Process Management

KERNEL MASTERS

Process Management

- **Process Concept**
  - Process ID & State
  - Process Tree
  - Process Control Block
  - Context Switching
  - Queues
  - Viewing Processes
- **Process Scheduling**
- **Process Creation**
  - Replacing Process Image
  - Replication of Processes
  - Waiting for Processes
  - Process Termination

Process Concept

- Like files, a process is a fundamental abstraction in Unix/Linux
  - An executing instance of a program
- A process is an "an address space with one or more threads executing within that address space, and the required system resources for those threads."
- The Linux kernel, supporting both pre-emptive multitasking and virtual memory, provides a process both a virtualized processor and a virtualized view of memory.
- Each process consists of one or more ***threads*** of execution
- A **thread** is the unit of activity within a process, the abstraction responsible for executing code.
- Each thread has
  - an id (*pid*)
  - a stack
  - state
  - program counter

Process ID (PID)

- Each process has a unique identifier, the *process ID* (maximum 32768)
- The process ID is represented by the pid_t type, defined in *<sys/types.h>*
- The **getpid( )** system call returns the process ID of the invoking process
- The **getppid( )** system call returns the ID of the parent of the invoking process.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
printf ("My pid=%d\n", getpid ( ));
printf ("Parent's pid=%d\n", getppid ( ));
return (0);
}
```

