UNIT-IV UNIX PROCESS

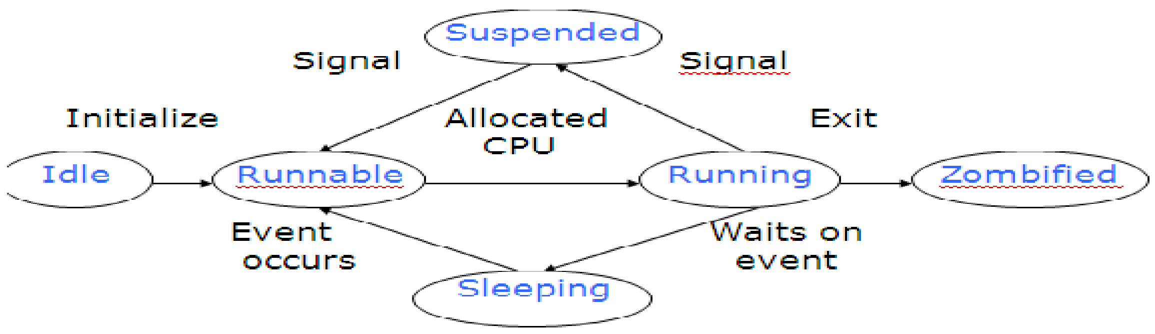
• PROCESS MANAGEMENT

- Every process in a UNIX system has the following attributes:
  - some code(a.k.a. text)
  - some data
  - a stack
  - a unique process ID number(PID)
- When UNIX is first started, there's only one visible process in the system. This process is called "init", and it has a process ID of 1.
- The only way to create a new process in UNIX is to duplicate an existing process, so "init" is the ancestor of all subsequent processes.

PROCESS STATES

- Every process in the system can be in one of six states. The six possible states are as follows:

1) ***Running***, which means that the process is currently using the CPU.
2) ***Runnable***, which means that the process can make use of the CPU as soon as it becomes available.
3) ***Sleeping***, which means that the process is waiting for an event to occur. For example, if a process executes a "read()" system call, it sleeps until the I/O request completes.
4) ***Suspended***, which means that the process has been "frozen" by a signal such as SIGSTOP. It will resume only when sent a SIGCONT signal. For example, a Control-Z from the keyboard suspends all of the processes in the foreground job.
5) ***Idle***, which means that the process is being created by a "fork() system call and is not yet runnable.
6) ***Zombified***, which means that the process has terminated but has not yet returned its exit code to its parent. A process remains a zombie until its parent accepts its return code using the "wait()" system call.



[*Process States*]

PROCESS COMPOSITION

- Every process is composed of several different pieces:
  - ***a code area***, which contains the executable(text) portion of a process

- ***a data area***, which is used by a process to contain static data
- ***a stack area***, which is used by a process to store temporary data
- ***a user area,*** which holds housekeeping information about a process
- ***page tables,*** which are used by the memory management system

User Area

Every process in the system has some associated "housekeeping" information that is used by the kernel for process management. This information is stored in a data structure called a user area. Every process has its own user area. User areas are created in the kernel's data region and are only accessible by the kernel; user processes may not access their user areas.

Fields within a process' user area include:

- a record of how the process should react to each kind of signal
- a record of the process' open file descriptors
- a record of how much CPU time the process has used recently

