1878

Western

UNIVERSITY · CANADA

Chapter 3C – Operations on Processes

Spring 2023

Operations on Processes

- Process Creation
- Process Termination
- Examples

Process Creation

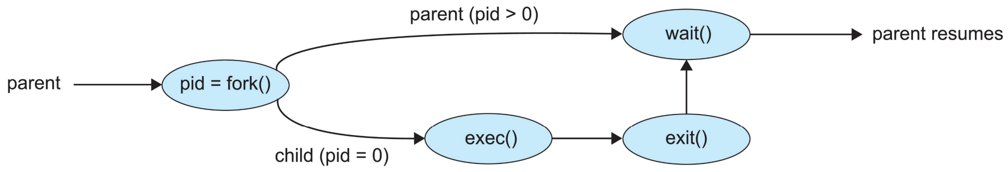
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)

Process Creation

- Resource sharing options
  - Parent and children share all resources mem, data, etc ....
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

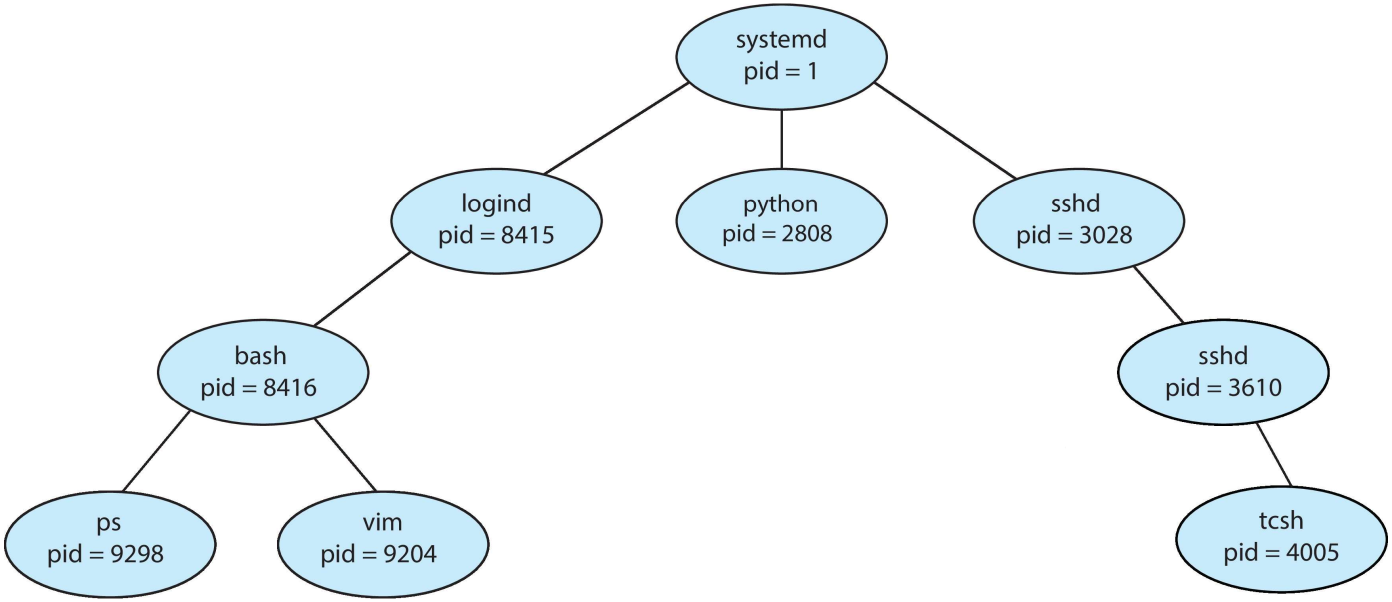
Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program
  - Parent process calls `wait()` waiting for the child to terminate



Process Creation

- A tree of processes in Linux



Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system

Process Termination

- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **Cascading termination**: All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process
  - `pid = wait(&status);`
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait()`, process is an **orphan**

Process Termination

- `pid = wait(&status);`
  is shorthand for:
  `pid = waitpid(-1, &wstatus, 0);`
  - `-1` – Wait for <u>any</u> child process
  - `wstatus` – Load the exit status into this integer
    - This can be `NULL` if we don't need the exit status
  - `0` – wait but depending on how the child returned. We will always just use 0

one spec child.

