Preview

- Process Termination
- Zombie Process
- wait() and waitpid() System Call
- Orphan Process
- exec System Calls
  - execl()
  - execv()
  - execle()
  - execve()
  - execlp()
  - execvp()

Process Termination

- No matter how a process terminated normally or abnormally, kernel execute a code closes all the open descriptors, release the memory used and so on.
- When a process terminated, the parents can obtain child's status from either the wait() or the waitpid() system call.
- If a parent terminates before the child, **systemd**(**init** in Unix)process becomes the parent process of any process whose parent terminated.

Process Termination

- When a parent process terminate, the parent process ID of the surviving process is changed to be 1 (Guaranteed every process has a parent)
  - Process ID = 0 : scheduler process
  - Process ID = 1 : systemd (init in Unix) process
- If a child terminate before parent, the kernel save a child's information (ID, termination status, CPU time) for the parent process termination.

Process Termination

- When a child process finishes execution, it will have an exit status to report to its parent process.
- Because of this last little bit of information, the process will remain in the operating system's process table as a ***zombie process***.
- A **zombie process** will not to be scheduled for further execution, but that it cannot be completely removed
- ps –el |grep 'Z' can prints the status of zombie.

Process Termination

- When a child exits, its parent is supposed to use the "**wait**" system call and collect the child process's exit information.
- The subprocess exists as a zombie process until this happens.
- However, if the parent process isn't programmed properly or has a bug and never calls "wait," the zombie process remains, eternally waiting for its information to be collected by its parent.

Process Termination

```
/* zombie.c: create a zombie process */
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>

int   main(void)
{
    int pid;
     pid=fork();
              /* parent sleep 100 sec */
     if(pid>0)
     {
                    sleep(100);
     }
          /* a child terminate parent don't have time to save a child info */
          /* child remains as a zombie */
      else
     {
                    exit(0);
     }
}
```

Process Termination

```c
/* zombie0.c: create a zombie process */
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>
int  main(void)
{
    int pid, pid1;
     pid=fork(); //create the first child
     if(pid>0)
     {
          pid1=fork(); //create the second child
          if (pid1 >0) //parent runs forever
          {
                    while (1)
                              ;
          }
          else // for second child
          {
                    _exit(0);
          }
     }
     else //for first child
     {
          _exit(0);
     }
}
```

Process Termination

- The exit status of a child will be used for parent process termination.
- When a child exits, the parent process will receive a SIGCHLD signal to indicate that one of its children has finished executing; the parent process will typically call the wait() system call at this point.
- That call will provide the parent with the child's exit status, and will cause the child to be *reaped*, or removed from the process table.

wait and waitpid() System Call

- When a process terminate either normally or abnormally, the kernel sent a signal (SIGCHD) to a parent.
- A parent can ignore the signal or call a function (wait or waitpid) to take care the signal.

```c
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid(pid_t pid, int *status, int option);
```


wait and waitpid() System Call

- The execution of **wait()** could have two possible situations.
  - If there are at least one child processes, the caller will be blocked until one of its child processes exits.
  - If there is no child process running, then this **wait()** has no effect at all.
- The status is the pointer where terminated process's status is saved.

Process Termination

```c
/* zombie01.c: create a zombie process */
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>
int  main(void)
{
    int pid, pid1;
     pid=fork(); //create the first child
     if(pid>0)
     {
          pid1=fork(); //create the second child
          if (pid1 >0) //parent runs forever
          {
                    wait();
                    while (1)
                       ;
          }
          else // for second child
          {
                    _exit(0);
          }
     }
     else //for first child
     {
          _exit(0);
     }
}
```

