Process

What is a Process

- A process is an execution-instance of a program[a].
- Running of a program requires a CPU, a memory, open files and other resources.
- The OS creates an illusion (virtual machine) that every process gets its own computing system.

[a]One program may have more than one execution instances running in parallel. (`$ g++ -Wall runParallel $ ./a.out & ./a.out, Ctrl+c, $ fg@, Ctrl+c`)

OS and Process

- A process is the logical unit of computation that the OS manages.
- A process may have several threads of computation, but for time being we assume that there is only one.
- OS allocates resource e.g. CPU time, memory space, IO facility to every process.

OS and Process

- OS creates a process.
- It loads the executable code and static data from a file present in a persistent memory e.g. disk.
- It creates other memory regions e.g. execution stack, heap etc. for running the code.

OS and Process

- It schedules a process ready for execution, suspends it if necessary, and finally terminates it (normal or abnormal).
- It receives request from a process for service and provides it if valid. The service may be an IO request, or a request for more memory etc.

OS and Process

- The OS protects one process from the interference of another process.
- It also insulates the overall system from the malfunction of a running process.
- At the same time it also facilitates communication between cooperating processes.

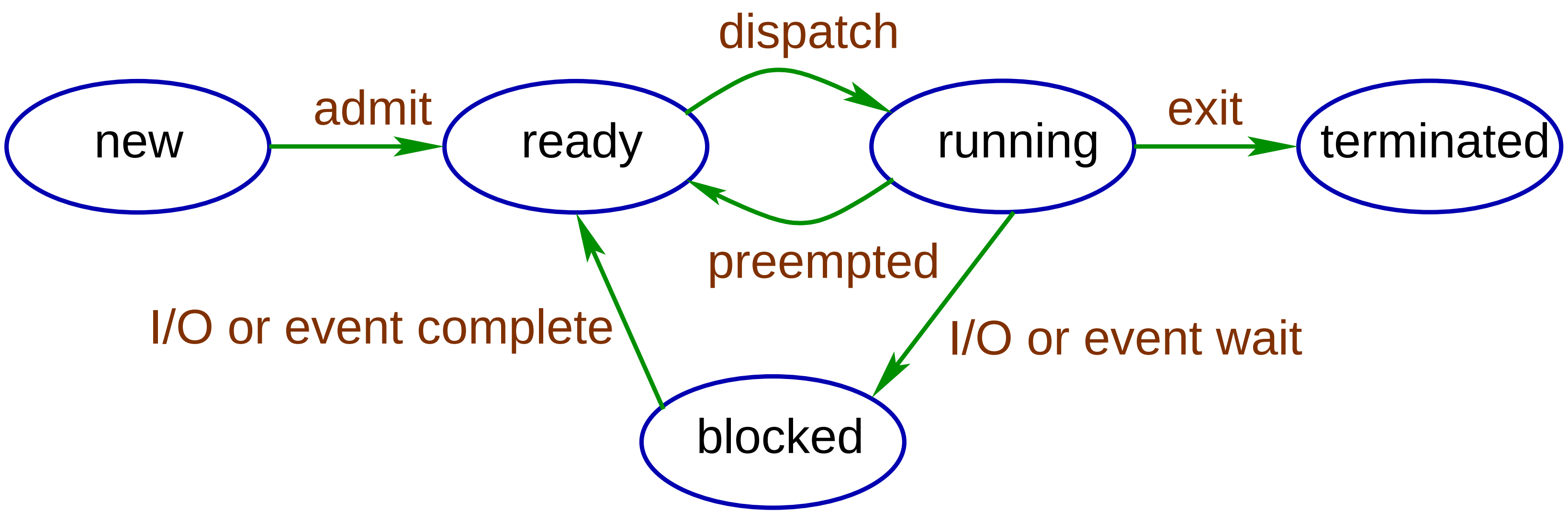
Process States

- In terms of internal computation a process may have large number of states depending on the content of the CPU, memory, open files, messages etc.
- But the OS is not concerned about these internal states.

Process States

- To the OS a process might have just been created, it is ready for execution but not running, it is running i.e. using the CPU, suspended due to some reason i.e. not ready for execution, or it has just finished.
- There may be finer divisions as well.
- Following is a simplified state-transition diagram of a process.

