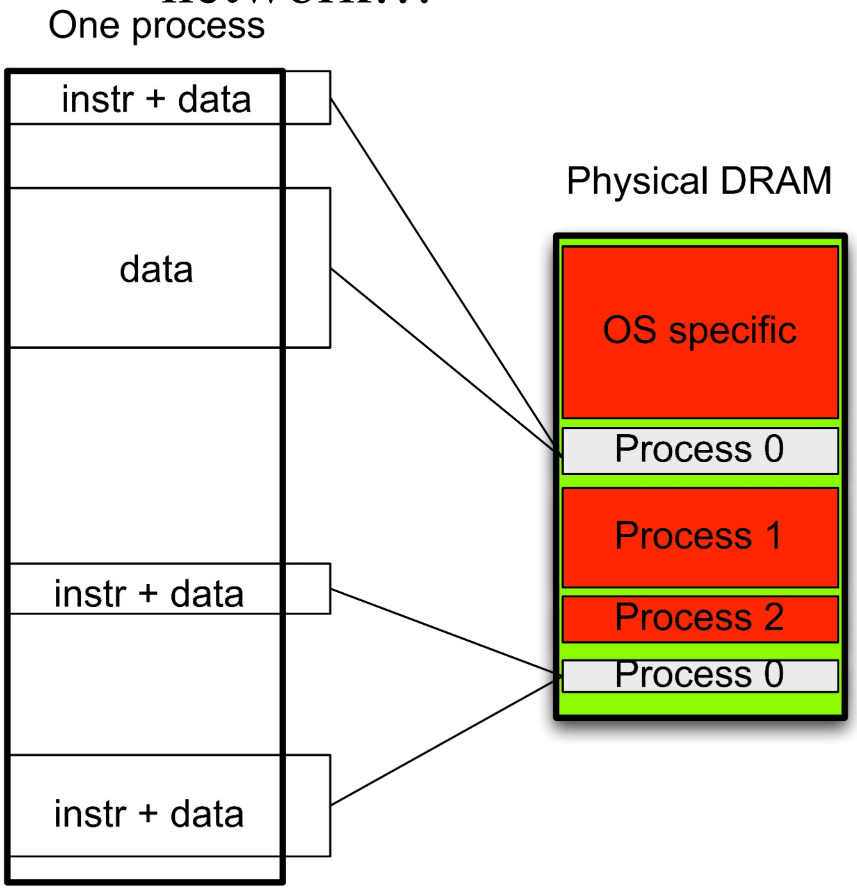




There is no strict format for the layout of a process

Processes and Memory

- Virtual memory of a process is mapped to physical memory.
- Physical memory is mixed: cache, RAM, disk, tape, network…



- Multiple processes share the same finite memory resources
  - The physical primary memory (DRAM/SRAM) is easily exhausted
- Whatever cannot fit is stored in secondary memory
  - OS does this management of virtual memory translation to physical memory (with some hardware help)

Initiating Processes

- the standard C library includes functions that invoke Unix *system calls*
- a set of these functions allows you to initiate and manage the running of other programs or *processes*
- the shell uses these functions to start the programs that correspond to the commands you type or put into a script

The main function

- when a program is started the main function is called

```
int main(int argc, char *argv[], char *envp[])
```

  - argc is the number of arguments passed
  - argv is an array of pointers to strings containing the arguments
  - envp is an array of pointers to strings containing the environment variables

Starting a program

- when the shell starts a command such as:

```
echo testing
```

it calls the main function with the arguments:

