CSC227: Operating Systems
Spring 2017

CHAPTER 3

PROCESS CREATION

Dr. Soha S. Zaghloul

LAYOUT

- Unix
- Windows

Process Operations

OSs should be able to perform principal operations on processes such as:

- Create a process
- Destroy (kill) a process
- Suspend a process
- Resume a process
- Change the priority of a process
- Block a process
- Wake up a process
- Dispatch a process
- Enable a process to communicate with other processes: this is known as Inter-Process Communication (IPC)

PROCESS CREATION (I) – INTRODUCTION

A process may spawn or fork (create) another new process.

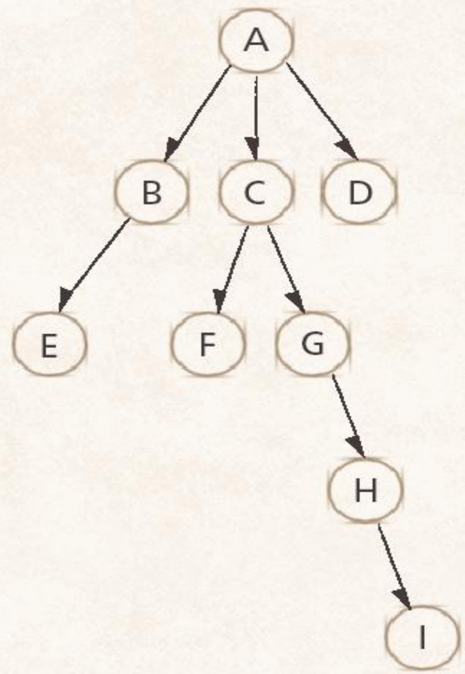
The creating process is called the parent. The created one is the child.

A child process may itself spawn another process.

This forms a hierarchical structure: a tree.

A process may create many children

A process has at most one parent



Process Creation (2) – Parent-Child Relationship

The OS defines the relationship between a parent and its spawned child in three main aspects:

- **Memory Address Space** – there are two possible scenarios:
  1. Both parent and child have the same memory address space
  2. Parent and child have different memory address spaces
- **Mode of Execution** – two scenarios are possible:
  1. The process continues to execute concurrently with its child
  2. The parent waits until some or all of its children terminate
- **Resources** – when a parent process spawns a child, the latter needs resources such as CPU time, memory, files, I/O devices, etc… to accomplish its task.

  The OS assigns resources to a child process in one of the following scenarios:
  1. The child gets its resources directly from the OS
  2. A child may be restricted to a subset of the parent's resources.

  The second scenario prevents any process from overloading the OS by creating too many child processes

Process Creation (3) – Examples of OSs

We'll study the process creation concept in two common operating systems:

- Unix
- Windows

PROCESS CREATION (4) – UNIX(I) – INTRODUCTION

In Unix-based OSs, the first created process is called *init*.

init is created as soon as the kernel loads.

Many processes are then spawned directly from *init*.

Each created process has its own identifier pid. This is stored in the corresponding PCB. The *init* process is given a pid = 1.

The following figure gives an example for a map of processes spawning in Unix:

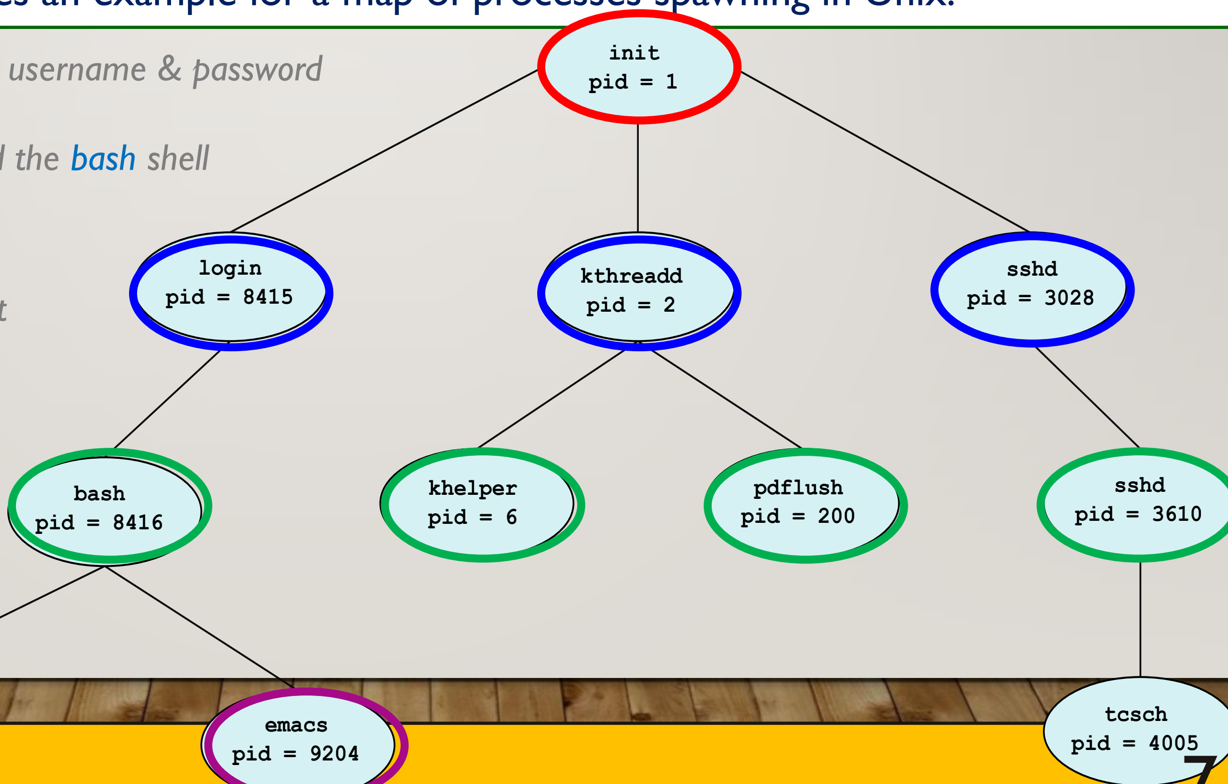
login authenticates the user with a username & password