Process States and Transitions

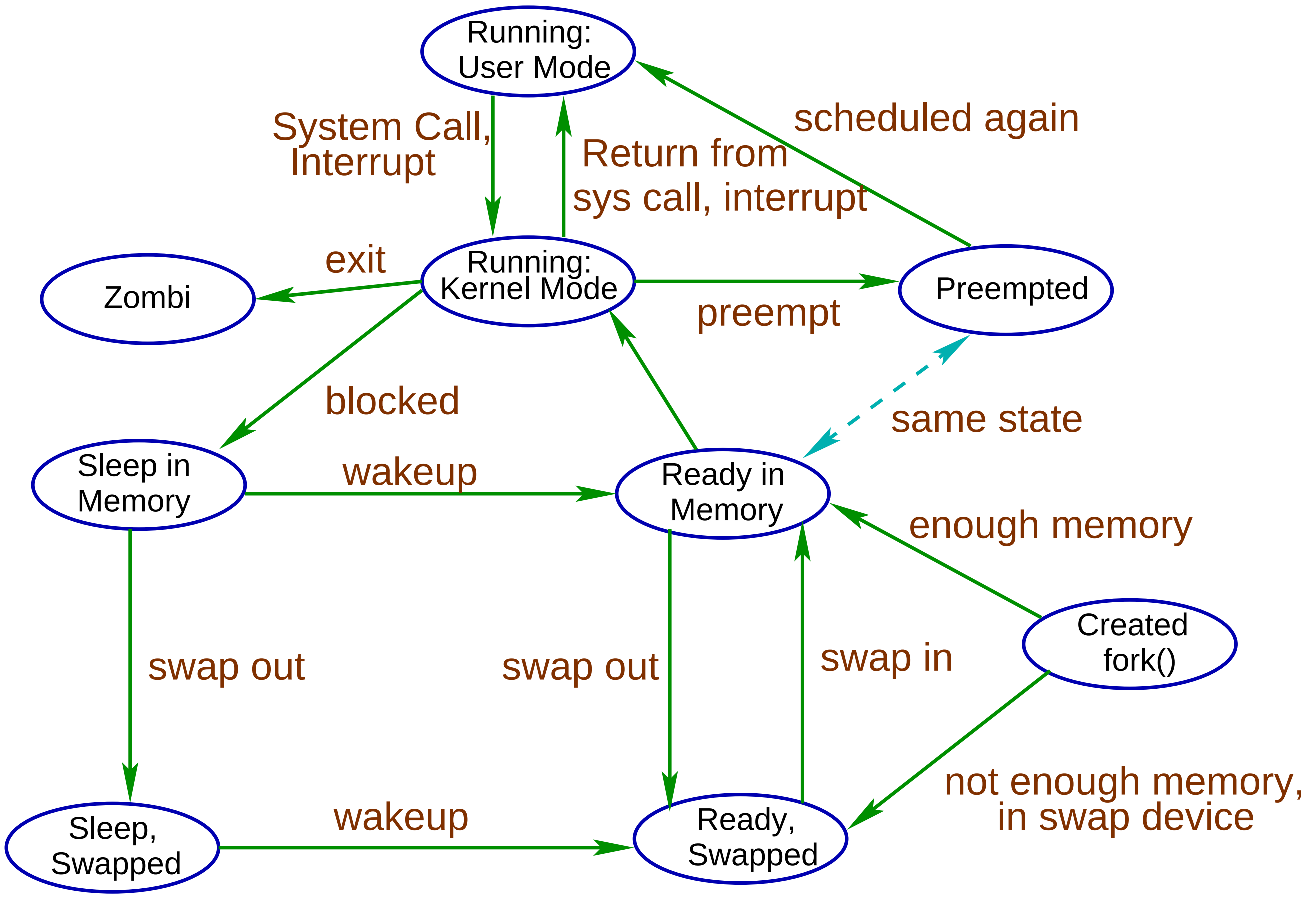


5 state model

Process States

The state transition of Unix system V is more complicated and is as follows.

States and Transitions in Unix (Maurice J Bach)



Design of the Unix Operating System (page 148)

Process Creation

- Our discussion is biased by Unix/Linux like OS.
- We already know that a Linux kernel starts the first process `init`[a].
- Every process has an identification number called process ID (PID).
- The PID of `init` is 1[b].

[a] Replaced by Upstart or systemd: https://www.tecmint.com/systemd-replaces-init-in-linux/

[b] Give the command: `ps -A | less`

Process Creation

- Any other process is a descendant of `init`.
- A user process can create a child process by sending a request (system call) to the OS. In Linux the call is `fork()`.
- On receiving a `fork()` request the OS creates a new process (child) in the image of the requesting (parent) process.

Process Creation

- The `fork()` call returns values to the parent as well as to the child process.
- It returns the PID of the child process to the parent. And returns 0 to the child process.

Process Creation in Windows

A new process is created in Windows by the `CreateProcess` function.

```
BOOL CreateProcess (
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpsaProcess,
    LPSECURITY_ATTRIBUTES lpsaThread,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
```

```
LPVOID lpEnvironment,
LPCTSTR lpCurDir,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcInfo)
```

http://www.informit.com/articles/article.aspx?p=362660&seqNum=2

Process Creation

Following program shows the creation of a child process in Linux (Unix).

```
#include <iostream>
using namespace std;
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main() {  // createProc1.c++
   int chPID, status ;

   chPID = fork();
   if(chPID == -1){
     cerr << "fork() failed\n";
     exit(1);
   }
   if(chPID > 0) {  // parent
      sleep(1);
```

```
    cout << "In parent: parent pid = "
         << getpid() << "\n"
         << "\tIn parent: child pid = "
         << chPID << "\n" ;
    waitpid(chPID, &status, 0) ;
}
else {  // child
    cout << "In child: child pid = "
         << getpid() << "\n"
         << "\tIn child: parent pid = "
         << getppid() << "\n" ;
```

```
    } return 0 ;
}
```

Process Creation

- We mentioned earlier that `fork()` is a system call.
- Any system call uses a machine instruction (x86_64) that causes a software interrupt.
- In the previous program there are three other system calls, `waitpid()`, `getpid()` and `getppid()`.

`waitpid()`

- This system call suspends the execution of the calling process (`main()` in this case) until the child, whose pid is specified, changes state e.g. terminates.
- The exit status of the child is stored in the variable pointed by &status.
- See the manual for more detail.