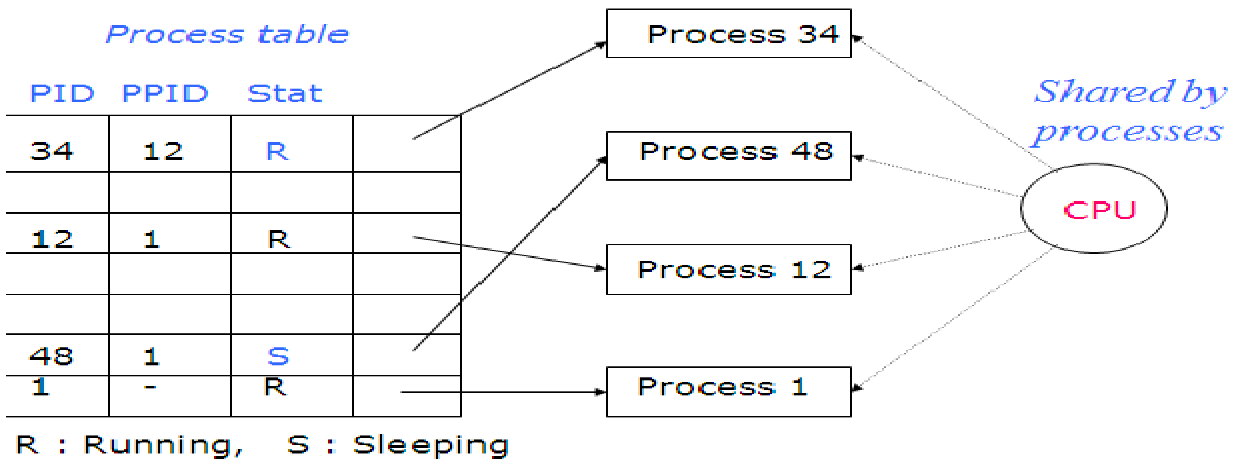
The Process Table

- There is a single kernel data structure of fixed size called the process table that contains one entry for every process in the system.

The process table is created in the kernel's data region and is accessible only by the kernel.

Each entry contains the following information about each process:

- its process ID(PID) and parent process ID(PPID)
- its real and effective user ID(UID) and group ID(GID)
- its state (running, runnable, sleeping, suspended, idle, or zombified)
- the location of its code, data, stack, and user areas
- a list of all pending signals

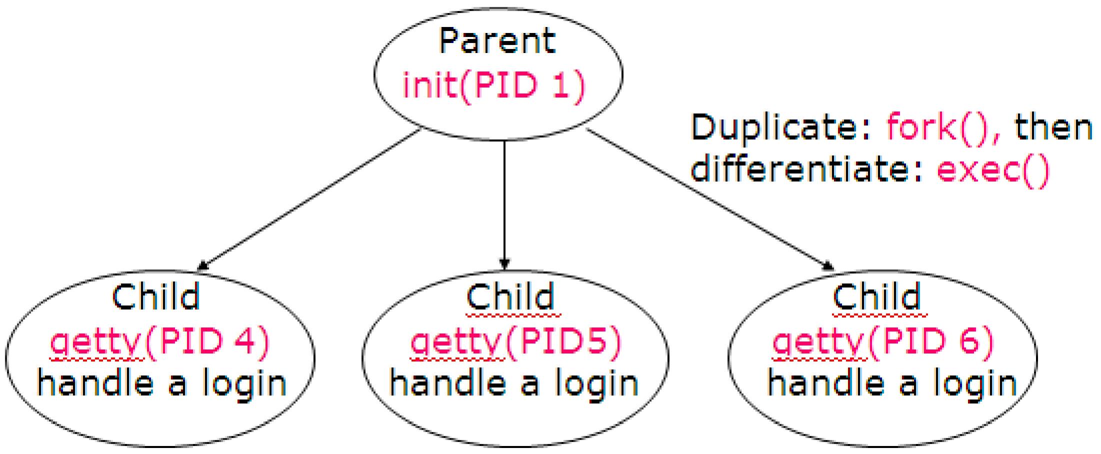


• PROCESS MANAGEMENT

- When a process duplicates, the parent and child processes are virtually identical (except for aspects like PIDs, PPIDs, and runtimes); the child's code, data, and stack are a copy of the parent's, and the processes even continue to execute the same code.