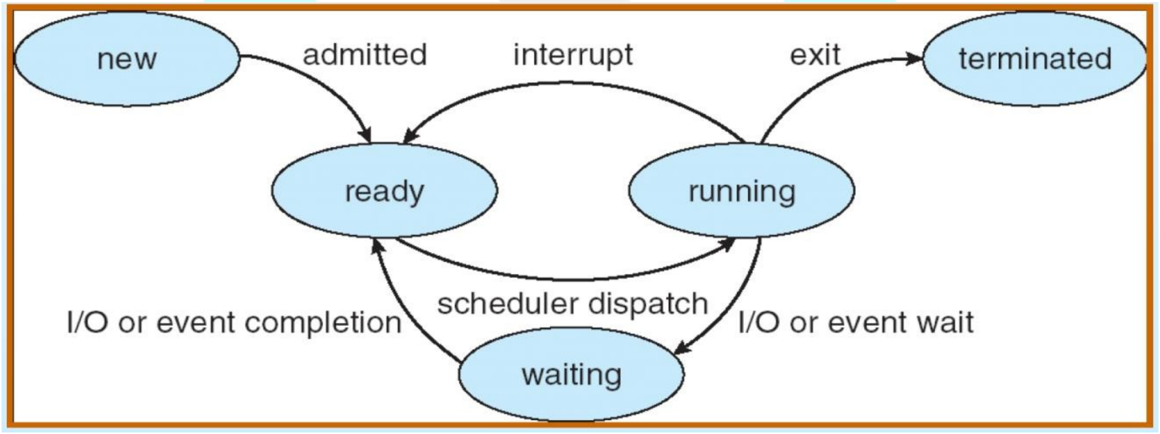
Results:

My pid=6811
Parents pid=6723

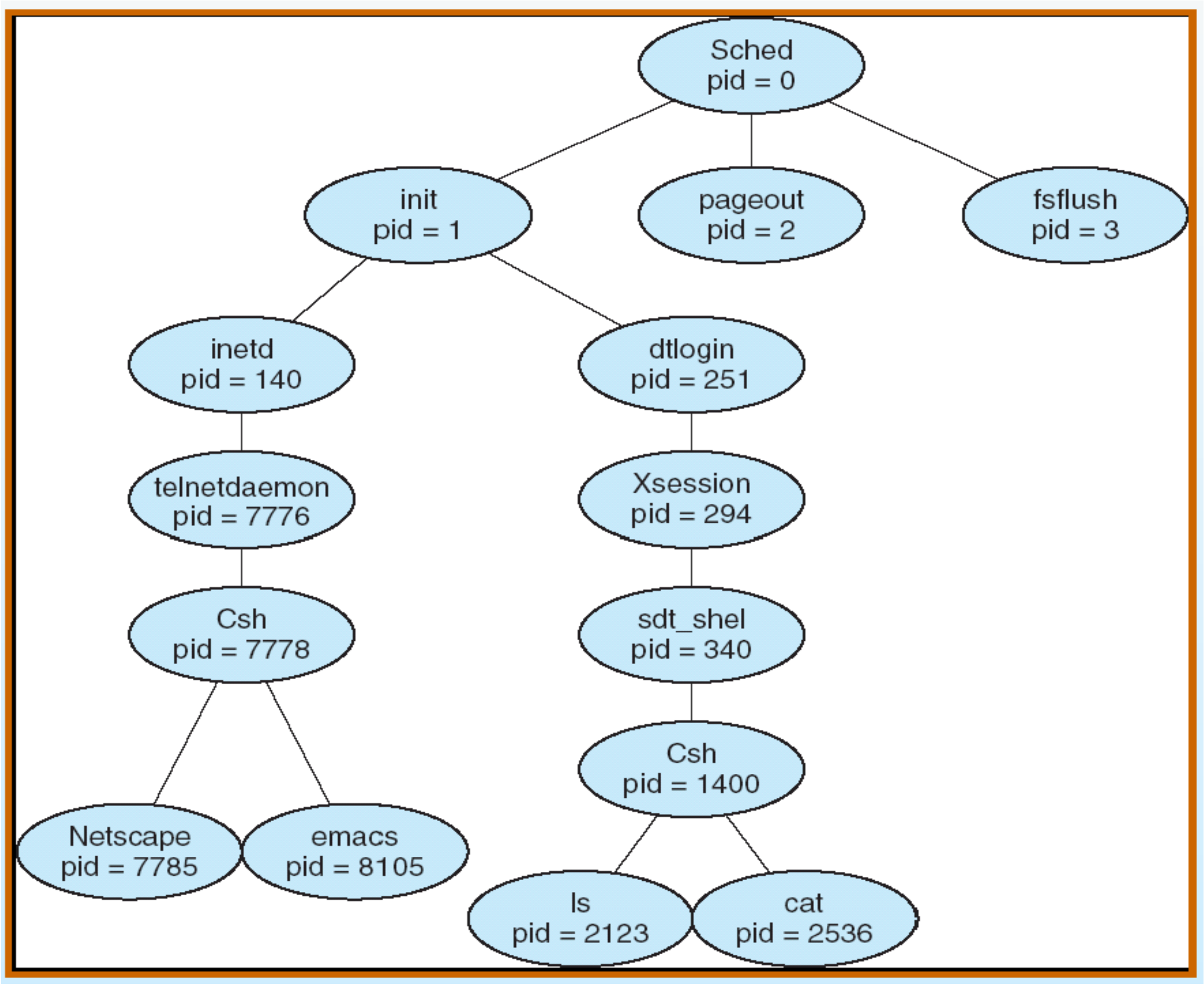
KERNEL MASTERS

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
- State values: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBL TASK_STOPPED, TASK_ZOMBIE



A tree of process



KERNEL MASTERS

Init Process

- The first process that the kernel executes after booting the system, called the *init process*, has the pid 1
- The init process handles
  - The remainder of the boot process
  - Initializing the system
  - Starting various services
  - Launching a login program
- The Linux kernel tries four executables, in the following order:
  - */sbin/init*: The preferred and most likely location for the init process.
  - */etc/init*: Another likely location for the init process.
  - */bin/init*: A possible location for the init process.
  - */bin/sh*: The Bourne shell, if it fails to find an init process

Process Control Block (PCB)

- Information associated with each process stored in a blo
memory known as **PCB or Process Descriptor.**
  - Process ID
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

Process Descriptor

>The kernel stores list of processes in a circular doubly list called tak_list.

>each element in the task list is pd of process

>task_struct is large data structure(1.7kb in 32 arch) defined at <linux/sched.h>

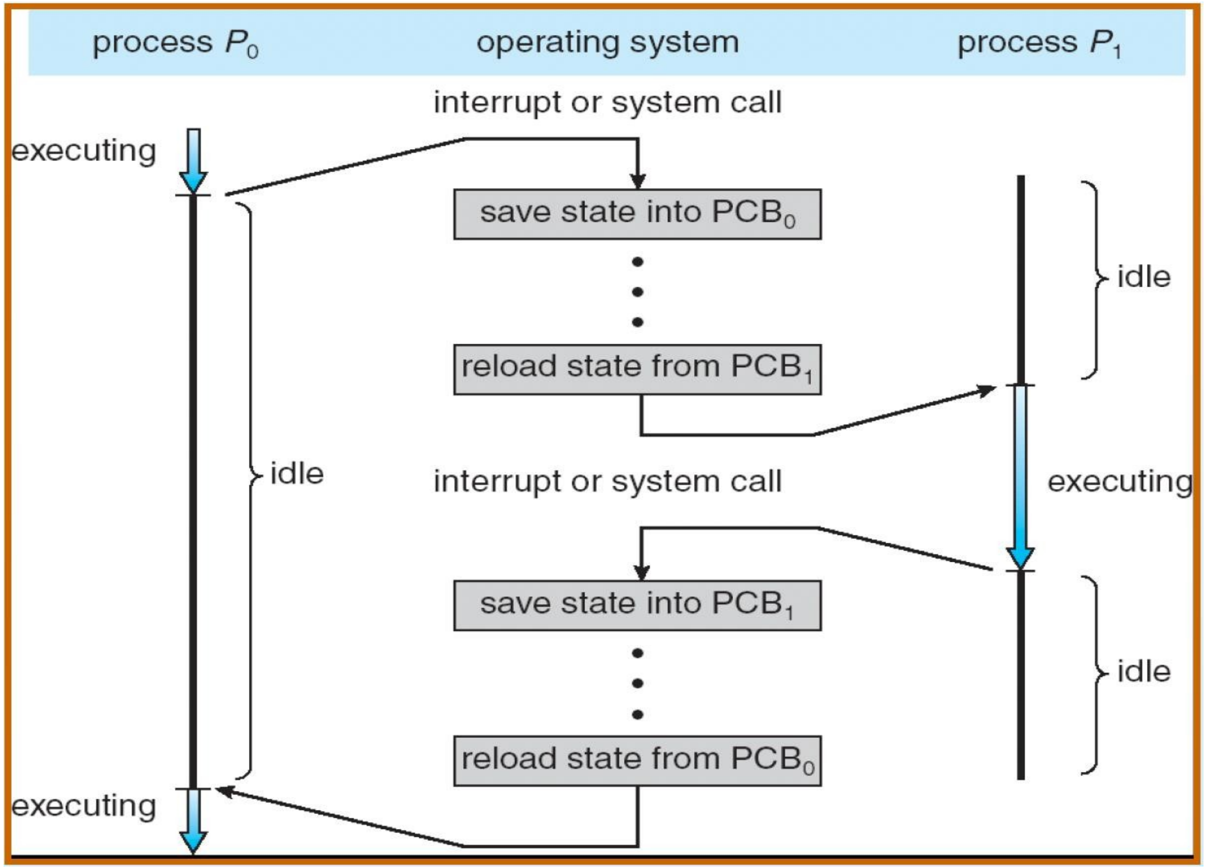
>inside the kernel, tasks are typically referenced directly by a pointer to their “task_struct structure”.