Inline Assembly Language in C++ Code

- Assembly language code can be embedded in a C or C++ program.
- The keyword `asm` is used to embed a segment of assembly language code.
- The keyword `volatile` is used to disable optimization by the GCC compiler on the specified assembly language code.

Inline Assembly Language in C++ Code

- Extended asm allows to pass input parameters to CPU registers from program variables; and output contents of CPU registers to variables.
- See internet for more information[a].

[a] https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/Using-Assembly-Language-with-C.html
https://www.cs.virginia.edu/ clc5q/gcc-inline-asm.pdf

Process Creation

- In the following code we simply replace `fork()` by the inline assembly code of x86_64 architecture.
- 57 is the code for `fork()`, loaded in the CPU register rax[a]. There is no parameter.
- The machine instruction syscall generates the software interrupt.

[a] ABI specifies that.

https://www.cs.utexas.edu/ bismith/test/syscalls/syscalls.html
http://syscalls.kernelgrok.com/

Process Creation

```
........
int main() { // createProc2.c++
   int chPID, status ;

   __asm__ __volatile__(
        "movq $57,%%rax \n\t"
        "syscall \n\t"
        :"=a" (chPID)
    );
  ..............
```

Note

`:"=a" (chPID)` instructs to transfer the content of `eax` to the program variable `chPID`. The assembly language code is -

```
movq $57,%rax
syscall
movl %eax, -28(%rbp)  # chPID is Mem[rbp-28]
```

Process Creation using Python

Let us see how one creates a child process in Python.

Process Creation in Python

```
#!/usr/bin/python
# createProc3.py creates two process,
#                  prints pid and ppid
import os
import time
def main():
    try:
        chPID = os.fork()
        if chPID > 0:
```

```
        time.sleep(1)
        print 'In parent: parent pid =', \
                        os.getpid(),'\n'
        print 'In parent: child pid  =', \
                        chPID, '\n'
        os.waitpid(chPID,0)
    else:
        print '\tIn child: child pid =', \
                        os.getpid(),'\n'
        print '\tIn child: parent pid  =', \
                        os.getppid(),'\n'
```

```
    except:
        OSError
        print 'fork() fails'
main()
```

Time Sharing

- Once the child is created, both the parent and the child are scheduled to run in time-shared mode.
- Following code demonstrates the time sharing.

Time Sharing

```
#!/usr/bin/python
# createProc4.py scheduling of parent and child, t
import os
def main():
    try:
        chPID = os.fork()
        if chPID == 0:
            msg = 'child running\n'
            n = 5
```

```
        else:
            msg = 'parent running'
            n = 6
        while n > 0:
            print msg
            count = n*10000000
            while count > 0: count = count - 1
            n = n - 1
    except:
        OSError
        print 'fork() fails'
```

`main()`

Process Termination

- The normal termination of a process is through the system call `exit()`.
- After the termination of a process (end of `exit()`), OS reclaims all the resources e.g. memory, internal data structure of OS, etc. used by the process.

Process Termination

- If the parent process terminates before the child, the init[a] process becomes the new parent of the child on Linux.
- Following code demonstrates this.

[a]It may have a different name e.g. `systemd` or `upstart` that replaces `init`. The reason is explained in `en.wikipedia.org/wiki/Upstart`

init or systemd is New Parent

```
/*
  createProc5a.c++ death of parent
*/
#include <iostream>
using namespace std;
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
```

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

int main() { // createProc5a.c++
    int chPID;

