# Operating Systems
# CS2006

Lecture 5

## Operation on Processes

13th February 2023

Dr. Rana Asif Rehman

# Operations on Processes

# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

# Process Creation (Cont.)

- 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

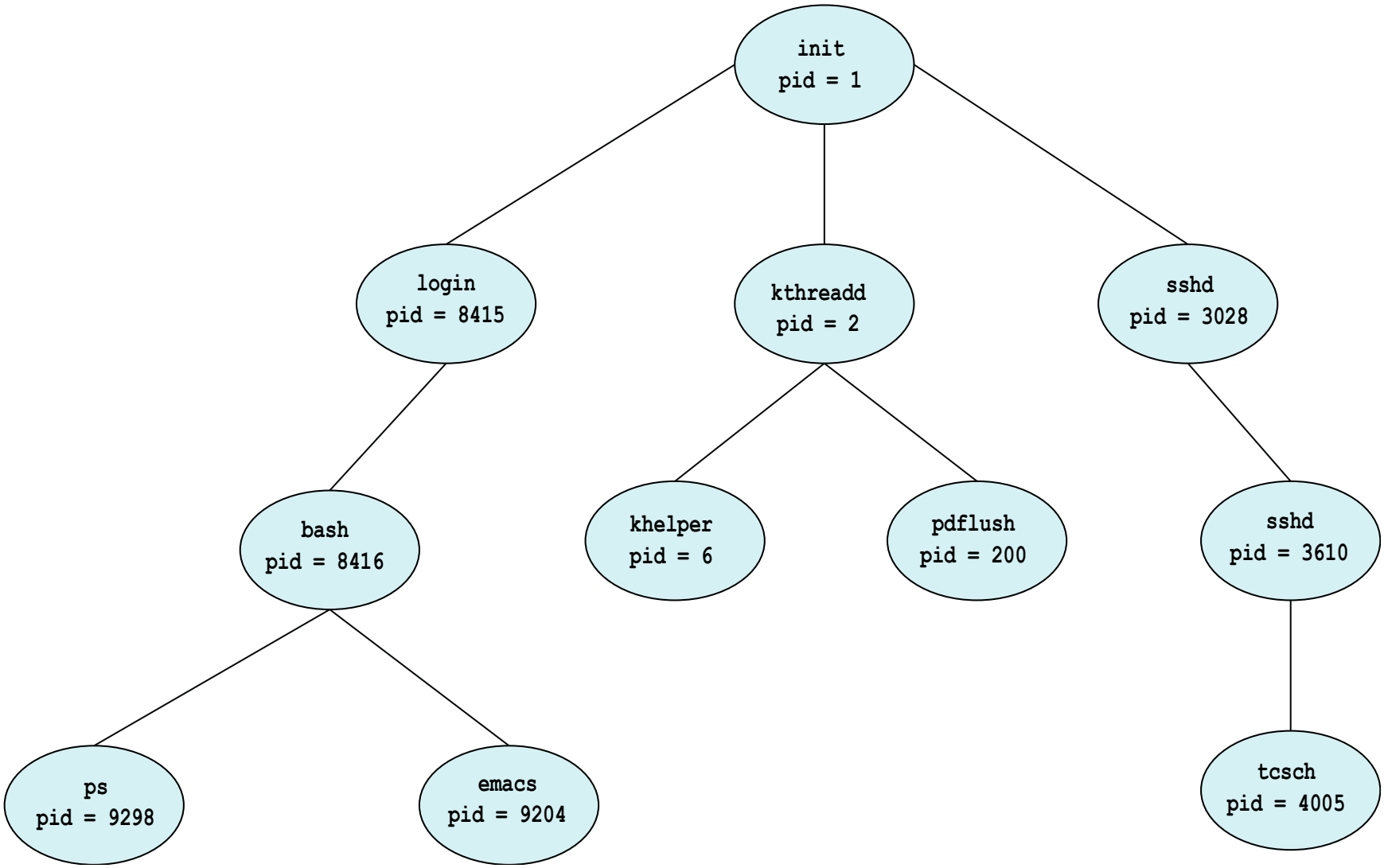
# Process Creation

- Includes
  - Build kernel data structures
  - Allocate memory
- Reasons to create a process
    - Submit a new batch job/Start program
    - User logs on to the system
    - OS creates on behalf of a user (printing)
    - Spawned by existing process

# Unix Process Creation

- When the system starts up it is running in kernel mode
- There is only one process, the initial process.
- At the end of system initialization, the initial process starts up another kernel process.
- The **init** kernel process has a process identifier of 1.