>Current macro: it is useful to be able to quickly look up the

process descriptor of the currently executing task.

KERNEL MASTERS

CPU Switch From Process to Proces



KERNEL MASTERS

Viewing Processes

- Linux Process Table
  - a data structure describing all of the processes that are currently loaded
- Viewing processes
  - The **ps** command shows the processes in the system or belonging to a user

$ **ps -af**

UID	PID	PPID	C	STIME	TTY	TIME	CMD
Root	433	425	0	18:12	tty1	00:00:00	[bash]

Process priority

$ **ps -l**

F S UID PID PPID C PRI NI SZ WCHAN TTY TIME CMD

000 S 500 1362 1262 2 80 0 789 schedu pts/1 00:00:00 oclock

KERNEL MASTERS

Replacing a Process image

execve()

Replacing a Process image

exec function replaces the current process with a new process specified by the path or file argument

```
int execl (const char *path, const char *arg0, ...., (char *)0);
int execlp (const char *file, const char *arg0, ...., (char *)0);
int execle (const char *path, const char *arg0, ...., (char *)0, char *const envp[]);
int execv (const char *path, char *const argv[]); //basic syscall
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
```

"l" indicates that the arguments are provided in a null terminated list; "v" in an array (vector);

"p" indicates the full PATH must be searched for the file;

"e" indicates a new environment is also supplied for the new process

ret = execl("/bin/ps", "ps", "-ax", 0); /* assumes ps is in /bin */

- replaces the current process image by loading the program pointed at by path