    chPID = fork();
    if(chPID == -1){
      cerr << "fork() failed\n";
      return 0;
    }
    if(chPID > 0) {  // parent
```

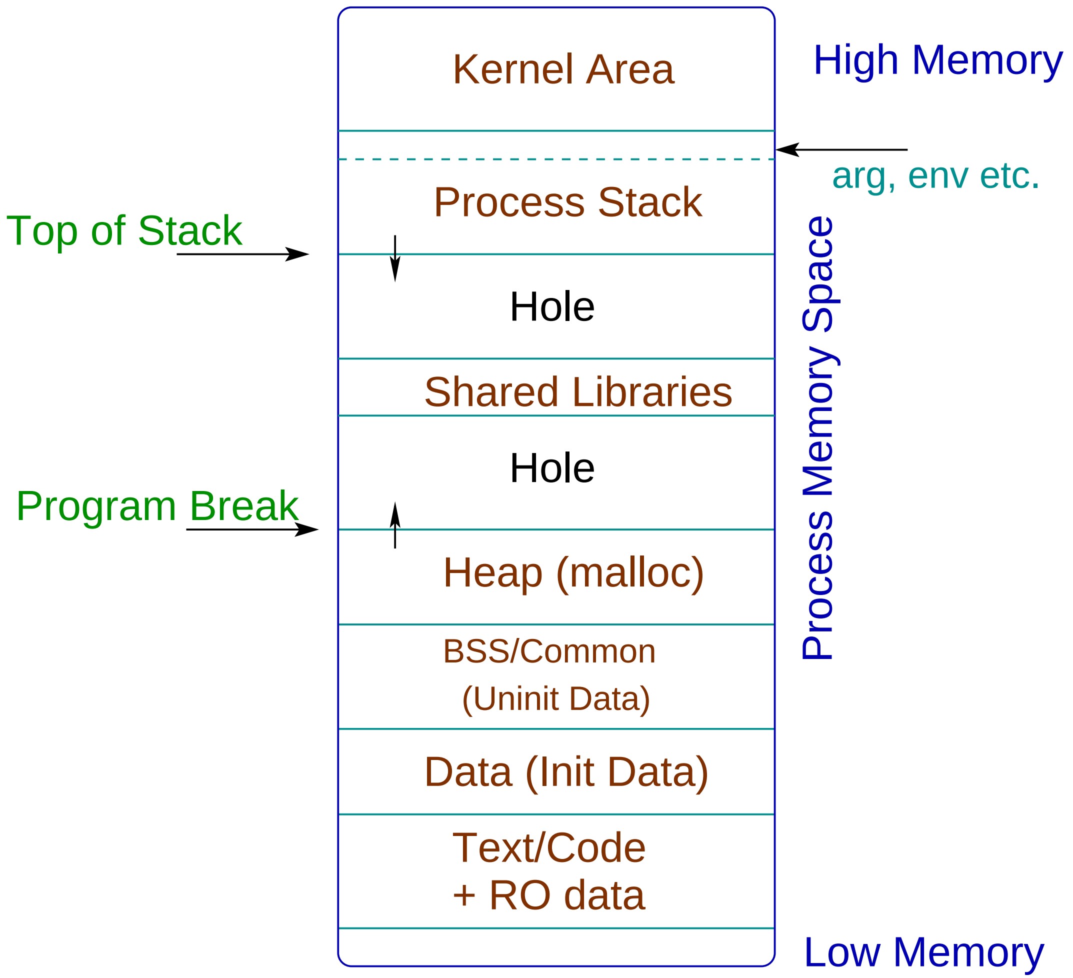
```
        sleep(1);
        cout << "In parent: parent pid = "
             << getpid() << "\n"
             << "\tIn parent: child pid = "
             << chPID << "\n" ;
    }
    else {  // child
        cout << "In child: child pid = "
             << getpid() << "\n"
             << "\tIn child: parent pid = "
             << getppid() << "\n" ;
```

```
        sleep(5);
        cout << "Again in child: child pid = "
             << getpid() << "\n"
             << "\tIn child: parent pid = "
             << getppid() << "\n" ;
    }
    return 0 ;
}
```

Memory Image of a Process

- The logical memory space of a process is divided in several parts.
- The OS kernel is mapped to the higher address and is protected.

Logical Memory of a Process



Memory Image of a Process

- The text segment contains the executable code. This portion is read-only to protect it from unintentional modification.
- It is also shareable so that more than one process can share the same code by mapping in their own logical spaces.
- Often the read-only data are also stored in this segment.

Memory Image of a Process

- There is data area that contains initialized global and static data. This segment has read and write permissions.
- This area is initialized from the executable file when the program is loaded in the main memory.

Memory Image of a Process

- The uninitialized global and static data occupy another memory region bss[a]. The region is kept it separate from the initialized data as the executable file does not store any information other than the address and size of an uninitialized object.
- The region is initialized to all zero during program loading.

[a]Block started by symbol.

Memory Image of a Process

- Remaining portion of the user space is divided mainly into stack and heap areas.
- The stack grows normally from higher address to lower address. But heap grows in the reverse direction. The top end of the heap is called program break.

Memory Image of a Process

- The stack holds data related to activation or call of functions e.g. local variables, parameters to functions, return values and return addresses.
- The heap grows due to run-time request of memory allocation.
- Shared libraries are mapped to some part of the logical memory space.

Memory Image of a Process

- The code and data of Linux kernel is mapped to the highest address of the logical memory space of a process.
- But this area is protected from direct user access.
- The logical memory address space of x86_64 architecture is huge. It is the lower order 48-bits of the 64-bit address i.e. 256 TB.

Different Memory Areas

Following program shows logical addresses of different objects in the user space.

Different Memory Regions

```
#include <iostream>
using namespace std;

int a, b =10;
int main() { // memoryRegions1.c++
    int (* mainPtr)() = main ;
    char *ro = (char *)"IIIT Kalyani", *p, c;
    static char s=10;
    p = new char;
```

```
    p = new char;
    cout <<"main() starts: " <<(void *)mainPtr <<
    cout <<"Read-only data: " <<(void *)ro << endl
    cout <<"Init global data: " <<(void *)&b << en
    cout <<"Uninit global data: " <<(void *)&a <<
    cout <<"Dynamic data: " <<(void *)p << endl;
    cout <<"Local data: " <<(void *)&c << endl;
    cout <<"Local static data: " <<(void *)&s << e
    return 0 ;
}
```

Different Memory Regions