# A Tree of Processes in Linux



# Process Creation

- These new processes may themselves go on to create new processes.
- All of the processes in the system are descended from the init kernel thread.
- You can see the family relationship between the running processes in a Linux system using the **pstree** command
- A new process is created by a **fork()** system call

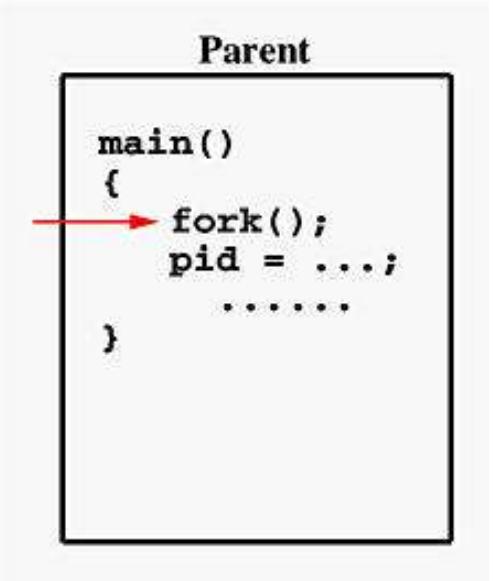
# The fork() system call

At the end of the system call there is a new process waiting to run once the scheduler chooses it

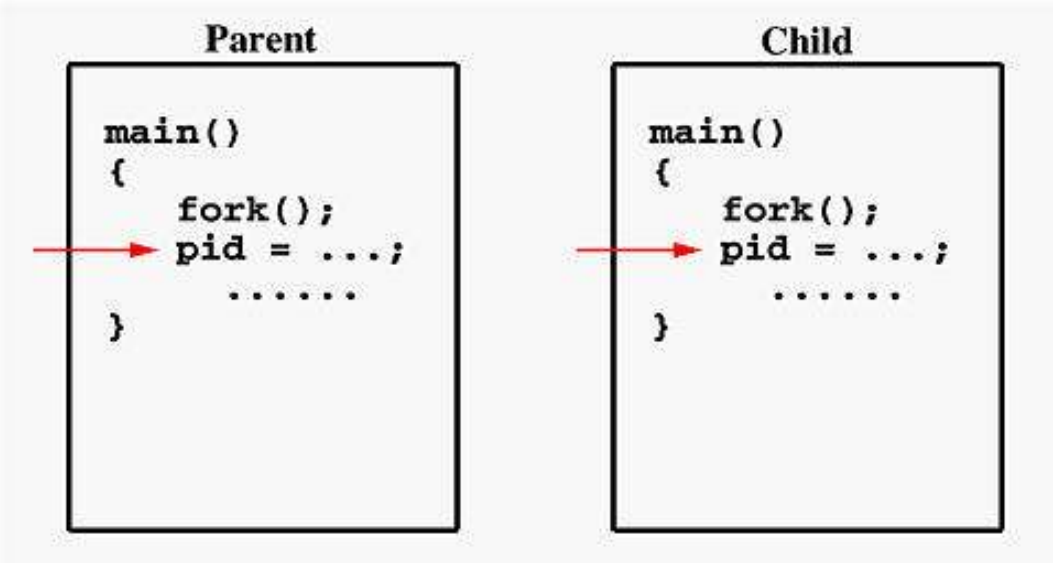
- A new data structure is allocated
- The new process is called the child process.
- The existing process is called the parent process.
- The parent gets the child's pid returned to it.
- The child gets 0 returned to it.
- Both parent and child execute at the same point after fork() returns

# Fork() System call

- Before calling fork()




- After calling fork()



# Example of fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program running fork() instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output?

Hello world!
Hello world!

# Example of fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();

    printf("Hello world!\n");
    return 0;
}
```

# Exercise – fork()

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Question: how many times will “hello” be printed?

# Unix Process Control

```
int main()
{
        int pid;
        int x = 0;

        x = x + 1;
        fork();
        x = 3;
        printf("%d",x)
;
}
```

- Minimal checking for error conditions
- Potential for confusion about open files and which program is writing to a file
- The new process created by the **fork** or **exec** has the three standard files that are automatically opened: **stdin**, **stdout**, and **stderr.**
- If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork
- If the parent and the child processes both read input from a stream, whatever is read by one process will be lost to the other
- Once something has been delivered from the input buffer to a process the pointer has moved on

# But we want the child process to do something else…