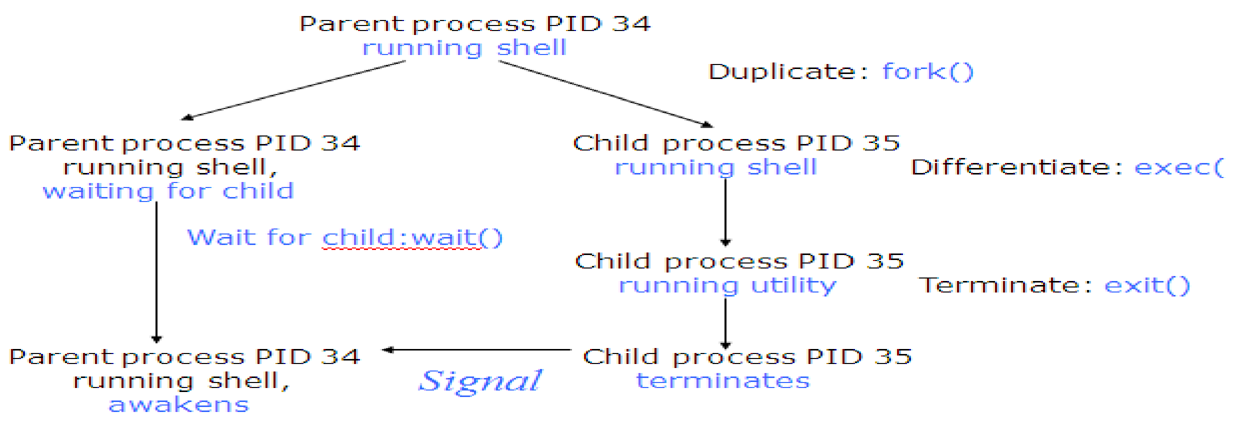
- A child process may replace its code with that of another executable file, there by differentiating itself from its parent.
- When "init" starts executing, it quickly duplicates several times. Each of the duplicate child processes then replaces its code from the executable file called "getty",which is responsible for handling user logins.

- The process hierarchy



Process hierarchy

- When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action.
- When a shell executes a utility in the foreground, it duplicates into two shell processes;
- the child-shell process replaces its code with that of the child utility, whereas the parent shell waits for the child process to terminate.
- When the child process terminates, the original parent process awakens and presents the user with the next shell prompt.



How a shell runs a utility

PROCESS MANAGEMENT SYSTEM CALLS:

Name	Function
fork	duplicates a process
getpid	obtains a process' ID number
getppid	obtains a parent process'ID number
exit	terminates a process
wait	waits for a child process
exec	replaces the code, data, and stack of a process.

Creating a New Process: fork()

- A process may duplicate itself by using "fork()", which works like this:
  System Call: pid_t **fork**(void)
  "fork()" causes a process to duplicate.
  The child process is an almost-exact duplicate of the original parent process; it inherits a copy of its parent's code, data, stack, open file descriptors,and signal table.
  the parent and child processes have different process ID numbers and parent process ID numbers.
  If "fork()" succeeds, it returns the PID of the child to the parent process and returns a value of 0 to the child process.
- A process may obtain its own process ID and parent process ID numbers by using the "getpid()" and "getppid()" system calls, respectively.
- Here's a synopsis of these system calls:

```
System Call :  pid_t   getpid(void)
               pid_t   getppid(void)
```

"getpid()" and "getppid()" return a process'ID number and parent process' ID number, respectively.
The parent process ID number of PID 1 (i.e., "init") is 1.

cat myfork.c ---> list the program.

```
 #include <stdio.h>
 main()
 {
    int  pid;
    printf("I'm the original process with PID %d and PPID %d. \n",  getpid(), getppid() );
    pid = fork();   /* Duplicate.  Child and parent continue from here */
    if ( pid!= 0 )  /* pid is non-zero, so I must be the parent */
      {
        printf("I'm the parent process with PID %d and PPID %d. \n",  getpid(), getppid() );
        printf(“My child’s PID is %d \n”, pid );
      }
   else  /* pid is zero, so I must be the child */
      {
        printf(“I’m the child process with PID %d and PPID %d. \n”, getpid(),
getppid() );
      }
      printf(“PID %d terminates. \n”, getpid() );  /* Both processes */
      /* execute this */
   }
```

```
 $ myfork          ---> run the program.
 I’m the original process with PID 13292 and PPID 13273.
 I’m the parent process with PID 13292 and PPID 13273.
 My child’s PID is 13293.
 I’m the child process with PID 13293 and PPID 13292.
 PID 13293 terminates.  ---> child terminates.
 PID 13292 terminates.  ---> parent terminates.
```

- The PPID of the parent process refers to the PID of the shell that executed the “myfork” program.

WARNING:

it is dangerous for a parent process to terminate without waiting for the death of its child.

The only reason our program doesn’t wait for its child to terminate is because we haven’t yet described the “wait()” system call!.