After authentication, login spawned the bash shell

Using bash, the user created the process ps and the emacs editor. ps lists complete information about active processes.

kthreadd is responsible for creating additional processes that perform tasks on behalf of the kernel, such as khelper and pdflush.

sshd manages clients using ssh



Process Creation (5) – Unix(2) – Memory Address Space (1)

In Unix, a new process is created using the *fork()* system call.

When spawned, the child process "copies" the memory address space of the parent.

fork() returns a negative number if there is an error ; a zero for the child process.

The child process overlays its address space using the *execlp()* command, a version of the *exec()* system call.

```
#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;
}
```

Process Creation (6) – Unix(3) – Memory Address Space (2)

In fact, in UNIX when a child process is created, the OS makes an exact copy of the parent's address space and give it to the child. Therefore, any modifications to the variables made after the *fork()* statement are not reflected in the address space of each other.

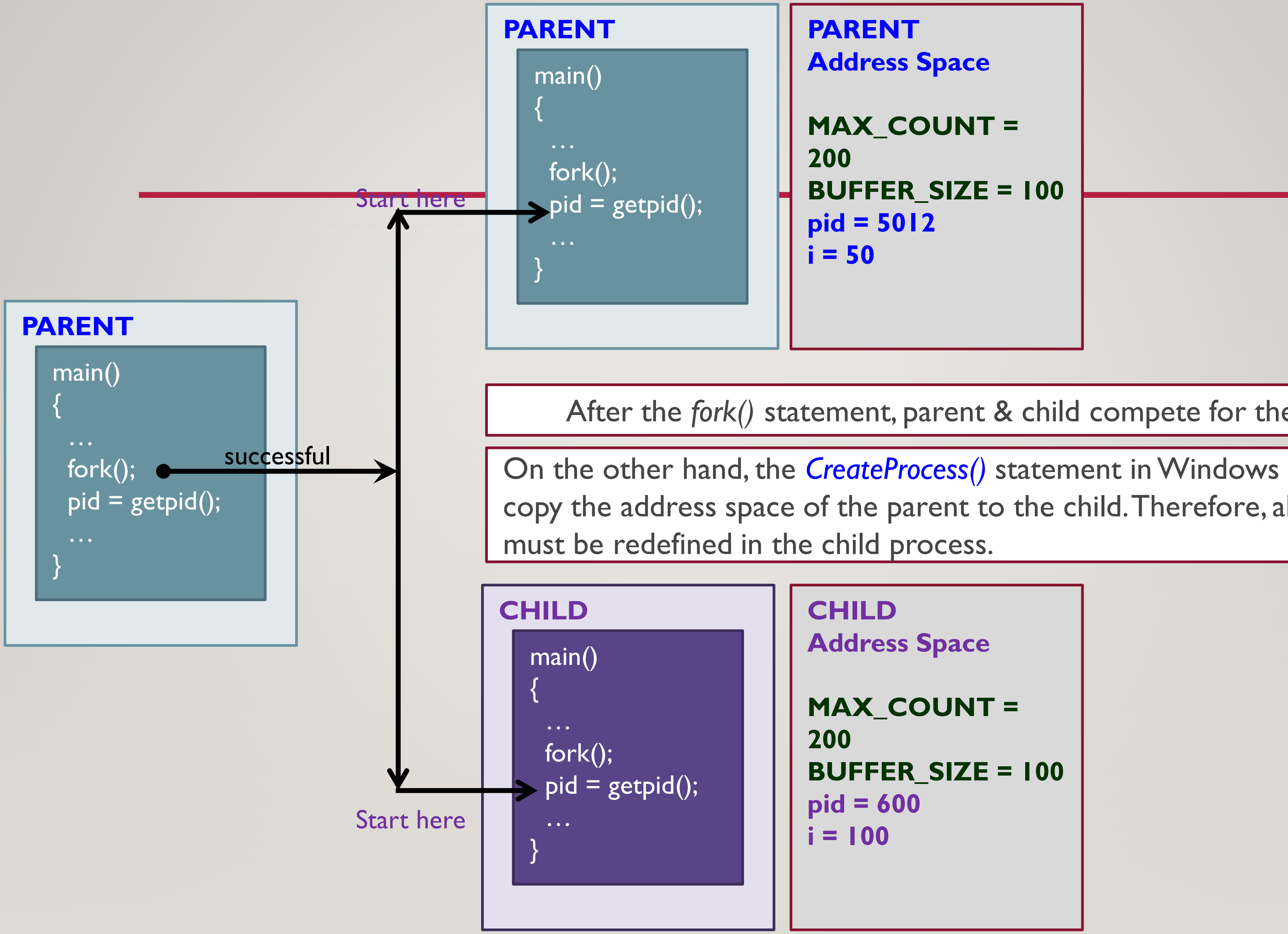
Consider the following example:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define MAX_COUNT 200;
#define BUFFER_SIZE 100;
void main(void)
{
  pid_t pid;
  int i;
  char buffer[BUFFER_SIZE];
  fork();
  pid = getpid();  //returns the ID of the running process
  for (i=1; i < MAX_COUNT; i++) {
   sprintf (buf, "this line is from pid = %d, value = %d\n", pid, i);
   write (1, buf, strlen(buf)); } //end for
} //end main
```

Values initialized <u>before</u> the *fork()* have the same values in both address spaces (parent & child)