```
int pid;
int status = 0;

pid = fork();
if (pid>0) {
      /* parent */
      ……
      pid = wait(&status);
} else {
      /* child */
      ……
      exit(status);
}
```

*The **fork** syscall returns a zero to the child and the child process ID to the parent*

*Parent uses **wait** to slee chil retu and*

***Fork** creates an exact copy of the parent process*

***Wait** variants allow wait on a*

*Child process passes status back to parent on **exit**, to report success/failure*

**a. Write down the output of the following code**

```
int value = 5, i = 0;
int main()
{
        pid_t pid;
        pid = fork();

        if (pid == 0) {
                value += 15;
                printf("Value1 = %d", value); /*LINE B*/
                //Equivalent to cout << "Value1 = " << value ;
                return 0;
        }
        else if (pid > 0) {
                wait(NULL);
                while(i < 3)
                {
                        printf("Value2 = %d\n", value); /* LINE A */
                        //Equivalent to cout << "Value2 = " << value << endl ;
                        value = value + 3;
                        i++ ;
                }
                return 0;
        }
}
```

```
int value = 5, i = 0;
int main()
{
        pid_t pid;
        pid = fork();

        if (pid == 0) {
                value += 15;
                printf("Value1 = %d",value); /*LINE B*/
                //Equivalent to cout << "Value1 = " << value ;
                return 0;
        }
        else if (pid > 0) {
                wait(NULL);
                while(i < 3)
                {
                        printf("Value2 = %d\n", value); /* LINE A */
                        //Equivalent to cout << "Value2 = " << value << endl ;
                        value = value + 3;
                        i++ ;
                }
                return 0;
        }
}
```

**Parent**

```
int value = 5, i = 0;
int main()
{
        pid_t pid;
        pid = fork();

        if (pid == 0) {
                value += 15;
                printf("Value1 = %d",value); /*LINE B*/
                //Equivalent to cout << "Value1 = " << value ;
                return 0;
        }
        else if (pid > 0) {
                wait(NULL);
                while(i < 3)
                {
                        printf("Value2 = %d\n", value); /* LINE A */
                        //Equivalent to cout << "Value2 = " << value << endl ;
                        value = value + 3;
                        i++ ;
                }
                return 0;
        }
}
```

**Child**

# Child Process Inherits

- Stack
- Memory
- Environment
- Open file descriptors
- Current working directory
- Resource limits
- Root directory

# Child process DOES NOT Inherit

- Process ID
- Different parent process ID
- Process times
- Own copy of files
- Resource utilization (initialized to zero)

# The wait() System Call

- The parent process will often want to wait until all child processes have been completed. This can be implemented with the wait() function call.
- **wait()**: Blocks calling process until the child process terminates. If child process has already teminated, the wait() call returns immediately.
- **waitpid()**: Options available to block calling process for a particular child process not the first one.

# The wait() System Call

- **A child program returns a value to the parent, so the parent must arrange to receive that value**
- **The wait() system call serves this purpose**
  - pid_t wait(int **status*)
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies, wait() returns one of the values immediately (and deallocates the zombie)

# Demonstrate all possible outputs.

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
        if (fork()== 0)
                printf("HC: hello from child\n");
        else
        {
                printf("HP: hello from parent\n");
                wait(NULL);
                printf("CT: child has terminated\n");
        }

        printf("Bye\n");
        return 0;
}
```

# Zombie vs. Orphan Process

- A zombie process is not the same as an orphan process. An orphan process is a process that is still executing, but whose parent has died. They do not become zombie processes; instead, they are adopted by init (process ID 1), which waits on its children.
- On Unix and Unix-like computer operating systems, a **zombie process** or **defunct process** is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the process that started the (now zombie) process to read its exit status. The term *zombie process* derives from the common definition of zombie—an undead person. In the term's colorful metaphor, the child process has *died* but has not yet been *reaped*. Also, unlike normal processes, **the kill command has no effect on a zombie process.**

# What is a zombie?

When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, at which stage the zombie is removed. The wait call may be executed in sequential code, but it is commonly executed in a handler for the SIGCHLD signal, which the parent receives whenever a child has died.

# What is a zombie?

- After the zombie is removed, its process ID and entry in the process table can then be reused. However, if a parent fails to call wait, the zombie will be left in the process table. In some situations this may be desirable, for example if the parent creates another child process it ensures that it will not be allocated the same process ID. On modern UNIX-like systems (that comply with Single Unix Specification v3 in this respect), the following special case applies: if the parent *explicitly* ignores SIGCHLD by setting its handler to SIG_IGN (rather than simply ignoring the signal by default) or has the SA_NOCLDWAIT flag set, all child exit status information will be discarded and no zombie processes will be left.