- `waitpid(pid,NULL,0);` – wait until child `pid` returns
- `while (wait(NULL)>0);` – wait for <u>all</u> child processes

Examples

- For all system and library calls, consult the man page (you may need to specify section 2 or 3)

-
  ```
  $ man man
  …
  1   Executable programs or shell commands
  2   System calls (functions provided by the kernel)
  3   Library calls (functions within program libraries)
  …
  ```

-
  ```
  $ man wait  ← The wait command built into bash
  ```

-
  ```
  $ man 2 wait ← The wait system call
  ```

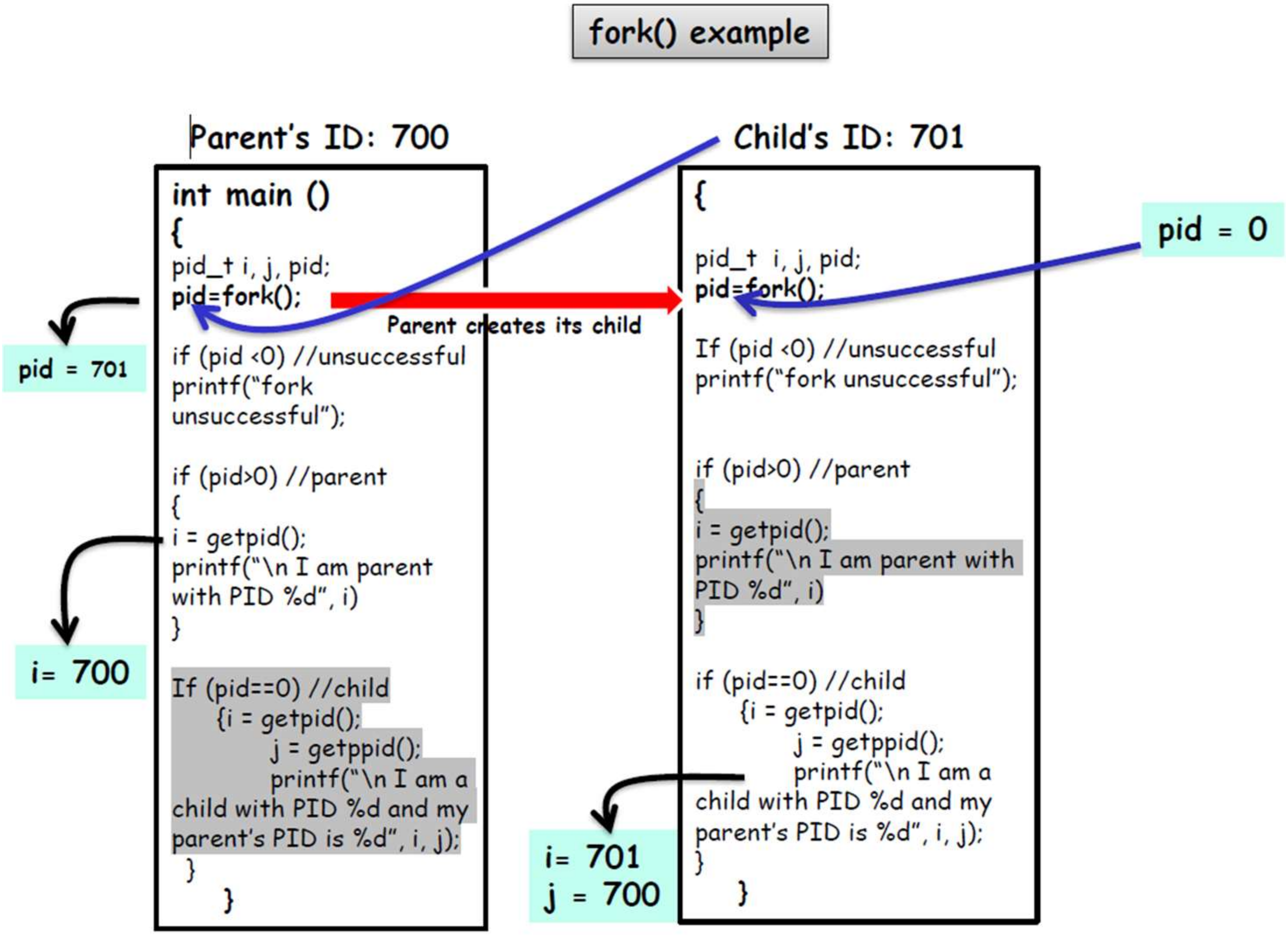
Examples

- The Unix system call for process creation is `fork()`
- The fork system call creates a child process that is a duplicate of the parent.
  - Child inherits state from parent process
    - Same program instructions, variables have the same values, same position in the code.
  - Parent and child have separate copies of that state
    - They are stored in separate locations in memory
    - Important: updating the value of a variable inside the child will NOT update that variable in the parent, and vice-versa.

