Process Types

- Sometimes OS may swap a blocked process to disk to free up more memory
- Or to improve process mix
- This is called ***Swapping***

Reference

- 10.3.2 Process Management System Calls in UNIX
- Modern Operating System
  - 2nd Edition, Andrew S. Tanenbaum

39

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

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

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.

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

Compiling C ++ code

- g++ test.cpp –o Output
- Running the code:
- ./Output

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

Unix Process Control

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

      x = x + 1;
      pid = fork();
      x = 3;
      printf(“%d”,x);
}
```

But we want the child process to do something else…

```
int pid;
int status = 0;

if (pid = fork()) {
      /* 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 status.

***Wait** variants allow wait on a specific child or…*

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

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

Child Process Inherits

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

Child process DOESNOT Inherit

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

How can a parent and child process communicate?

- Through any of the normal IPC mechanism schemes.
- But have special ways to communicate
- For example
  - The variables are replicas
  - The parent receives the exit status of the child

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)

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.

exit()

void exit(int *status*);

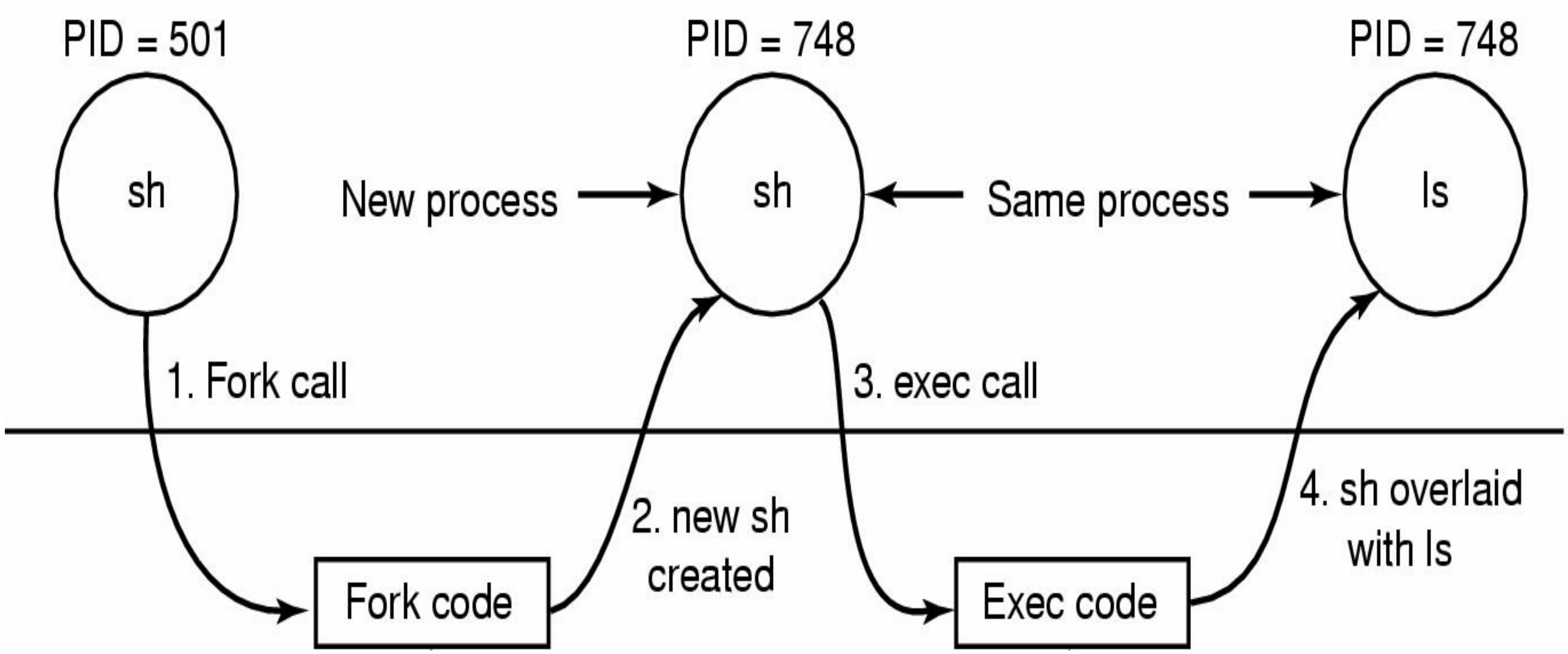
- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the “result” of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - 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

execv()

- We usually want the child process to run some other executable
- For Example, *ls*

55

The *ls* Command



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

execv

- int execv(const char *path, char *const argv[]);
- the current process image with a new process image.
- ***path*** is the filename to be executed by the child process
- When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:
  - int main (int argc, char *argv[]);
- The ***argv*** array is terminated by a null pointer.
- The null pointer terminating the ***argv*** array is not counted in ***argc***.

```
while (TRUE) {                              /* repeat forever /*/
    type_prompt( );                         /* display prompt on the screen */
    read_command(command, params);          /* read input line from keyboard */

    pid = fork( );                          /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);           /* error condition */
        continue;                           /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);           /* parent waits for child */
    } else {
        execve(command, params, 0);         /* child does the work */
    }
}
```