# What is a zombie?

- Since there is no memory allocated to zombie processes except for the process table entry itself, the primary concern with many zombies is not running out of memory, but rather running out of process ID numbers.

- To remove zombies from a system, the SIGCHLD signal can be sent to the parent manually, using the kill command. If the parent process still refuses to reap the zombie, the next step would be to remove the parent process. When a process loses its parent, init becomes its new parent. Init periodically executes the wait system call to reap any zombies with init as parent.

# What is a zombie?

- In the interval between the child terminating and the parent calling wait(), the child is said to be a ‘zombie’.
- Even though its not running its taking up an entry in the process table.
- The process table has a limited number of entries.

# What is a zombie?

- If the parent terminates without calling wait(), the child is adopted by init.
- The solution is:
- Ensure that your parent process calls wait() or waitpid or etc, for every child process that terminates.

# Process Termination

- Batch job issues Halt instruction
- User logs off
- Process executes a service request to terminate
- Parent terminates so child processes terminate
- Operating system intervention
  - such as when deadlock occurs
- Error and fault conditions
  - E.g. memory unavailable, protection error, arithmetic error, I/O failure, invalid instruction

# Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates
    - Cascading termination

# exit()

**void exit(int *status*);**

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead

# exec()

- **exec** is the name of a family of functions that includes **execl**, **execv**, **execle**, **execve**, **execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function.

- There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image.

- A call to any *exec* function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions or cleanup handlers shall be called.

# exec()

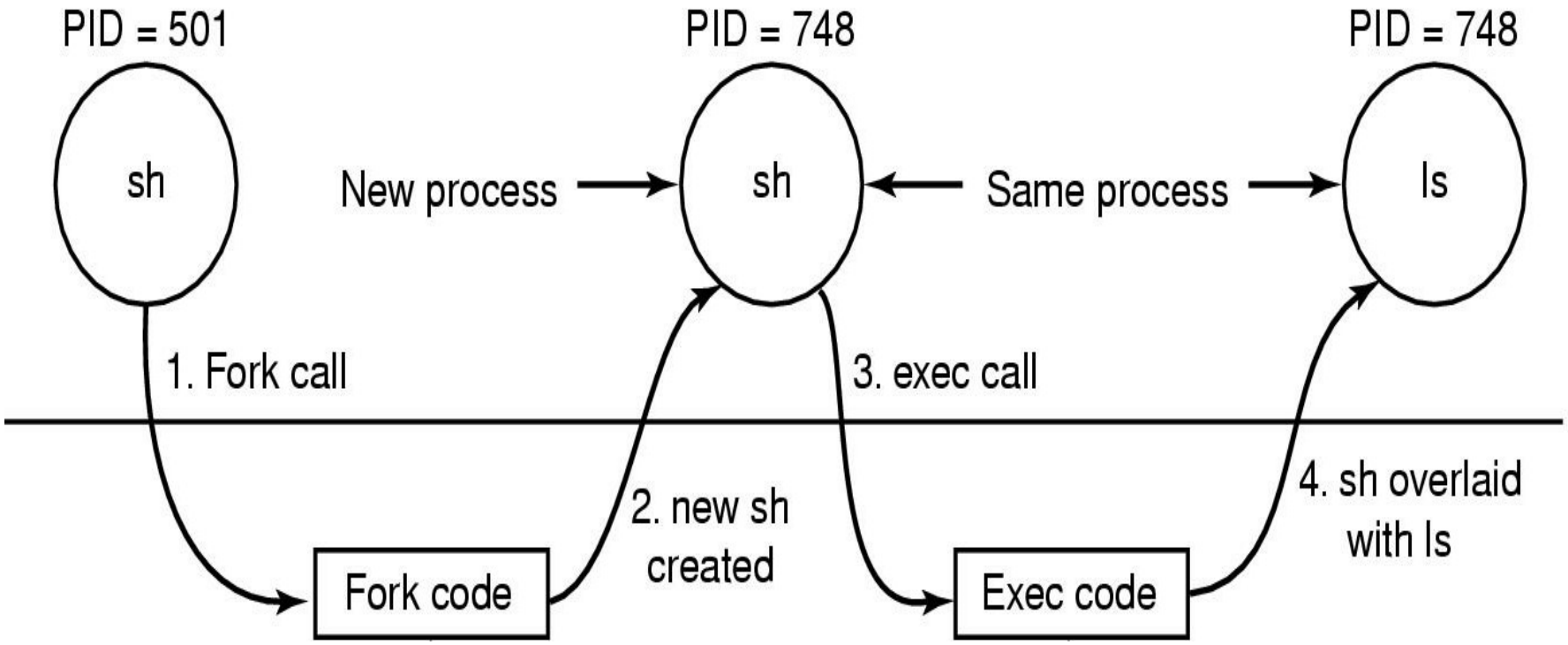
- If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be -1, and *errno* shall be set to indicate the error.

