CSE333 LAB

Processes in Unix

Sanem Arslan Yılmaz

Zuhal Altuntaş

[Adopted from UNIX SYSTEMS Programming: Communication, Concurrency, and Threads]

Process Identification

- A process is an instance of a program that is executing.
- UNIX identifies processes by a unique integer value called the process ID (pid)
  - Each process also has a parent process ID, which is initially the process ID of the process that created it.

- The **getpid()** and **getppid()** functions return the process ID and parent process ID

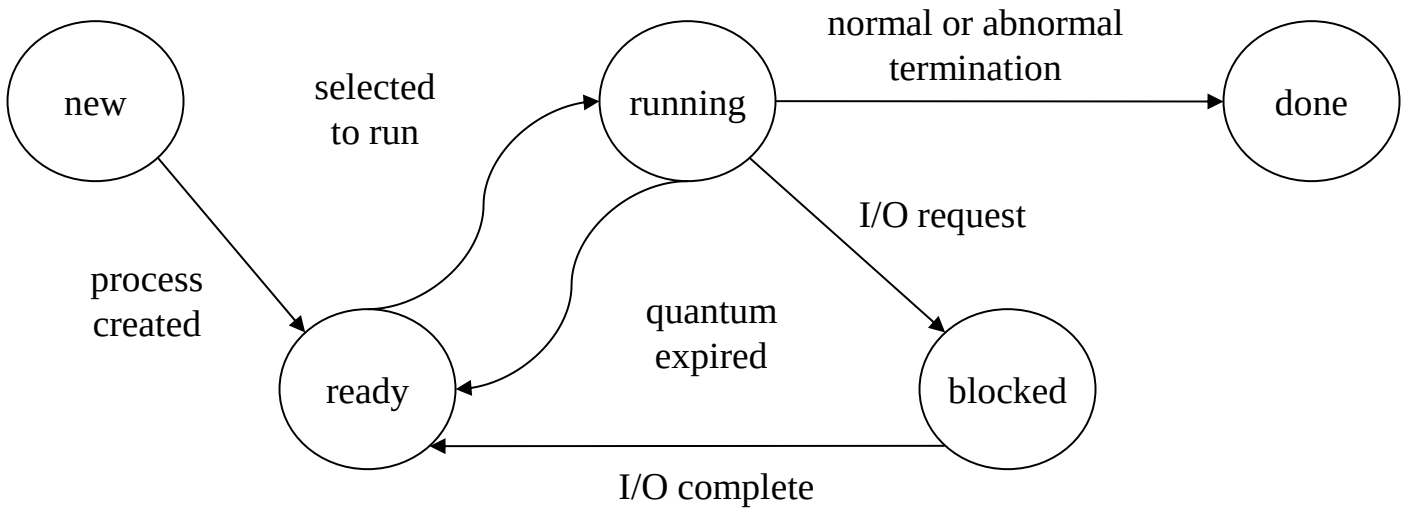
```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- Example use:

```
printf("My process ID is %ld\n", getpid());
```

Process State

- The state of a process indicates its status at a particular time
- Most operating systems use the following process states:
  - **New**: being created
  - **Running**: instructions are being executed
  - **Blocked**: waiting for an event such as I/O
  - **Ready**: waiting to be assigned to a processor
  - **Done**: finished execution (terminated)



ps Utility

- The **ps (process status)** utility displays information about processes currently handled by the OS
- The **ps** utility is executed at a UNIX shell prompt; by default it displays information about processes for the current user.
- Column headings

```
UID   - user ID                STIME - starting time of the process
PID   - process ID              TTY   - controlling terminal
PPID  - parent process ID       TIME  - cumulative execution time
C     - (obsolete)              CMD   - command name
```

- Example output produced by **ps -ef**

```
   UID    PID  PPID  C    STIME TTY      TIME CMD
  root      0     0  0   May 20 ?        0:02 sched
  root      1     0  0   May 20 ?        2:38 /etc/init -
  root      2     0  0   May 20 ?        0:00 pageout
  root      3     0  1   May 20 ?      173:15 fsflush
  root    433     1  0   May 20 console  0:00 /usr/bin/login
  root 13259   436  0 13:54:49 ?        0:00 /usr/lib/ssh/sshd
jjt107 13603 13261  0 13:55:29 pts/3    0:00 ps -ef
  root  2017  2398  0   May 22 pts/5    0:00 sh
  root  4210  5144  0   May 20 pts/1    0:00 sh
   dan 28527 28500  0   May 22 pts/6    0:00 -sh
jjt107 13261 13259  0 13:54:59 pts/3    0:00 -csh
  root 28499   436  0   May 22 ?        0:00 /usr/lib/ssh/sshd
  root  3110   436  0   May 25 ?        0:00 /usr/lib/ssh/sshd
  root 11090   436  0   May 23 ?        0:00 /usr/lib/ssh/sshd
```

UNIX Process Creation

- A process creates another process by calling the **fork**() function
  - The calling process is called the ***parent*** and the created process is called the ***child***
  - The fork function copies the parent's memory image so that the new process receives a copy of the address space of the parent
  - Both processes continue at the instruction directly after the statement containing the `fork()` call (executing in their respective memory images)