```
$ ./a.out
main() starts: 0x400876
Read-only data: 0x400ae4
Init global data: 0x601068
Uninit global data: 0x601194
Dynamic data: 0xdb0c40
Local data: 0x7fffe283b7ff
Local static data: 0x60106c
```

Logical Memory to Physical Memory

- The logical memory space of a process is divided into different segments and/or pages.
- The main memory of the system is also divided into memory frames.
- Logical pages are mapped to different (need not be contiguous) physical memory frames called page frames.

Logical Memory to Physical Memory

- The mapping of logical memory space to actual memory is maintained by the OS using the memory management hardware and memory resident table(s) e.g. segment table, page table.
- Access permission to a portion of the logical space can be restricted e.g. read-only, read and execute, through the mapping table.

Logical Memory of Parent and a Child

- The memory image of the child process created by fork() is identical to the parent.
- But often the child process gives an exec() call to upload another executable file into its logical address space[a].
- So it is unnecessary to immediately create a separate copy the parent's memory-image for the child process.

[a] A command interpreter e.g. `bash` is doing that all the time.

Logical Memory of Parent and Child

- Initially identical pages of both parent and child are mapped (by the page table) to same page frames in the memory.
- But all page frames are marked as read-only. A reference counter per frame is maintained to keep track of the number of processes sharing the frame.

Logical Memory of Parent and Child

- There is no issue as long as the shared pages are only read.
- But if there is a write access to a writable page frame of reference count > 1 by any process, it is copied to a new frame.

Logical Memory of Parent and Child

- The page table of the writing-process is modified and the reference count of both the frames are updated.
- This technique known as copy-on-write (CoW) is used to speed-up process creation.

Logical Memory of Parent and Child

Given the above scenario, explain the output of the following program.

Explain the Output

```
// procMem1.c++ memory of child process
#include <iostream>
using namespace std;
#include <unistd.h>
#include <cstdio>
#include <cstdlib>
#include <sys/types.h>

int main() { // procMem1.c++
```

```
int chPID, n ;

chPID = fork();
if(chPID == -1) { // fork fails
  cerr << "fork() fails\n";
  return 0;
}
if(chPID > 0){ // parent
  n=10;
  cout << "In parent &n:" << (void *)&n
       << ", n: " << n << endl;
```

```
    }
    else {  // child
      n=15;
      cout << "In child  &n:" << (void *)&n
           << ", n: " << n << endl;
    }
    return 0 ;
}
```

Explain the Output

```
$ a.out
In parent &n:0x7ffe4936c0f0, n: 10
In child  &n:0x7ffe4936c0f0, n: 15
```

Shell Commands

- In a command interpreter such as bash there are certain commands that are internal e.g. `cd, pwd` etc. that are built in as system calls in the code of the interpreter.
- But there are other commands e.g. `ls, file`, a.out etc. that are independent executable files.

Internal Command

- The command interpreter executes the internal commands directly. As an example, to change the directory (`cd`) it uses the appropriate system call or invokes the wrapper function `chdir()` with the new path as a parameter.
- Following is an example.

Changing Current Directory

```
// changeDir.c++ changes the current working dire
//                by calling chdir()
//                $ ./a.out <path>
#include <iostream>
using namespace std;
#include <cstdlib>
#include <unistd.h>
int main(int ac, char *av[]){
    char *cwdP;  // changeDir.c++
```

```
    if(ac < 2){
      cout << "No path specified\n";
      return 0;
    }
    cwdP=get_current_dir_name();
    cout << cwdP << endl;
    chdir(av[1]);
    cwdP=get_current_dir_name();
    cout << cwdP << endl;
    return 0;
}
```

Changing Current Directory

```
$ a.out ..
/home/goutam/IIITKalyani/operatingSystem/lect/l2P
/home/goutam/IIITKalyani/operatingSystem/lect
```

Command as an Executable File

- To run an executable file e.g. `ls` or `a.out`, it is necessary to load the corresponding image in the memory. But the command interpreter cannot destroy its own image by loading another executable in its space.
- So, it creates a child process.

Replacing Process Image

- The part of the code executed in the child process gives an `exec()` call[a] which replaces the parent's image in the child process with the image of the executable file specified in the command.
- The path of the executable file and other information are passed as parameters to the `exec()` call.

[a] Already present in the interpreter program.

Replacing Process Image

- Following is a sample program that uses `execve()` system call to load the image of a program computing factorial of a positive integer.
- The factorial program takes its input through the command line.

Factorial Program

```
//  factorial.c++ computes factorial, takes
//                command line argument
//  $ g++ -Wall factorial.c++ -o factorial
#include <iostream>
using namespace std;
#include <cstdlib>
int main(int count, char *vects[]) {
    int n, i, fact = 1 ;
```

```
    if(count < 2) {
      cerr << "Missing 2nd argument\n" ;
      return 0 ;
    }
    n = atoi(vects[1]) ;
    for(i=1; i<=n; ++i) fact *=i ;
    cout << n << "! = " << fact << endl;
    return 0 ;
}
```

Execute Factorial

```
$ ./factorial
Insufficient command line argument
$ ./factorial 0
0! = 1
$ ./factorial 5
5! = 120
```

execve() ./factorial

```
/**********************************
 * This program uses execve system  *
 * call. Execute -                  *
 * $ ./a.out ./factorial 6          *
 * *********************************/
#include <iostream>
using namespace std;
#include <sys/types.h>
#include <unistd.h>
```

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

int main(int argc, char *argv[], char *envp[]) {
    int chPID, status ;
    char **agv;              // execve1.c++

    if(argc < 3){
      cerr << "Less number of arguments\n";
      return 0;
    }
    agv=argv+1;
```

```
chPID = fork();
if(chPID == -1) {
  cerr << "fork() error\n";
  return 0;
}
if(chPID > 0) { // Parent
  cout << "Inside Parent\n" ;
  waitpid(chPID, &status, 0) ;
  cout << "child " << chPID << " terminates\n
}
else { // Child
```

```
        int err;
        cout << "Inside Child\n" ;
        err = execve(agv[0], agv, envp) ;
//      err = execvp(agv[0], agv) ;
        if(err == -1){
          cerr << "exec fails\n" ;
          return 0;
        }
      }
 return 0 ;
}
```

execve() ./factorial

```
$ ./a.out factorial 5
Inside Parent
Inside Child
5! = 120
child 7582 terminates
```