argc = 2

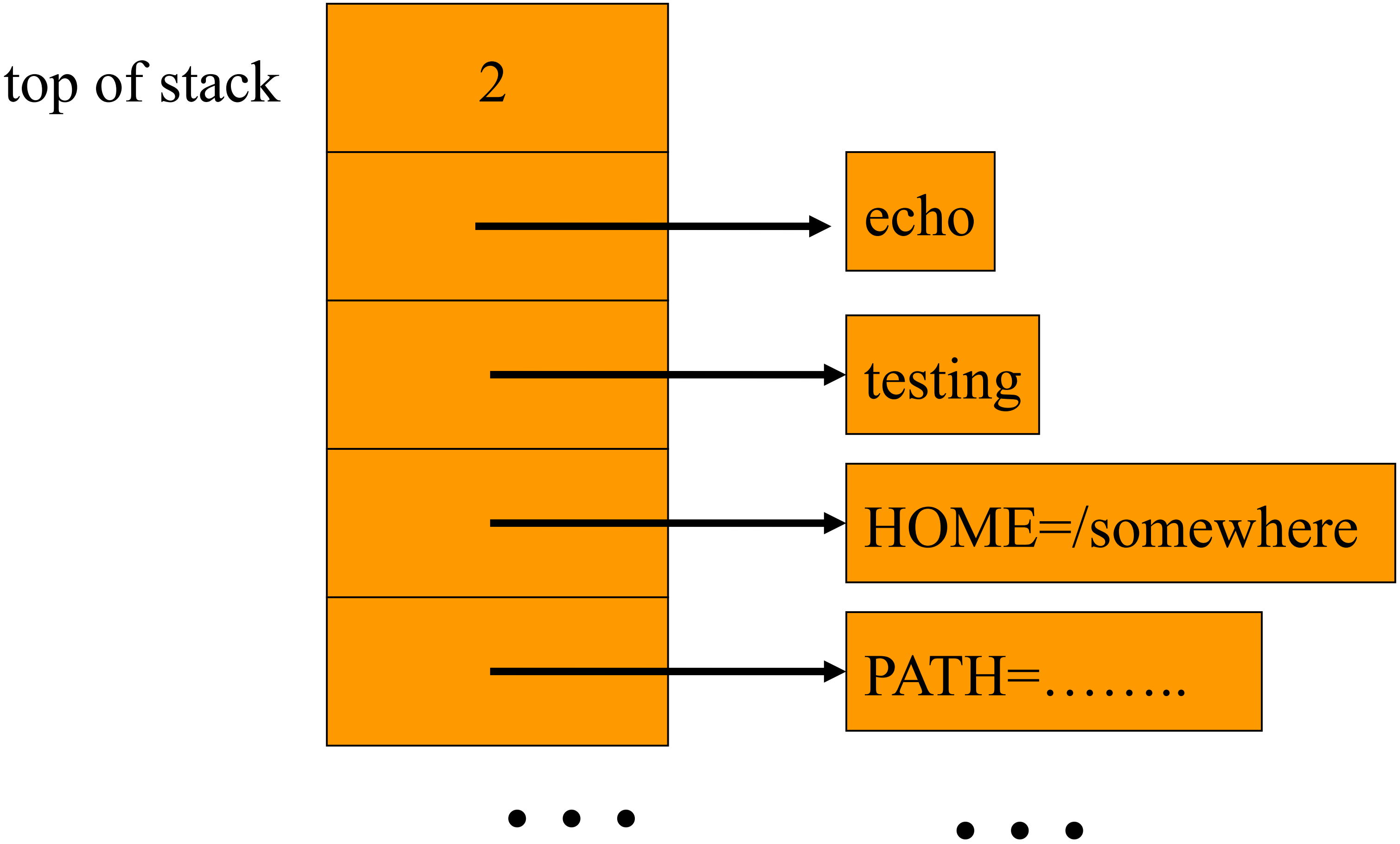
argv[0] = “echo”

argv[1] = “testing”

envp[0] = “*VARNAME=value*”

envp[1] = …

On the stack:



Initiating processes

- the following functions will start another process: execl, execle, execv, execve
- eg:

```
int execl(const char *path,
             const char *arg,
            const char *arg,…
            (char *)0)
```

signifies the end of the list of pointers to arguments

- exec in its various forms switches the program execution to another program
- your program is terminated and the other program’s main function is called
- if exec is successful it doesn’t return
- if it does return, and the return result is negative, then the program was not found
- if it returns zero or greater, then the exec function itself has failed!