- The new process shall inherit at least the following attributes from the calling process image:
  - Process ID
  - Parent process ID
  - Process group ID
  - Session membership
  - Real user ID
  - Real group ID
  - Supplementary group IDs
  - Time left until an alarm clock signal (see alarm())
  - Current working directory
  - Root directory

# exec()

- We usually want the child process to run some other executable
- For Example, *ls - list directory contents*
- Overlays, performed by the family of exec system calls, can change the executing program, but can not create new processes.

# The *ls* Command




Steps in executing the command *ls* type to the shell

# exec()- Variations

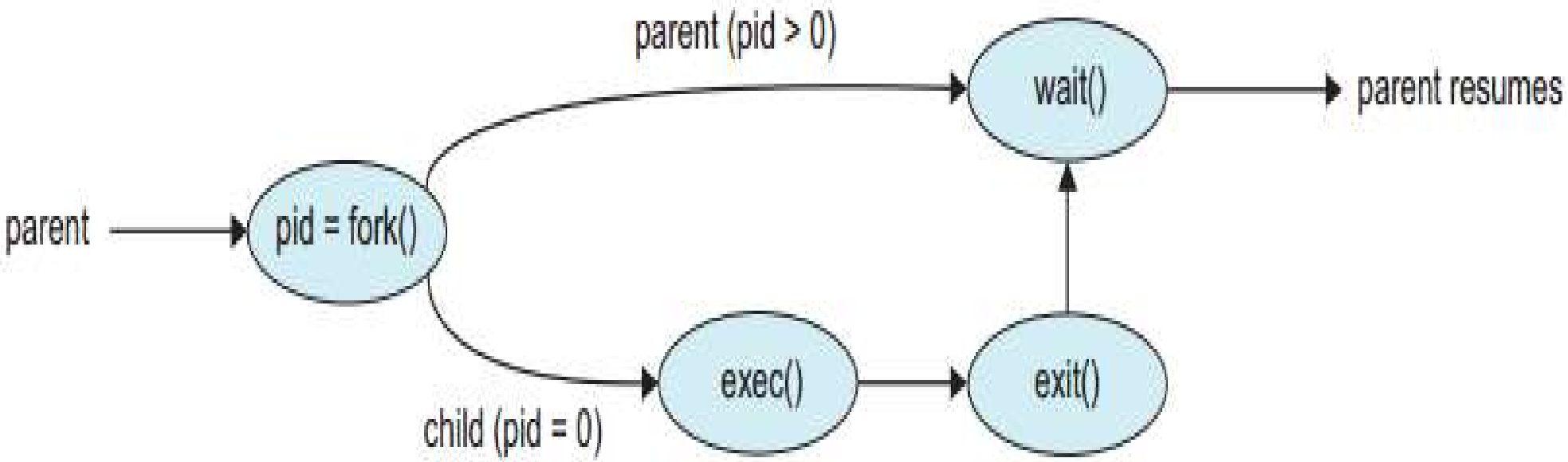
- **e:** It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.
- **l:** l is for the command line arguments passed a list to the function
- **p:** p is the path environment variable which helps to find the file passed as an argument to be loaded into process.
- **v:** v is for the command line arguments. These are passed as an array of pointers to the function.