System Call for `execve()`

- The syscall code for `execve()` is 59.
- The ABI specification is as follows:
  - `rax` ← syscall code,
  - `rdi` ← 1^{st} parameter,
  - `rsi` ← 2^{nd} parameter,
  - `rdx` ← 3^{rd} parameter.
  - `eax` holds the return value.

System Call for `execve()`

In the inline assembly language code,

- `rdi` is `D`,
- `rsi` is `S`,
- `rdx` is `d`,
- `eax` is `a`.

execve() ./factorial

```
.......
int main(int argc, char *argv[], char *envp[]) {
    int chPID, status ;
    char **agv = argv+1;
       // execve2.c++
.......
```

execve() ./factorial

```
// execve(agv[0], agv, envp) ;
__asm__ __volatile__(
  "movq $59, %%rax\n\t"
  "syscall\n\t"
  :"=a" (err)
  :"D" (agv[0]), "S" (agv), "d" (envp)
                    ) ;
```

......

Try with `execv()`

```
$ a.out factorial 5
Inside Parent
Inside Child
5! = 120
child 9737 terminates
```

Try with `execv()`

- Try with `execv()` to write a similar code.
- `factorial1` takes input from `stdin`.
- See the manual for the meaning of ‘v’, ‘p’, ‘e’ etc.
- A Python version of the `execv()` call is as follows.

os.execv() ./fact.py

```
#!/usr/bin/python
# execv3.py loads a process image in the child
import os
import sys
if len(sys.argv)==1:
    print 'Less arguments'
    sys.exit(0)
def main():
    path = sys.argv[1]
```

```
arg2 = sys.argv[1:]
try:
    chPID = os.fork()
    if chPID > 0:
        print 'In parent'
        os.waitpid(chPID,0)
        print 'Child', chPID, 'ends'
    else:
        print 'In child'
        try: os.execv(path, arg2)
        except: OSError
```

```
    except: OSError
main()
```

os.execv() ./fact.py

```
$ execv3.py fact.py 5
In parent
In child
5 ! = 120
Child 12087 ends
```

Child Process and Open Files

- An IO device is often treated as a special file under `/dev`.
- Three device files, stdin, stdout and stderr, are normally open (inherited from the parent) in a process.
- An open file is identified by its file descriptor in a process. It is a non-negative integer, an index to file-descriptor table.

Child Process and Open Files

- The file descriptors for the standard input (stdin) is 0 (`STDIN_FILENO` defined in `unistd.h`), for the standard output (stdout) it is 1 (`STDOUT_FILENO`) and for the standard error (stderr) it is 2 (`STDERR_FILENO`).
- A child process inherits all the open file descriptors from the parent after the `fork()`.

Child Process and Open Files

```
//  fileDes1.c++ printing file descriptor of
//                      parent and child
#include <iostream>
using namespace std;
#include <cstdio>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
```

```
int main() { // fileDes1.c++
   int chPID, status ;
   chPID = fork();
   if(chPID < 0){
     cerr << "fork() error\n";
     return 0;
   } //
   if(chPID != 0) {  // parent
      cout << "In parent:\n"
           << fileno(stdin) << ": stdin\n"
           << fileno(stdout) << ": stdout\n"
```

```
            << fileno(stderr) << ": stderr\n";
        waitpid(chPID, &status, 0) ;
    }
    else {  // child
        cout << "In child:\n"
            << fileno(stdin) << ": stdin\n"
            << fileno(stdout) << ": stdout\n"
            << fileno(stderr) << ": stderr\n";
    }
    return 0 ;
}
```

Output: File Descriptors

```
$ ./a.out
In parent:
0: stdin
1: stdout
2: stderr
In child:
0: stdin
1: stdout
2: stderr
```

`open()` and `close()`

- Given a file pathname the system call `open()` makes an entry in the table[a] of open files and returns the file descriptor, a reference or index of the entry.
- The descriptor has the smallest non-negative integer that does not correspond to any other open file.
- On error the call returns -1.

[a] The entry keeps track of file offset and a few other information.

`open()` and `close()`

- Given a file descriptor the system call `close()` dissociates the descriptor from its file.
- This descriptor is available for new file to open.
- On success it returns 0 and on error it returns -1.

Redirecting Output

- Following code shows how one can close the file descriptor 1 of the `stdout`, and open a file with 1 as its descriptor.
- Functions that are suppose to write on stdout will subsequently write in the open file, a redirection of the output.
- Similarly input can also be redirected from a file.