ret = execlp("ps", "ps", "-ax", 0); /* assumes /bin is in PATH */

- To use the "v" or array option

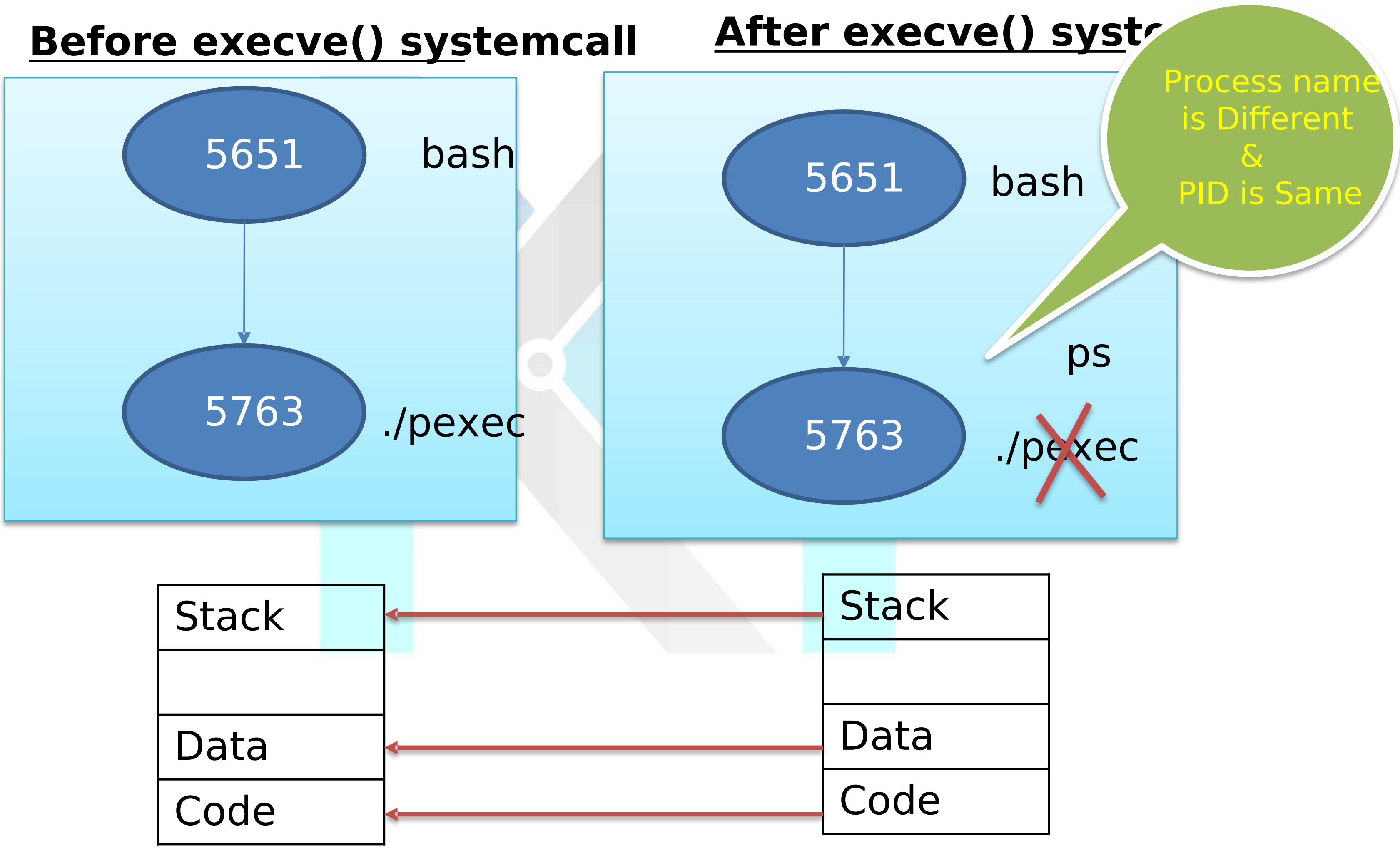
const char *args[ ] = { "ps", "-ax", NULL };

ret = execv ("/bin/ps", args); or **ret = execvp ("ps", args);**

KERNEL MASTERS

Before execve() systemcall

After execve() syste



“pexec” memory map

“ps” memory map

KERNEL MASTERS

exec call

- A successful invocation of exec call does not return; it ends by jumping to the entry point of the new program, and the just-executed code no longer exists in the process' address space
- On error execl() returns -1, and sets errno to indicate the problem (examples of errno values: EACCESS, ENOEXEC, ENOMEM, etc)

  Note: errno variable is defined in <errno.h> include fil
- On successful exec call
  - some properties of process are same: pid, priority, owning user and grou
  - some properties change: signals, memory locks, statistics
  - open files are retained; generally these are closed before the exec call

KERNEL MASTERS

How to create a process in Linux?

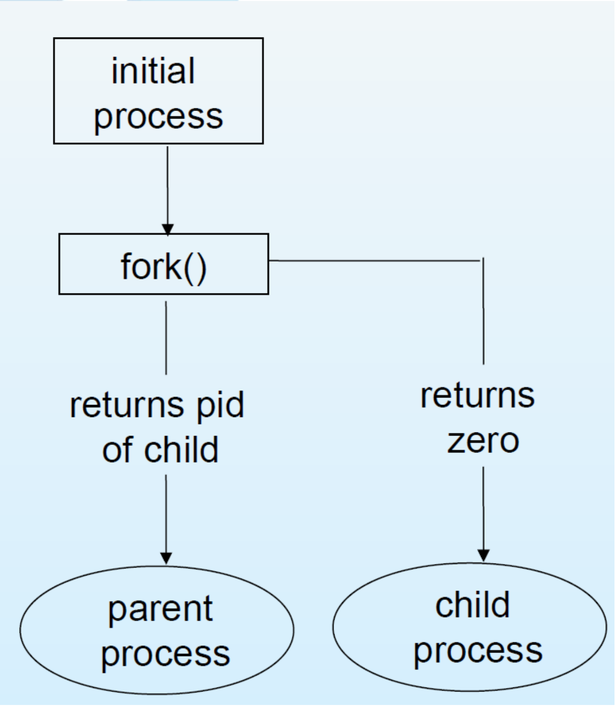
Approach 1:

New process is same as parent proces
with fork ()
(or) Duplicating a Process Image

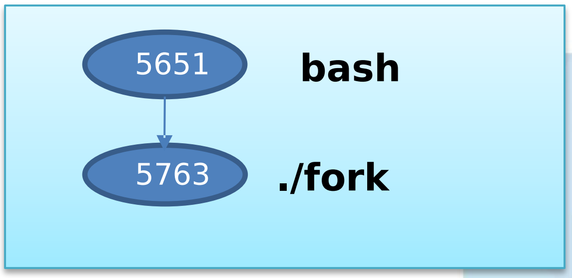
Duplicating a Process Image

- We can create a new process by calling **fork**. This system call duplicates current process (creates a new entry in the process table with same attributes as the current process)
- Both processes continue from next instruction.