Examples

- If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child
- If `fork()` fails, it returns -1 to the parent (no child is created)
- `pid_t` data type represents process identifiers
- Other calls:
  - `pid_t getpid()` – returns the PID of calling process. Call is always successful.
  - `pid_t getppid()` – returns the PID of parent process. Call is always successful.

Examples



```
int main ()
{
pid_t i, j, pid;
pid=fork();

if (pid <0) //unsuccessful
printf("fork
unsuccessful");

if (pid>0) //parent
{
i = getpid();
printf("\n I am parent
with PID %d", i)
}

If (pid==0) //child
     {i = getpid();
           j = getppid();
           printf("\n I am a
child with PID %d and my
parent's PID is %d", i, j);
 }
    }
```

```
{

pid_t  i, j, pid;
pid=fork();

If (pid <0) //unsuccessful
printf("fork unsuccessful");


if (pid>0) //parent
{
i = getpid();
printf("\n I am parent with
PID %d", i)
}

if (pid==0) //child
     {i = getpid();
           j = getppid();
           printf("\n I am a
child with PID %d and my
parent's PID is %d", i, j);
}
     }
```

Examples

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
  pid_t pid;
  int i;
  pid=fork();
  if( pid> 0 ) { /* parent */
      for( i=0; i< 10; i++ )
          printf("\t\t\tPARENT%d\n", i);
  }
  else { /* child */
      for( i=0; i< 10; i++ )
          printf( "CHILD %d\n", i);
  }
  return 0;
}
```

- What is the possible output?

```
                PARENT 0
                PARENT 1
                PARENT 2
                PARENT 3
                PARENT 4
                PARENT 5
                PARENT 6
                PARENT 7
                PARENT 8
                PARENT 9
        CHILD 0
        CHILD 1
        CHILD 2
        CHILD 3
        CHILD 4
        CHILD 5
        CHILD 6
        CHILD 7
        CHILD 8
        CHILD 9
```

Examples

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
  pid_t pid;
  int i;
  pid=fork();
  if( pid> 0 ) { /* parent */
      for( i=0; i< 10; i++ )
          printf("\t\t\tPARENT%d\n", i);
  }
  else { /* child */
      for( i=0; i< 10; i++ )
          printf( "CHILD %d\n", i);
  }
  return 0;
}
```

- What is the possible output?

```
        PARENT 0
        PARENT 1
        PARENT 2
        PARENT 3
        PARENT 4
        PARENT 5
        PARENT 6
    CHILD 0
    CHILD 1
    CHILD 2
        PARENT 7
        PARENT 8
        PARENT 9
    CHILD 3
    CHILD 4
    CHILD 5
    CHILD 6
    CHILD 7
    CHILD 8
    CHILD 9
```

Examples

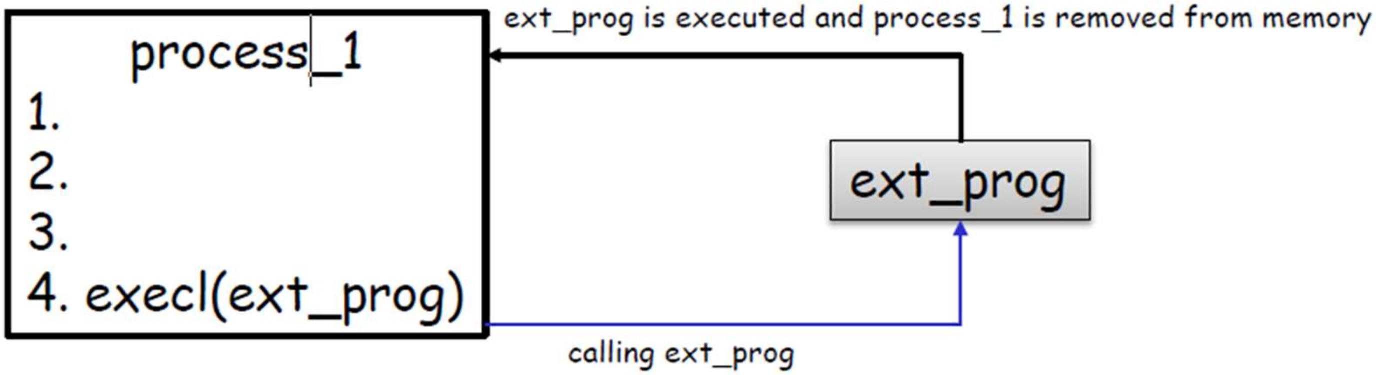
- Output is **nondeterministic** - Cannot determine output by looking at code
- Processes get a share of the CPU to give another process a turn
  - The switching between the parent and child depends on many factors: machine load, process scheduling