Values modified <u>after</u> the *fork()* are not reflected in the copy's address space

Process Creation (7) – Unix(4) – Memory Address Space (3)



After the *fork()* statement, parent & child compete for the CPU.

On the other hand, the *CreateProcess()* statement in Windows does not copy the address space of the parent to the child. Therefore, all variables must be redefined in the child process.

Process Creation (6) – Unix(3) – Resources

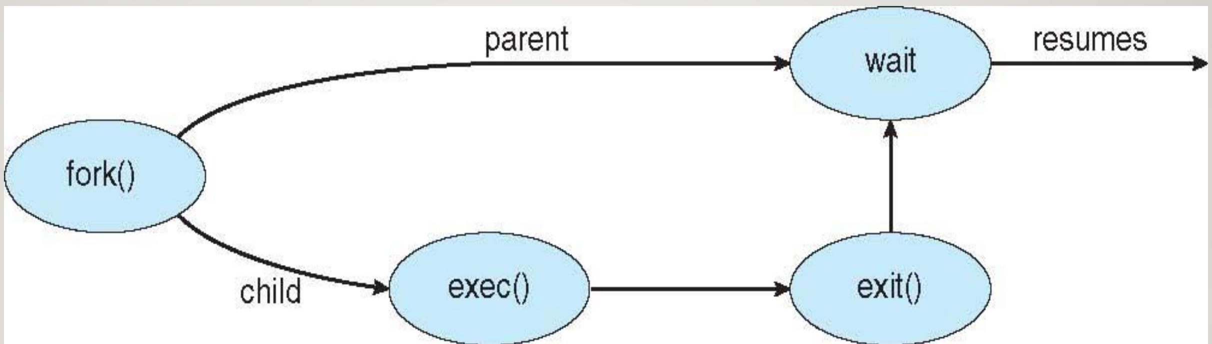
The child process then inherits privileges and scheduling attributes from the parent, as well as certain resources, such as open files.

Process Creation (7) – Unix(4) – Mode of Execution

In Unix, the parent waits for the child process to complete.

This is achieved using the *wait()* command. *Refer to the code in slide 8*.

When the child completes, then the parent resumes its execution.




```
   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;
}
```

PROCESS CREATION (8) – WINDOWS(1) – MEMORY ADDRESS SPACE

Windows uses the *CreateProcess()* to create a child process.

In Windows, a child process has a different address space than its parent.

CreateProcess() has many parameters.

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Process Creation (9) – Windows(2) – Resources

In Windows, the OS assigns resources to child processes directly regardless of those of the parent process.

Process Creation (10) – Windows(3) – Mode of Execution

Like Unix, the parent process waits for the completion of its child before resuming execution.

Re-consider the code previously given in slide 11.

```
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

PROCESS CREATION