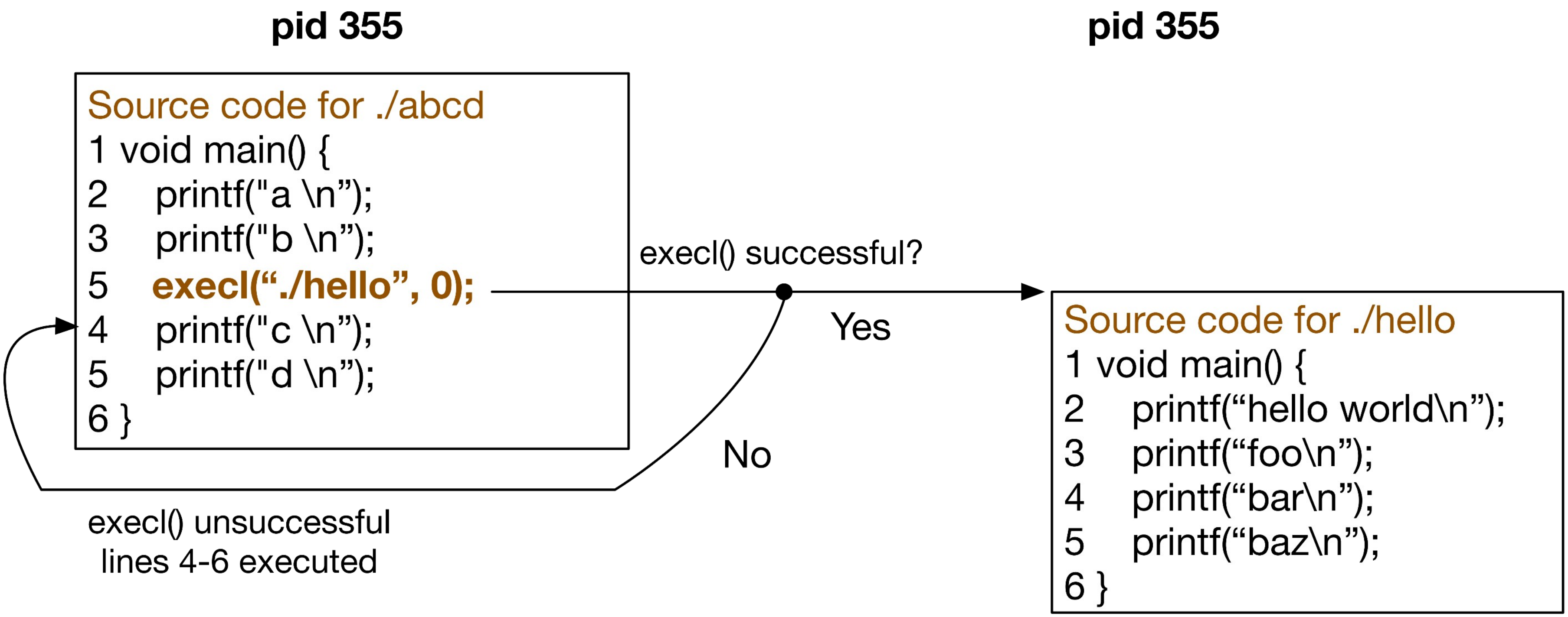
Example

```
if(execl("/bin/sort", "sort", "myfile", (char *)0) == -1)
{
		perror(argv[0]);
		exit(1);
}
/*program should never reach this point*/
```

system function for printing error messages after a system call error

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
	if( execl("/usr/bin/sort", "sort", "words.txt", (char *)0) == -1) {
		perror(argv[0]);
		exit(1);
	}
	return 0;
}
```

exec does not create a new process



pid 355

```
Source code for ./abcd
1 void main() {
2     printf("a \n”);
3     printf("b \n”);
5     execl(“./hello”, 0);
4     printf("c \n”);
5     printf("d \n”);
6 }
```

execl() unsuccessful
lines 4-6 executed

pid 355

```
Source code for ./hello
1 void main() {
2     printf(“hello world\n”);
3     printf(“foo\n”);
4     printf(“bar\n”);
5     printf(“baz\n”);
6 }
```

Parallel execution

- exec is like a GOTO
  - it jumps to another program and doesn’t return
- it is possible to start another program but still continue to execute using the *fork* function

Fork function

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

```
pid_t fork(void);
```

- creates a *child* process that is a copy of the memory image of the parent

Fork function

- **both** the parent and the child programs run in parallel
- the return value from the fork function is different for the parent and the child
- fork returns:
  - 0 in the child process
  - the *process id* of the child in the parent process
  - -1 in the parent process if the fork failed

Fork function

- by checking the return value of the fork function the running program can determine if it is the parent or the child

Who am I?

```
…
if (!(result = fork()))
{
/* child in control */
        …
}
/* parent in control */
if (result < 0)
{
   printf(“fork failed”);
   exit(1);
}
…
```

```
…
if (!(result = fork()))
{
/* child in control */
        …
}
/* parent in control */
if (result < 0)
{
   printf(“fork failed”);
   exit(1);
}
…
```

- usually, the child process will then use one of the exec functions to start a new program
- the parent continues to execute
- the parent can ignore the child or wait for it to exit
- there are Unix system functions that allow parents to control the child

Wait function

- the parent can wait until the child exits and get the exit value

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

waits for any child process to exit

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

waits for a ***specific*** process to exit

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

Wait function

- the wait function returns the process id of the child process
- the exit value of the child can be extracted from the status value
- other information in the status value indicates if the child failed or was terminated (rather than terminated normally)

Security implications

- fork bomb
- sharing resources between parent/child
- scheduling expectations
- zombie processes

Summary

- Processes are an abstraction for the OS
- the exec system call functions allow you to start another program running but the parent is terminated
- the fork system call function will make a copy of a process and both parent and child processes will continue to execute
- the wait system call function allows a parent process to wait for a child process to exit

- picture acknowledgement:

  http://www-sop.inria.fr/epidaure/research.php