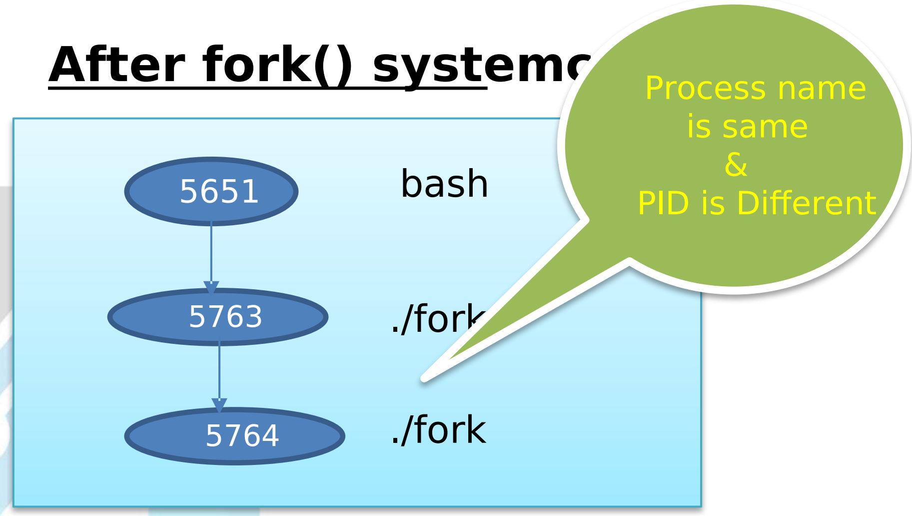
```
pid_t new_pid;
- - -
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
- - -
break;
default : /* We are parent */
- - -
break;
}
```

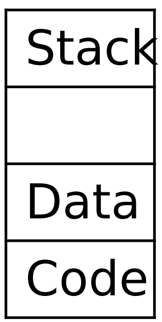


Before fork() systemcall

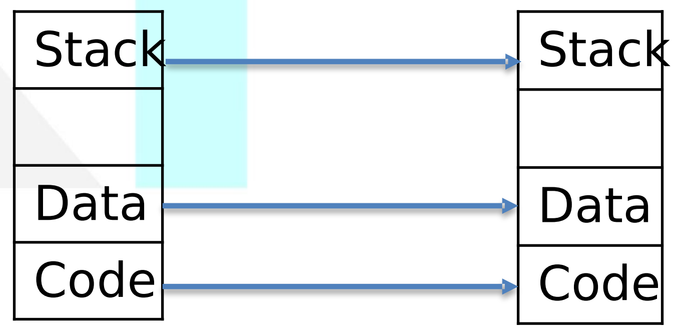


After fork() systemc





“fork” memory map



“Parent” memory map

“Children” memory map

fork call

- Fork creates a new process which is a copy of the calling process. That means that it copies the callers memory **(code, globals,heap and stack), registers and file descriptors.**
- **The successful fork() call**
  - The fork() call makes a copy of the parent process structure for the chil
    - Address space, resource limits, umask, controlling terminal, directory structu current working directory, file pointers etc
  - The following will be different
    - PID, PPID, resource utilizations (child set to 0), signals etc
- **On failure**
  - fork() returns – 1, and error set in errno (EAGAIN, ENOMEM)

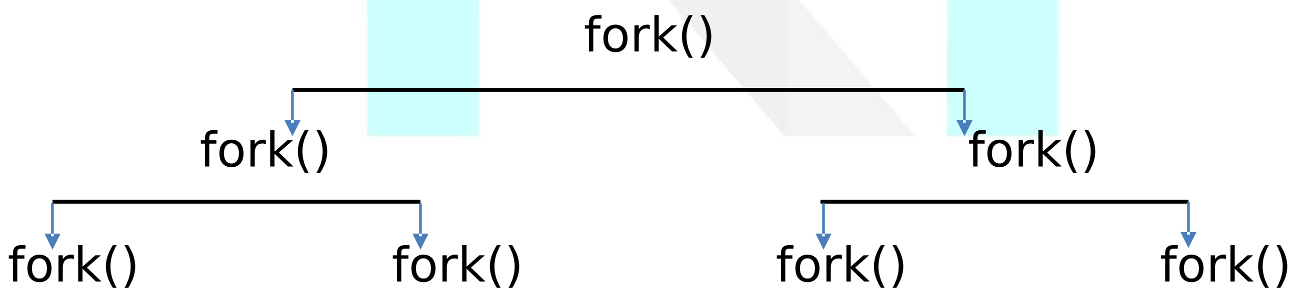
QUIZ

```
main()
{
      fork();
      fork();
      printf("Hello World");
}
```

Output:

Hello World
Hello World
Hello World
Hello World

How it Works?



How to create a process in Linux?