Redirecting Output

```
//  fileDes2.c++ close stdout,open new file for o
//                $ ./a.out <output file name of c
#include <iostream>
using namespace std;
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

int main(int ac, char **av ) { // fileDes2.c++
    int chPID, status, fd ;
    if(ac < 2){
        cerr << "Less arguments\n";
        return 0;
    }
    chPID = fork();
    if(chPID == -1){
```

```
        perror("fork() error\n");
        exit(1);
    } //
    if(chPID != 0) {  // parent
         cout << "In parent\n";
         waitpid(chPID, &status, 0) ;
    }
    else {  // child
         if(close(fileno(stdout)) == -1){
             cerr << "File close error" << endl;
             return 0;
```

```
        }
        fd = open(av[1], O_WRONLY | O_CREAT, 0666);
        if(fd == -1){
          cerr << "File open error" << endl;
          return 0;
        }
        cout << "In child" << endl;
        close(fd);
    }
    return 0 ;
}
```

Redirecting Output in Python

```
#!/usr/bin/python
# fileDes3.py redirecting output in child
import os, sys
def main():
    fileNm = raw_input('Enter the output file nam
    try:
        chPID = os.fork()
        if chPID > 0:
            print 'In parent'
```

```
        os.waitpid(chPID,0)
        print 'Child', chPID, 'ends'
    else:
        print 'In child'
        try:
            os.close(sys.stdout.fileno())
        except:
            OSError
            print "os.close() fails"
        try:
            os.open(fileNm, os.O_CREAT+os.O_WRON
```

```
            except:
                OSError
                print "os.open() fails"
            print "Again in child"
    except:
            OSError
            print "fork() fails"
main()
```

OS Data Structure for Process

- We have already mentioned that it is necessary to save the state of a process when it is preempted or suspended.
- OS maintains a data structure called process control block (PCB) for each process.
- It maintains a list of PCBs for all processes present in a system.

OS Data Structure for Process

- The PCB is created during process creation and is removed on termination.
- The image of a child process is often overwritten using an `exec()` call.
- Some of the fields of the child's PCB e.g. PID, parents PID, open files etc. are unchanged. But some other fields e.g. memory mapping etc. are modified after the `exec()` call.

OS Data Structure for Process

- There are large number of fields in a PCB. In Linux a PCB is called a process descriptor. It is of type `task_struct`.
- The size of `task_struct` of Linux is more than a several KB which contains lots of information.
- Some of the essential and basic information saved in a PCB are as follows.

Some Information Saved in PCB

- Process identification - PID.
- State of the process - ready, running, suspended, zombie etc.
- CPU-FPU state i.e. the content of different CPU registers, program counter (PC), program status word (PSW) etc.[a].

[a] Often this is not directly kept in the PCB, but on the system stack. A pointer to that may be saved in the PCB.

Some Information Saved in PCB

- Memory management information e.g. content of the page-table base register. Information about segmentation etc.
- Information about the open files i.e. IO status.
- Pointers to parent, child and sibling.
- Scheduling priority information.

Some Information Saved in PCB

`/usr/src/kernels/3.10.0-229.4.2.el7.x86_64/include/linux`

The `procfs` File System and `/proc`

- The procfs is a special file system of Linux. It is available under the directory `/proc`.
- It provides information about the system and also about different process.
- A listing of the directory provides the following information.

```
ls /proc
```

```
1       2546  3125 ...   devices    pagetypeinfo
10      2564  32   ...   diskstats  partitions
             ...................
23      3039  4175 ...   cpuinfo    mtrr
```

The `procfs` File System and `/proc`

- The numbers are subdirectories related to different processes. The subdirectory name is the PID.
- The directory cpuinfo provides information about CPU, the directory `meminfo` provides information about the memory subsystem, the directory `fs` provides information about the file system etc.

Shared Code

We use `/proc/self/pagemap` to show that the code of a parent process and its child process share the same main memory page frame.

```
/*
 * shareCode.c++
 * $ sudo ./a.out
 */
#include <iostream>
using namespace std;
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/stat.h>
#include <fcntl.h>
#define WORDSZ 64
#define PTentSZ 8

// shareCode.c++ $ sudo ./a.out
void printBits(uint64_t wrd){
// prints the bits of 64 word
```

```
int bit[WORDSZ]={0};

for(int i=0; i<WORDSZ; ++i) {
    bit[i] = wrd%2;
    wrd /= 2;
}
for(int i=WORDSZ-1; i>=0; --i) {
    cout << bit[i] ;
    if(i%4 == 0) cout << " " ;
}
cout << endl;
```

```
}


int main(){
    int fd, pid, pageSz = sysconf(_SC_PAGESIZE);
    int mainPn = (long int)main/pageSz;
    uint64_t data=0;

    cout << "Address main(): " << hex
         << (unsigned long int)main << " Page no.
         << hex << mainPn << endl;
    pid = fork();
```

```
if(pid == -1){
  cerr << "Fork() error\n";
  return 0;
}
if(pid > 0){ // parent
  cout << "Parent pid: " << getpid() << endl;
  fd = open("/proc/self/pagemap", O_RDONLY);
  if(fd == -1) {
    cerr << "File open error\n";
    return 0;
  }
```

```
}
else {          // child
   cout << "Child: " << getpid() << endl;
   fd = open("/proc/self/pagemap", O_RDONLY);
   if(fd == -1) {
      cerr << "File open error\n";
      return 0;
   }
}
pread(fd, &data, sizeof(uint64_t), PTentSZ*ma
if(data != 0) {
```

```
        cout << " Table entry: " << hex << data <<
        printBits(data);
    }



    close(fd);
    return 0;
}
```

Output

```
$ g++ -Wall shareCode.c++
$ sudo ./a.out
Address main(): 400cdf Page no.: 400
Parent pid: 3429
 Table entry: a0800000000845bd
1010 0000 1000 0000 0000 0000 0000 0000
     0000 0000 0000 1000 0100 0101 1011 1101
Child: 342a
 Table entry: a1800000000845bd
```

```
1010 0001 1000 0000 0000 0000 0000 0000
     0000 0000 0000 1000 0100 0101 1011 1101
```

- Bit-63: page present in main memory.
- Bit-62: page swapped out.
- Bit-61: file-page or shared.
- Bit-56: page exclusively mapped.
- Bit-55: pte is soft-dirty.
- Bit-0-54: page frame no. (if present), swap info (if swapped).

