L9: Operations on Processes

Dr. Rajashree Dash

Associate Professor,
Department of CSE,
ITER, Siksha O Anusandhan Deemed to be University

Outline

Operation on processes

- The processes in the system can execute concurrently and must be created and deleted dynamically.
- Thus the operating system must provide mechanisms for:
  - process creation
  - process termination

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).
- Resource sharing options:
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no .
- Address space:
  - Child is duplicate of parent.
  - Child has a program loaded into it.
- Execution options
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

Process creation: UNIX example

- fork() system call creates new process.
- exec() system call used after a fork() to replace the process' memory space with a new program.

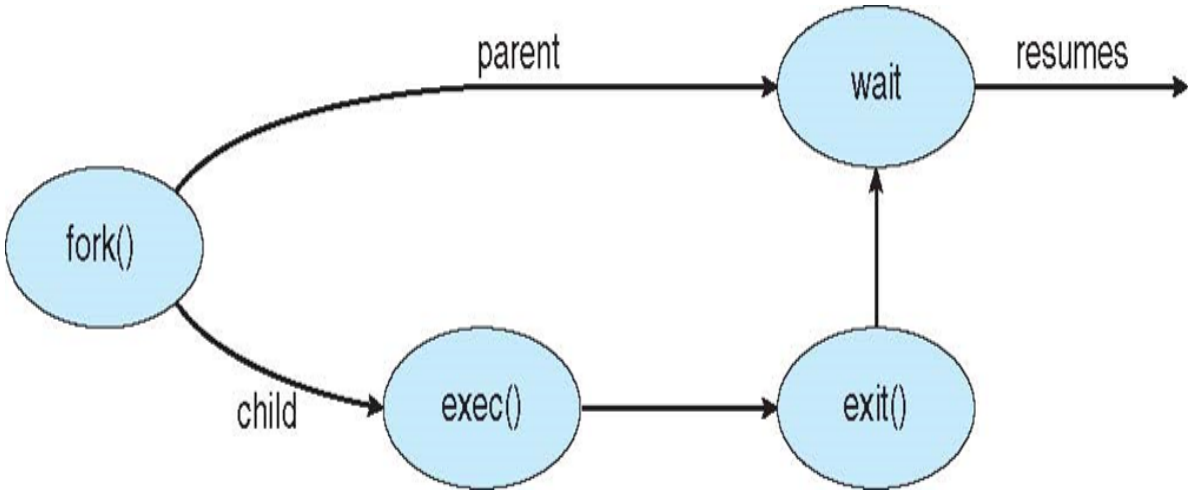

Figure: Process creation using fork() system call

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.
- 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 systems does not allow a child to continue if its parent terminates. Then all its children must also be terminated.

Process termination

- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon is referred to as cascading termination.
- UNIX example
  - A process is terminated by using the exit() system call, providing an exit status as a parameter:
    exit(1);
  - Under normal termination, exit() may be called either directly or indirectly (by a return statement in main()).
  - A parent process may wait for the termination of a child process by using the wait() system call.

Process termination

- The wait() system call passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated.
- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

Process termination

- A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
- If a parent did not invoke wait() and instead terminated, child processes became orphans. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Process creation based question

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

int main(void) {
  int x;
  x = 0;
  printf("I am process %ld and my x is %d\n", (long)getpid(), x);
  fork();
  x = 1;
  printf("I am process %ld and my x is %d\n", (long)getpid(), x);
return 0;
}
```

Process creation based question

```
#include <stdio.h>
#include <unistd.h>
Int value =5;
int main(void) {
pid_t pid;
pid = fork();
if (pid == 0)
{
value+=15;
return 0;
}
else if (pid>0)
{
wait(NULL);
printf("parent: value = %d",value);
return 0;
}
}
```

Process creation based question

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
 fork();
fork();
fork();
 return 0;
}
```

How many processes are created?

Process creation based question

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
int i;
pid_t cpid;
for (i=0;i<4;i++)
 {
        cpid=fork();
        if(cpid !=0)          // if(cpid==0)
        break;
}

printf(" process\t %d child \t %d\n", getpid(), cpid);
return 0;
}
```

How many processes are created?