Approach 2:

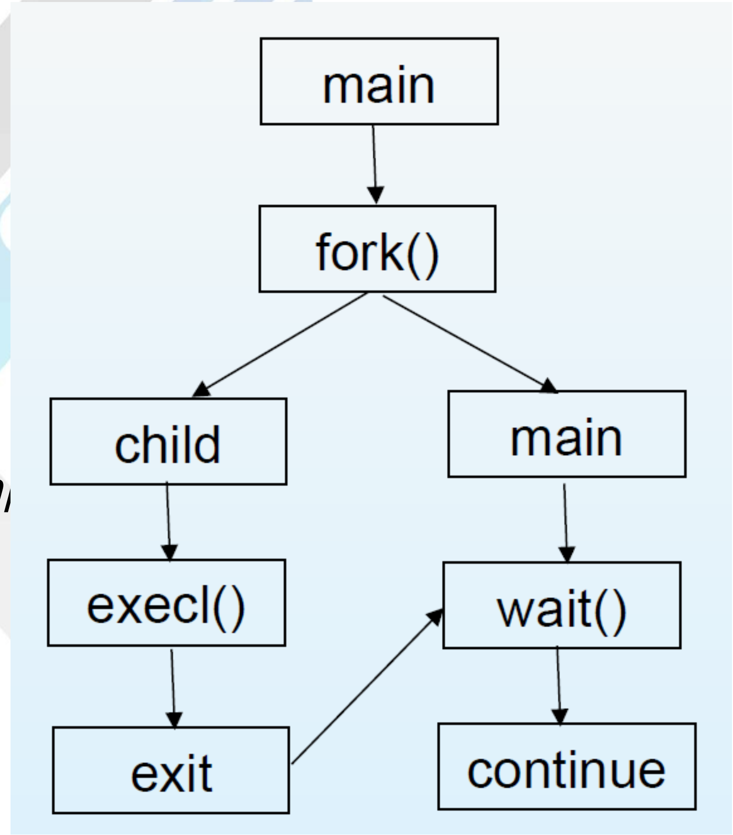
New process is **NOT** same as parent process with fork () & exec()

KERNEL MASTERS

fork and exec

```
#include <sys/wait.h>
int main()
{
pid_t pid;
/* fork another process */
pid = fork();

if (pid == 0) { /* child process */
execl ("/bin/ls", "ls", NULL);
}
else { /* parent process */
/* parent will wait for the child to com
*/
wait (NULL);
printf ("Child Complete");
exit(0);
}
}
```



KERNEL MASTERS

Waiting for a Process – wait()

- Parent process can wait for the child to finish by calling

pid_t wait (int *stat_val);

- The call returns PID & exit status of the child process in stat_val
- Need macros to interpret

  WIFEXITED(stat_val) – Nonzero if the child is terminated normally

  WEXITSTATUS(stat_val) – child exit code If WIFEXITED is nonzero

  WIFSIGNALED(stat_val) – Nonzero if child terminated on uncaught signal

  WTERMSIG(stat_val) – signal number if WIFSIGNALED is nonzero
- To wait for a specific process
- **pid_t waitpid (pid_t pid, int *status, int options);**
  - Options WNOHANG – Do not block

Process Termination

Process Termination

- Process executes last statement (**exit 0** for successful exit, **exit 1**, or >0 for exit with error condition) to inform the operating system to delete it.
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
- Parents may wait (via **wait**) for a child process to terminate