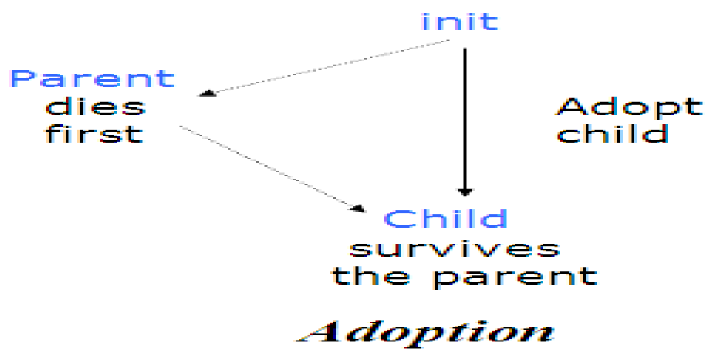
Orphan Processes:

- If a parent dies before its child terminates, the child is automatically adopted by the original “init” process, PID 1.
- the parent process terminated before the child did.

```
  $ cat  orphan.c       ---> list the program.
  #include <stdio.h>
  main()
  {
    int pid;
    printf(“I’m the original process with PID %d and PPID %d. \n”, getpid(),
getppid() );
    pid = fork();  /* Duplicate. Child and parent continue from here */
    if ( pid!= 0 )  /* Branch based on return value from fork() */
     {
       /* pid is non-zero, so I must be the parent */
       printf(“I’m the parent process with PID %d and PPID %d. \n”, getpid(),
getppid() );
       printf(“My child’s PID is %d \n”, pid );
```

```
      }
    else
      {
        /* pid is zero, so I must be the child */
        sleep(5);   /* Make sure that the parent terminates first */
        printf("I'm the child process with PID %d and PPID %d. \n", getpid(),
getppid() );
      }
    printf("PID %d terminates. \n", getpid() );  /* Both processes */
    /* execute this */
 }
$ orphan    ---> run the program.
I'm the original process with PID 13364 and PPID 13346.
I'm the parent process with PID 13364 and PPID 13346.
PID 13364 terminates.
 I'm the child process with PID 13365 and PPID 1. ---> orphaned!
PID 13365 terminates.
```

- illustration of the orphaning effect:



Terminating a Process: exit()

- A process may terminate at any time by executing "exit()", which works as follows:

System Call: void **exit(** int status)

"exit()" closes all of a process' file descriptors; deallocates its code, data, and stack; and then terminates the process.

When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code status to be accepted.

A process that is waiting for its parent to accept its return code is called a *zombie* process.

A parent accepts a child's termination code by executing "wait()".

Orphan Processes

- The kernel ensures that all of a terminating process' children are orphaned and adopted by "init" by setting their PPID to 1.

The “init” process always accept its childrens’ termination codes.
“exit()” never returns.
- The termination code of a child process may be used for a variety of purposes by the parent process.Shells may access the termination code of their last child process via one of their special variables. bash stores the termination code of the last command in the variable $?:

```
$ cat  myexit.c   --->  list the program.
 #include <stdio.h>
 main()
 {
   printf(“I’m going to exit with return code 42 \n”);
   exit(42);
 }
 $ myexit   ---> run the program.
 I’m going to exit with return code 42
 $ echo  $?   ---> display the termination code.
 42
 $ _
```

Zombie Processes

- A process that terminates cannot leave the system until its parent accepts its return code.
- If its parent process is already dead,it’ll already have been adopted by the “init” process, which always accepts its childrens’ return codes.
- If a process’ parent is alive,but the parent never executes a “wait()” system call, the process’ return code will never be accepted and the process will remain a zombie.
- A zombie process doesn’t have any code, data, or stack, so it doesn’t use up many system resources,
- it does continue to inhabit the system’s fixed-size process table.
- The next program created a zombie process,which is shown in the output from the **ps** utility.

```
 $ cat zombie.c       ---> list the program.
 #include <stdio.h>
 main()
 {
   int pid;
   pid = fork();  /* Duplicate */
   if ( pid!= 0 )  /* Branch based on return value from fork() */
     {
        while (1) /* Never terminate, and never execute a wait() */
          sleep(1000);
     }
   else
     {
        exit(42);  /* Exit with a silly number */
      }
  }
$ zombie &    ---> execute the program in the background.
 [1] 13545
 $ ps              ---> obtain process status.
   PID      TT     STAT         TIME       COMMAND
 13535     p2       S           0:00        -ksh (ksh)    ---> the shell.
 13545     p2       S           0:00        zombie       ---> the parent process.
 13546     p2       Z           0:00        <defunct>   ---> the zombile child.
 13547     p2       R           0:00        ps
 $ kill  13545    ---> kill the parent process.
 [1]  Terminated       zombie
 $ ps                ---> notice that the zombie is gone now.
   PID      TT     STAT         TIME      COMMAND
  13535   p2       S           0:00       -ksh( ksh )
  13548   p2       R           0:00       ps
```

Waiting for a child(): wait()
 - A parent process may wait for one of its children to terminate and then accept its child's termination code by executing "wait()".