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[])
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

# C Program Forking Separate Process

```
#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



# Example--  Fork()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
pid_t new_pid;
new_pid = fork();
switch(new_pid)
{
case -1 : /* Error */
printf("Error");
break;
case 0 : /* I am child */
printf("Child");
break;
default : /* I am parent */
printf("Parent");
break;
}
}
```

# Example--  Fork()

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
 cout<<"A\n";
 pid_t p1 = fork();
 if( p1 == 0 )
 {
            cout<<"B\n";
            fork();
 }
 else
 {
            cout<<"C\n";
 }
 cout<<"D\n";
return 0;
}
```

# Example--  exec

```
#include <stdio.h>
#include <unistd.h>
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
for(int i=1; i < argc; i++)
        {
        if(fork()==0)
                {
          execlp(argv[i],argv[i],NULL);
       cout<<"exec complete<<endl";
                }
        }
return 0;
}
```

# Example-- fork, exec, wait

```
using namespace std;
int main()
{
char *argv[10];
cout<<"Enter filename: ";
argv[0]="gedit";
cin>>argv[1];
argv[2]=NULL;
pid_t = pid;
int status;
if((pid = fork()<0)// forking child process
 {
            printf(""forking child process failed\n");
            exit(1);
 }
 else if(pid == 0)
 {
            execvp(*argv,argv);
 }
else
 {
      while(wait(&status)!=pid);// parent process wait for completion of child process
completion
 }
}
```

# Example-- fork, exec, wait

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for
it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an
error occurs.  */
    fprintf (stderr, "an error occurred in
execvp\n");
    abort ();
  }
}
```

```
int main ()
{
  /* The argument list to pass to the "ls"
command.  */
  char* arg_list[] = {
    "ls",    /* argv[0], the name of the
program.  */
    "-l",
    "/",
    NULL     /* The argument list must end
with a NULL.  */
  };

  /* Spawn a child process running the "ls"
command.  Ignore the
     returned child process id.  */
  spawn ("ls", arg_list);

  printf ("done with main program\n");

  return 0;
}
```

# Example-- fork, exec, wait, Exit

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char *argv[], char *env[] )
{
  pid_t my_pid, parent_pid, child_pid;
  int status;

/* get and print my pid and my parent's pid. */

  my_pid = getpid();    parent_pid = getppid();
  printf("\n Parent: my pid is %d\n\n", my_pid);
  printf("Parent: my parent's pid is %d\n\n", parent_pid);

/* print error message if fork() fails */
  if((child_pid = fork()) < 0 )
  {
    perror("fork failure");
    exit(1);
  }

/* fork() == 0 for child process */

  if(child_pid == 0)
  {  printf("\nChild: I am a new-born process!\n\n");
    my_pid = getpid();    parent_pid = getppid();
    printf("Child: my pid is: %d\n\n", my_pid);
    printf("Child: my parent's pid is: %d\n\n", parent_pid);
    printf("Child: I will sleep 3 seconds and then execute - date - command \n\n");

    sleep(3);
    printf("Child: Now, I woke up and am executing date command \n\n");
    execl("/bin/date", "date", 0, 0);
    perror("execl() failure!\n\n");

    printf("This print is after execl() and should not have been executed if execl were
successful! \n\n");

    _exit(1);
  }
/*
 * parent process
 */
  else
  {
    printf("\nParent: I created a child process.\n\n");
    printf("Parent: my child's pid is: %d\n\n", child_pid);
    system("ps -acefl | grep ercal");  printf("\n \n");
    wait(&status); /* can use wait(NULL) since exit status
                from child is not used. */
    printf("\n Parent: my child is dead. I am going to leave.\n \n ");
  }

  return 0;
}
```

# Example of fork( )

```
int main(int argc, char **argv)
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s sees PID of %d\n",
                name, child_pid);
        return 0;
    } else {
        printf("I am the parent %s. My child is %d\n", name, child_pid);
        return 0;
    }
}
```

```
% ./forktest
Child of forktest sees PID of 0
I am the parent forktest. My child is 486
```

# Fork + Exec – shell-like

```
int main(int argc, char **argv)
{  char *argvNew[5];
   int pid;
   if ((pid = fork()) < 0) {
      printf( "Fork error\n“);
      exit(1);
   } else if (pid == 0) { /* child process */
         argvNew[0] = "/bin/ls";
         argvNew[1] = "-l";
         argvNew[2] = NULL;
         if (execve(argvNew[0], argvNew, environ) < 0) {
            printf( "Execve error\n“);
            exit(1);
         }
   } else { /* parent */
          wait(pid); /* wait for the child to finish */
   }
}
```

# References


- Operating System Concepts (Silberschatz, 8th edition) Chapter 3
- http://siber.cankaya.edu.tr/OperatingSystems/ceng328/node87.html
- http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/processControl.htm
- http://www.tutorialspoint.com/operating_system/os_processes.htm
- http://wiki.answers.com/Q/Explain_process_control_block
- http://www.gitam.edu/eresource/comp/gvr(os)/4.2.htm
- http://www.sal.ksu.edu/faculty/tim/ossg/Process/p_create.html