  - If a child process terminates before the parent does wait, Linux does notdeleteit fully but keeps the exit information for the parent (**zombie)**
- If a parent process exits
  - Some operating system do not allow child to continue if its parent terminates
    - All children terminated - *cascading termination*
  - In Linux, if a parent terminates before a child, the child is re-parented to another process in the group or to the init process
- The library call exit() is a wrapper over the kernel syscall _exit(). exit() flushes pending I/O, closes file descriptors and does other cleanup (memory, semaphores, etc) before calling _exit()

Process Termination

```
exit(n);
```

Exit status
Passed back to parent
0 means success
1-255 means failure

```
int status;
wait(&status);
```

Call waits until a child process terminates.
Returns PID of the child

The child's exit status is returned here.
Pass 0 (NULL) if not interested

KERNEL MASTERS

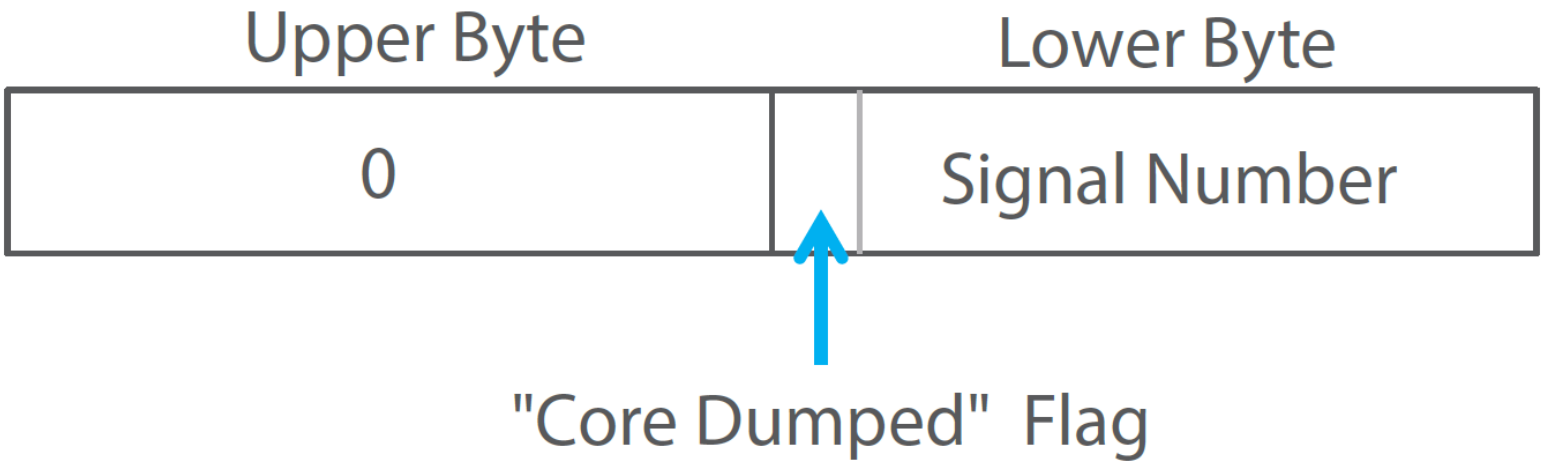
Exit Status – Normal Termination

Upper Byte	Lower Byte
Exit status (0-255)	0

Conventionally: zero = success, nonzero = "failure"

MACRO	Meaning
`WIFEXITED(status)`	True if child exited normally
`WEXITSTATUS(status)`	The exit status

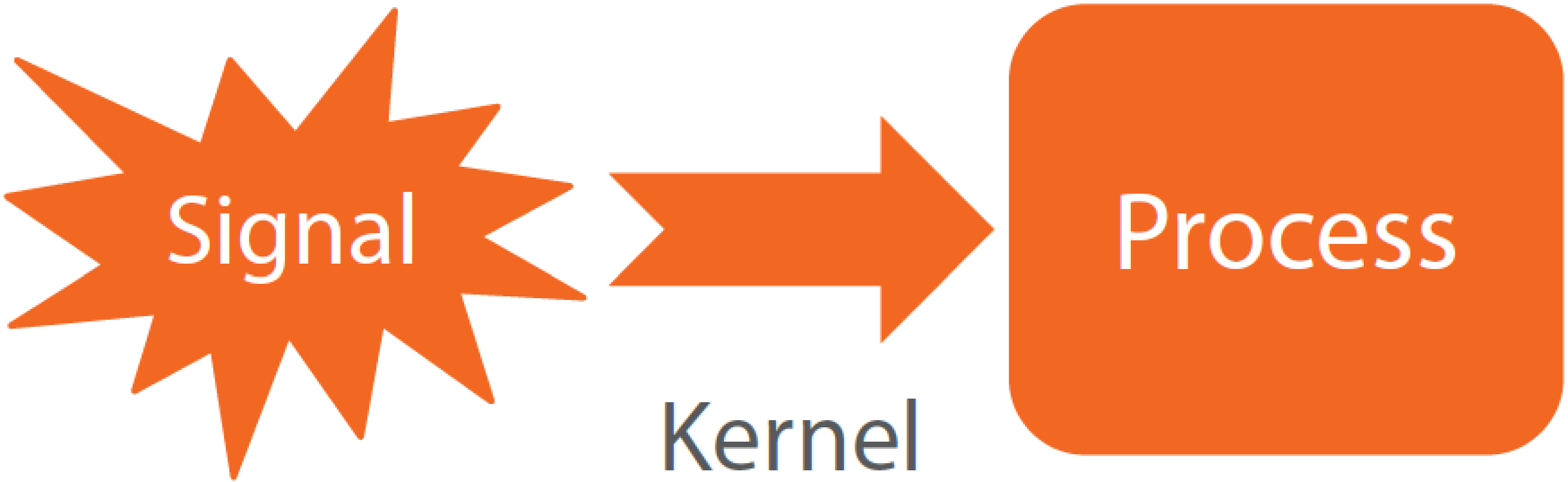
Exit Status – Killed by Signal



MACRO	Meaning
`WIFSIGNALED(status)`	True if child terminated by signal
`WTERMSIG(status)`	The signal number

Signals

What is a Signal?