Examples

- The system call `execl()` replace a process (the caller process) with a new loaded program

- `execl()` loads a binary file into memory (destroying the memory image of the program calling it)

- On success, `execl()` **never** returns; on failure, `execl()` returns -1



Examples

- Program A

  ```
  int i= 5;
  printf("%d\n",i);
  execl("B", "", NULL);
  printf("%d\n",i);
  ```

- Program B

  ```
  main(){
    printf("hello\n");
  }
  ```

- What is the possible output?

  ```
  5
  hello
  ```

- Why not?

  ```
  5
  hello
  5
  ```

Examples

- fork() and execl()

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
  pid_tpid;
  pid= fork();
  if (pid> 0){
      wait(NULL);
      printf("Child Complete");
  }
  else{
      if (pid== 0) {
          execl("B", "", NULL);
      printf("\n You'll never see this line..");
  }
}
```

this is a child process .

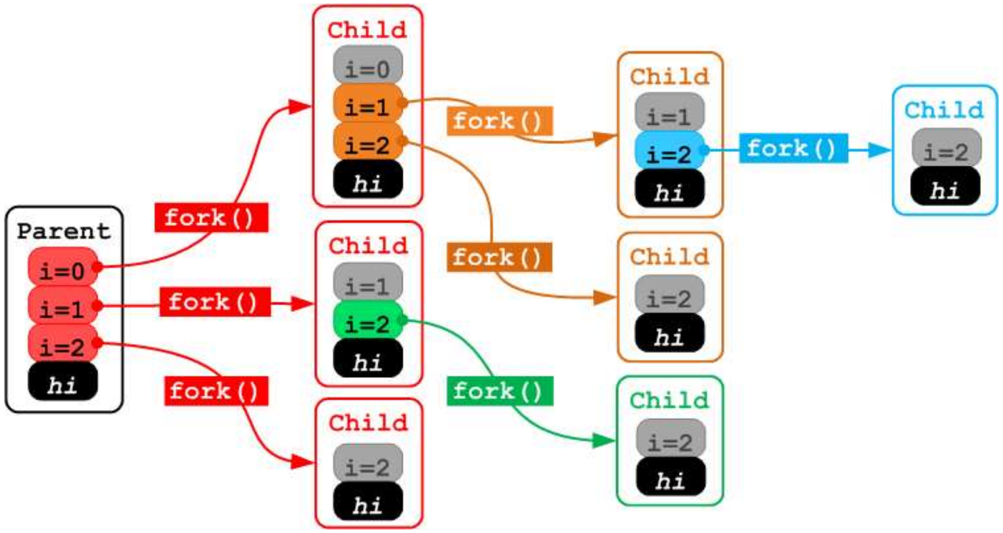
Examples

- How many processes are created by this program?

```
#include <stdio.h>
#include <unistd.h>
void main() {
  int i;
  for (i=0;i<3;i++){
      fork();
  }
  printf("hi\n");
}
```

Examples

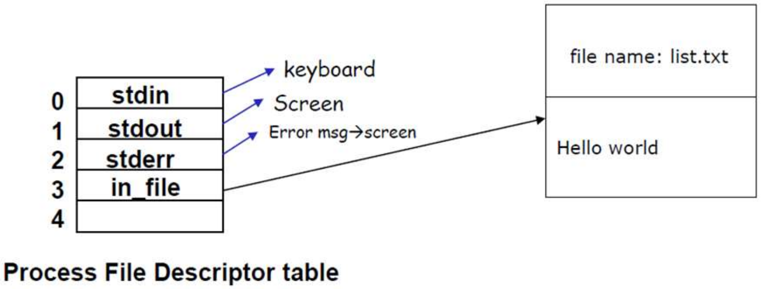
- https://stackoverflow.com/questions/26793402/visually-what-happens-to-fork-in-a-for-loop
- 2^n processes
- 2^n -1 children



Examples

- Forks and files
  - Every process has a **process file descriptor table**
  - Each entry in process file descriptor table represents **stdin**, **stdout**, **stderr**, and **file pointer**.
  - Assume that there was something like this in a program

    -
      ```
      int in_file;
      in_file= open("list.txt", O_RDONLY);
      ```



Examples

- Open a file before a fork
    - The child process gets a copy of its parent's process file descriptor table.
    - The child and parent share a file pointer because the open came before the fork.
- Open a file after a fork
    - Assume that parent and child each open a file after the fork
    - They get their own entries in the file descriptor table
        - This implies that the file position information is different
- Suppose that hello.txt consists of Hello, world!. What is the output of…
    - open() before fork()
    - open() after fork()

Examples