System Call : pid_t **wait**(int* status)
"wait()" causes a process to suspend until one of its children terminates.
A successful call to "wait()" returns the PID of the child that terminated and places a status code into status that is encoded as follows:
- If the rightmost byte of status is zero,the leftmost byte contains the low eight bits of the value returned by the child's "exit()" or "return()" system call.
- If the rightmost byte is nonzero, the rightmost seven bits are equal to the number of the signal
 that caused the child to terminate, and the remaining bit of the rightmost byte is set to 1 if the child produced a core dump.

 - If a process executes a "wait()" system call and has no children, "wait()" returns immediately with a value of -1.
 - If a process executes a "wait()" system call and one or more of its children are already zombies, "wait()" returns immediately with the status of one of the zombies.

```
$ cat mywait.c                ---> list the program.
 #include <stdio.h>
 main()
 {
   int pid, status, childPid;
   printf("I'm the parent process and my PID is %d\n", getpid() );
   pid = fork();  /* Duplicate */
   if ( pid!=0 ) /* Branch based on return value from fork() */
    {
      printf("I'm the parent process with PID %d and PPID %d\n", getpid(),
getppid() );
      childPid = wait( &status );  /* Wait for a child to terminate. */
      printf(" A child with PID %d terminated with exit code %d \n", childPid,
status >> 8 );
    }
   else
    {  printf("I'm the child process with PID %d and PPID %d \n", getpid(),
getppid() );
       exit(42);  /* Exit with a silly number */
     }
   printf("PID %d terminates \n", getpid() );
 }
$ mywait          ---> run the program.
 I'm the parent process with PID 13465 and PPID 13464
 I'm the parent process with PID 13464 and PPID 13409
 A child with PID 13465 terminated with exit code 42
 PID 13465 terminates
```

Differentiating a Process: exec()

- A process may replace its current code, data, and stack with those of another executable by using one of the "exec()" family of system calls.
- When a process executes an "exec()" system call, its PID and PPID numbers stay the same - only the code that the process is executing changes.

System Call: int **execl**(const char* path, const char* arg0,
const char* arg1,…, const char* argn, NULL)
int **execv**(const char* path, const char* argv[])
int **execlp**(const char* path, const char* arg0,
const char* arg1, …, const char* argn, NULL)
int **execvp**(const char* path, const char* argv[])

The "exec()" family of system calls replaces the calling process' code, data, and stack with those of the executable whose pathname is stored in *path*.

- "execlp()" and "execvp()" use the $PATH environment variable to find *path*.
- If the executable is not found, the system call returns a value of -1; otherwise, the calling process replaces its code, data, and stack with those of the executable and starts to execute the new code.

- “execl()” and “execlp()” invoke the executable with the string arguments pointed to by arg1 through argn. arg0 must be the name of the executable file itself, and the list of arguments must be null terminated.
- “execv()” and “execvp()” invoke the executable with the string arguments pointed to by argv[1] to argv[n],where argv[n+1] is NULL.
argv[0] must be the name of the executable file itself.
- In the next example, the program displays a small message and then replaces its code with that of the “ls”.

```
$ cat myexec.c       ---> list the program.
#include <stdio.h>
main()
{
   printf(“I’m process %d and I’m about to exec an ls -l \n”, getpid() );
   execl( “/bin/ls”, “ls”, “-l”, NULL );  /* Execute ls */
   printf(“This line should never be executed \n”);
}
$ myexec             --->  run the program.
I’m process 13623 and I’m about to exec an ls -l
total 125
-rw-r--r--    1  glass          277   Feb   15  00:47   myexec.c
-rwxr-xr-x    1  glass        24576  Feb   15  00:48   myexec
```