wait and waitpid() System Call

- By using macros in <sys/wait.h>, we can check a terminated process's status.
- The status field will be filled in by wait or waitpid function.

```c
#include <sys/wait.h>
int WIFEXITED(int status);
                    if child process terminate normally, return true

int WIFSIGNALED(int status);

                  if child process terminate abnormally, return true
```

```c
/*wait.c  demonstrate wait() system call */
#include  <stdio.h>
#include  <string.h>
#include  <sys/wait.h>
#include <unistd.h>
#include  <sys/types.h>
#define   MAX_COUNT  1000
#define   BUF_SIZE   100

void  main(void)
{
    pid_t   pid1, pid,  p1;
    int     status;
    int     i;
    char    buf[BUF_SIZE];
    printf("*** Parent is about to fork  ***\n");
    if ((pid1 = fork()) < 0)
    {
            printf("Failed to fork process 1\n");
            exit(1);
    }
    else if (pid1 == 0) /* child's part */
    {
        p1= getpid();
        for (i = 1; i <MAX_COUNT; i++)
        {
            sprintf(buf,"child %d is running\n", p1);
            write(1, buf, strlen(buf));
        }
        _exit(0);
    }
    else /* parent's part */
    {
            sprintf(buf, "*** Parent enters waiting status .....\n");
            write(1, buf, strlen(buf));
            pid = wait(&status); /*wait for child finish it's job */
            if (WIFEXITED(status))
                printf(" A child process terminate normally\n");
             else
                printf(" A child process terminate abnormally\n");
            sprintf(buf, "*** Parent detects process %d was done ***\n", pid);
            write(1, buf, strlen(buf));
            printf("*** Parent exits ***\n");
            exit(0);
    }
}
```

Orphan Process

- An **Orphan Process** is nearly the same thing which we see in real world.
- Orphan means someone whose parents are dead.
- The same way this is a process, whose parents are dead, that means parents are either terminated, killed or exited but the child process is still alive.


```c
/* orphan.c shows  example orphan call */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

int main (int argc, const char *argv[])
{
    pid_t pid;
    char buf[100];
    if ( (pid = fork()) < 0)  /* create child */
    {
        printf("fork error\n");
        exit (1);
    }
    if (pid == 0) /* child process */
    {
        while (1)
        {
            sprintf(buf, "*** Child %d is running. Its parent is %d .....\n", getpid(), getppid());
            write(1, buf, strlen(buf));
            sleep(1);
            if (getppid() == 1)
                printf("My parent is passed away. Now, I am a orphan with step parent %d\n", getppid());
        }
    }
    else /* parent process */
    {
       while (1)
       {
            sprintf(buf, "***I %d  am still alive\n", getpid());
            write(1, buf, strlen(buf));
            sleep(1);
       }
    }

    exit (0);
}
```

exec System Call

- By using exec system call, a child process can execute another program.
- Once a process call a exec system call, that process is completely replaced by the new program.
- The new program starts executing at its main function. The main function might need arguments.
- The process ID does not change across an exec system call, since it is not created.
- The content of text, data, heap and stack segment will be replaced by new program.

exec System Call

```c
#include <unistd.h>
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*, (char *)0, char
    *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);


Return -1 on error, no return on success
```


exec System Call

- Six system call can be recognized by
  - Argument list or Argument vector
  - File name or path name
  - With or without environment

exec System Call

- If *filename* contains a slash, it is consider as a pathname.
- Otherwise, the executable file is searched for the directory specified PATH environment variable.
- If a file name find out but not executable, then it is consider as shell script and tries to invoke /bin/sh.
- With execle and execve, environment variable can be passed to the function.

exec System Call

- Normally, a process allows its environment to be propagated to its children.
- But some cases, a process need to specify a certain environment for a child.

  Ex) the login program need create different environment for each user's login.


```c
/* execex.c shows execv system call */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

int main (int argc, const char *argv[])
{
    pid_t pid;
    int mult = 1, i;
    if (argc == 1)
    {
        printf("argument error \n");
        exit (1);
    }
    if ( (pid = fork()) < 0)  /* create child */
    {
         printf("fork error\n");
         exit (1);
    }
    else if (pid == 0)
    {
        /* a child execute different program */
        if (execv ("/home/separk/Lecture/cosc350/example/ch4/summation", argv) <0)
        {
            printf ("execl ERROR");
            exit (1);
        }
    }

    for (i = 1; i <argc; i++)
          mult *= atoi(argv[i]);
    printf ("The multiplication of arguments is %d \n", mult);
    exit (0);
}
```