KERNEL MASTERS

Signals

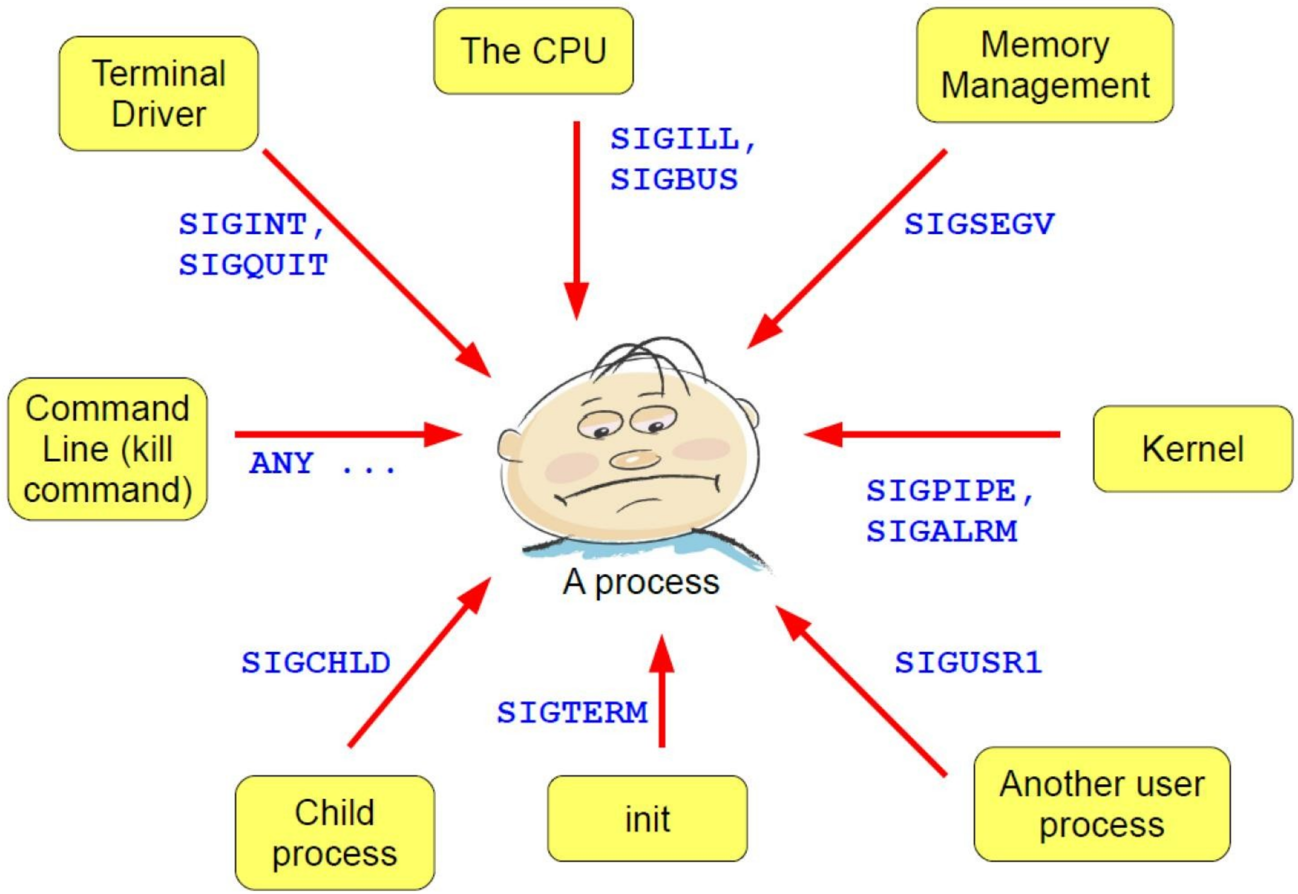
- *Signals* are software interrupts for handling asynchronous events
  - External – eg. the interrupt character (Ctrl-C)
  - Internal – as when the process divides by zero
  - A process can also send a signal (“raise”) to another process.

Action	Description
Term	Default action is to terminate the process.
Ign	Default action is to ignore the signal.
Core	Default action is to terminate the process and dump core
Stop	Default action is to stop the process.
Cont	Default action is to continue the process if it is currently stopped.

Signal Types

Signal Name	Number	Default Action	Description
SIGHUP	1	Term	Some daemons interpret this to mean "re-read your configuration file"
SIGINT	2	Term	The signal sent by ^C on terminal
SIGTRAP	5	Core	Trace/breakpoint trap
SIGFPE	8	Core	Arithmetic error, e.g. divide by zero
SIGKILL	9	Term	Lethal signal, cannot be caught or ignored
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Expiry of alarm clock timer
SIGTERM	15	Term	Polite "please terminate" signal
SIGCHLD	17	Ignore	Child process has terminated

Signals



Terminal Driver
The CPU
Memory Management
SIGILL,
SIGBUS
SIGINT,
SIGQUIT
SIGSEGV
Command Line (kill command)
ANY ...
Kernel
SIGPIPE,
SIGALRM
A process
SIGCHLD
SIGTERM
SIGUSR1
Child process
init
Another user process

Signals

- **Signal life cycle**
  - A signal is “raised”
  - Kernel stores and delivers the signal
  - The process handles the signal

- **Signal handling**
  - **SIGKILL & SIGSTOP** cannot be ignored.
  - Catch and handle the signal by registered functions (signal handlers)
    - SIGINT and SIGTERM are two commonly caught signals.
  - Default action – terminate the process (result in core dump)

Signals

The following system calls and library functions allow the caller to send a signal:

- **Sending a signal:**
  - raise(3) Sends a signal to the calling thread.
  - kill(2) Sends a signal to a specified process, to all members of a specified process group, or to all processes on the system.
- **Catching a signal:**
  - sigaction(2) or signal(2) process can change user defined signal.
- **Waiting for a signal**
  - pause(2) Suspends execution until any signal is caught.

Process Priority

Process Priority

- Unix has historically called process priorities *nice values*
  - Legal nice values range from –20 to 19 inclusive, (default value of 0) (t
    lower a process’ nice value, the higher its priority)
  - Linux provides system calls for retrieving and setting a process’ nice va
- **int nice (int inc);**
  - If inc = 0, nice returns current value
  - For inc > 0, nice increments the nice value by inc & returns the new va
- getpriority(), setpriority(), renice()
  - Get and set priority for individual process, group or user