-
```
#include <stdio.h>
#include <unistd.h> //fork()
#include <fcntl.h>  //open() (unbuffered open)
#include <sys/wait.h> //wait()
int main(){
  int fd; char c; pid_t pid;
  fd=open("hello.txt", O_RDONLY);
  pid=fork(); //Open a file before a fork
  if (pid> 0){
      read(fd, &c, 1);
      printf("fd: %d, parent: c = %c\n", fd, c);
      wait(NULL);
  }
  else if (pid== 0){
      read(fd, &c, 1);
      printf("fd: %d, child: c = %c\n", fd, c);
      return 0;
  }
  fclose(fd);
}
```

- Output
  -
    ```
    fd: 3, parent: c = H
    fd: 3, child: c = e
    or
    fd: 3, child: c = H
    fd: 3, parent: c = e
    ```

Examples

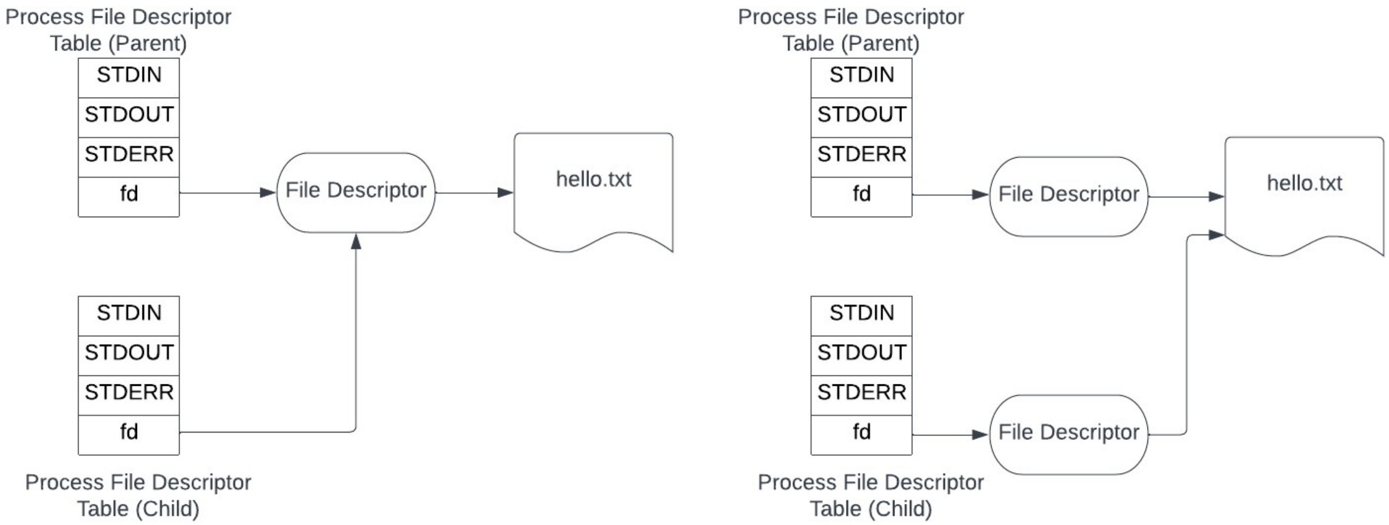
-
  ```
  #include <stdio.h>
  #include <unistd.h> //fork()
  #include <fcntl.h>  //open() (unbuffered open)
  #include <sys/wait.h> //wait()
  int main(){
    int fd; char c; pid_t pid;
    pid=fork(); //Open a file after a fork
    fd=open("hello.txt", O_RDONLY);
    if (pid> 0){
        read(fd, &c, 1);
        printf("fd: %d, parent: c = %c\n", fd, c);
        wait(NULL);
    }
    else if (pid== 0){
        read(fd, &c, 1);
        printf("fd: %d, child: c = %c\n", fd, c);
        return 0;
    }
    fclose(fd);
  }
  ```

- Output

  -
    ```
    fd: 3, parent: c = H
    fd: 3, child: c = H
    or
    fd: 3, child: c = H
    fd: 3, parent: c = H
    ```

Examples

- It is probably better to open files **after** a fork so the parent and child do not confuse each other


Process File Descriptor Table (Parent)
STDIN
STDOUT
STDERR
fd
File Descriptor
hello.txt
STDIN
STDOUT
STDERR
fd
Process File Descriptor Table (Child)
Process File Descriptor Table (Parent)
STDIN
STDOUT
STDERR
fd
File Descriptor
hello.txt
STDIN
STDOUT
STDERR
fd
File Descriptor
Process File Descriptor Table (Child)

1878

Western

UNIVERSITY · CANADA