Cannot Show CoW

I was expecting to show copy on write (CoW). But it is not working as I expected. (`cow.c++`).

Processor: `ls /proc/cpuinfo`

```
processor : 0
........................
model name : Intel(R) Core(TM)2 Duo CPU E6550  @ 
cpu MHz : 1998.000
cache size : 4096 KB
cpu cores : 2
........................
processor : 1
........................
```

```
model name : Intel(R) Core(TM)2 Duo CPU E6550  @
........................
cpu MHz : 1998.000
cache size : 4096 KB
cpu cores : 2
```

Memory: `ls /proc/meminfo`

```
MemTotal:          4039016 kB
MemFree:            670064 kB
Buffers:            247056 kB
Cached:            1445992 kB
SwapCached:              0 kB
Active:            1898144 kB
Inactive:          1210120 kB
Active(anon):      1562936 kB
Inactive(anon):     380496 kB
```

```
Active(file):      335208 kB
Inactive(file):    829624 kB
Mlocked:               40 kB
SwapTotal:        4882428 kB
SwapFree:         4882428 kB
Dirty:                172 kB
Shmem:             528224 kB
...........................
DirectMap4k:       116288 kB
DirectMap2M:      4067328 kB
```

Run the Following Program

```
#include <iostream>
using namespace std;
#include <sys/types.h>
#include <unistd.h>

int main(){ // seeProcfs1.c++

    int pid = getpid();
    cout << pid << " is PID\n";
```

```
    while(1);

    return 0;
}
```

```
$ ./a.out
9746 is PID
```

The code is in a while-loop. We look into the subdirectory `/proc/9746`.

```
$ ls /proc/9746/

attr               cpuset   limits
autogroup          cwd      loginuid
auxv               environ  maps
cgroup             exe      mem
clear_refs         fd       mountinfo
cmdline            fdinfo   mounts
comm               io       mountstats
coredump_filter    latency  net
..................................
```

```
ns               sched       syscall
numa_maps        schedstat   task
oom_adj          sessionid   wchan
oom_score        smaps
oom_score_adj    stack
pagemap          stat
personality      statm
root             status
```

Some Contents of /proc/9746/

```
$ cat /proc/9746/cmdline
./a.out
$ ls -l /proc/9746/fd/
total 0
lrwx------ 1 ... 64 Jul 17 09:56 0 -> /dev/pts/4
lrwx------ 1 ... 64 Jul 17 09:56 1 -> /dev/pts/4
lrwx------ 1 ... 64 Jul 17 09:56 2 -> /dev/pts/4
$ cat /proc/9746/limits
Limit                    Soft Limit   Units
```

Max cpu time	unlimited	seconds
Max file size	unlimited	bytes
Max data size	unlimited	bytes
Max stack size	8388608	bytes
Max core file size	0	bytes
Max resident set	unlimited	bytes
Max processes	31417	processes
Max open files	1024	files
Max locked memory	65536	bytes
Max address space	unlimited	bytes
Max file locks	unlimited	locks

```
Max pending signals         31417        signals
Max msgqueue size           819200       bytes
Max nice priority           0
Max realtime priority       0
Max realtime timeout        unlimited
$ /proc/9746/maps
00400000-00401000 r-xp 00000000 08:05 1577007 ./a
00600000-00601000 r--p 00000000 08:05 1577007 ./a
00601000-00602000 rw-p 00001000 08:05 1577007 ./a
00606000-00638000 rw-p 00000000 00:00 0          [h
7f3424574000-7f342466f000 r-xp 00000000 08:02 145
```

```
                        /lib/x86_64-linux-gnu/libm-2.1
7f3424a24000-7f3424c23000 ---p 001b4000 08:02 145
                        /lib/x86_64-linux-gnu/libc-2.1
7f3425158000-7f342517a000 r-xp 00000000 08:02 145
                          /lib/x86_64-linux-gnu/ld-2.1
7fffbb13b000-7fffbb15c000 rw-p 00000000 00:00 [st
7fffbb1ff000-7fffbb200000 r-xp 00000000 00:00 0
                                                    [v
ffffffffff600000-ffffffffff601000 r-xp 00000000 0
                                                [vsysc
```

Bibliography

1. https://filippo.io/linux-syscall-table/
2. https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
3. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
4. https://docs.python.org/3/library/os.html#process-management
5. http://www.python-course.eu/forking.php
6. Beginning Linux Programming by Neil Mathew & Richard Stones, 3^{rd} ed., Wiley Pub., 2004, ISBN 81-265-0484-6.