```
#include <unistd.h>

pid_t fork(void);
```

- The return value from the **`fork`**`()` function is used to determine which process is the parent and which is the child; the child gets the value 0 while the parent gets the child's process ID
  - When the **`fork`**`()` call fails, it returns –1 and sets `errno` (a child is not created)

Fork Example #1 (ch03/twoprocs.c)

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

int main(void) {
  pid_t childpid;

  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork");
    return 1;
  }
  if (childpid == 0)                    /* child code */
    printf("I am child %ld\n", (long)getpid());
  else                                  /* parent code */
    printf("I am parent %ld\n", (long)getpid());
  return 0;
}
```

How to Compile:

gcc twoprocs.c –o twoprocs.out

How to Run:

./twoprocs.out

Fork Example #2 (ch03/badprocessID.c)

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

int main(void) {
  pid_t childpid;
  pid_t mypid;

  mypid = getpid();
  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork");
    return 1;
  }
  if (childpid == 0)                          /* child code */
    printf("I am child %ld, ID = %ld\n", (long)getpid(), (long)mypid);
  else                                        /* parent code */
    printf("I am parent %ld, ID = %ld\n", (long)getpid(), (long)mypid);
  return 0;
}
```

How to Compile:

gcc badprocessID.c –o badprocessID.out

How to Run:

./badprocessID.out

Fork Example #3 (ch03/simplechain.c)

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

int main (int argc, char *argv[]) {
  pid_t childpid = 0;
  int i, n;

  if (argc != 2){  /* check for valid number of command-line arguments */
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    return 1;
  }
  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if (childpid = fork())
      break;

fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n", i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```

How to Compile:

gcc simplechain.c –o simplechain.out

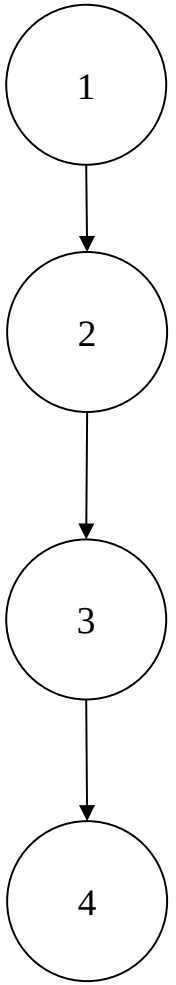
How to Run:

./simplechain.out 4

Fork Example #3 (ch03/simplechain.c)

Sample Run

```
% ./simplechain.out 4
i:1  process ID: 2736  parent ID:  40   child ID: 3488
i:2  process ID: 3488  parent ID: 2736  child ID:  512
i:4  process ID:  120  parent ID: 512   child ID:    0
i:3  process ID:  512  parent ID: 3488  child ID:  120
```



Fork Example #4 (ch03/simplefan.c)

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

int main (int argc, char *argv[]) {
  pid_t childpid = 0;
  int i, n;

  if (argc != 2){   /* check for valid number of command-line arguments */
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    return 1;
  }
  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) <= 0)
      break;

fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n", i, (long)getpid(),  (long)getppid(), (long)childpid);
  return 0;
}
```

How to Compile:

gcc simplefan.c –o simplefan.out

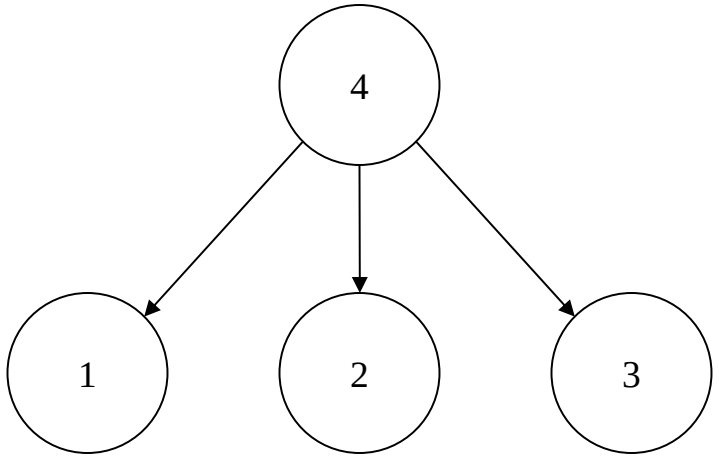
How to Run:

./simplefan.out 4

Fork Example #3 (ch03/simplefan.c)

Sample Run

```
% ./simplefan.out 4
i:1  process ID: 2736  parent ID:  120  child ID:    0
i:2  process ID: 3488  parent ID:  120  child ID:    0
i:4  process ID:  120  parent ID:   40  child ID:  512
i:3  process ID:  512  parent ID:  120  child ID:    0
```

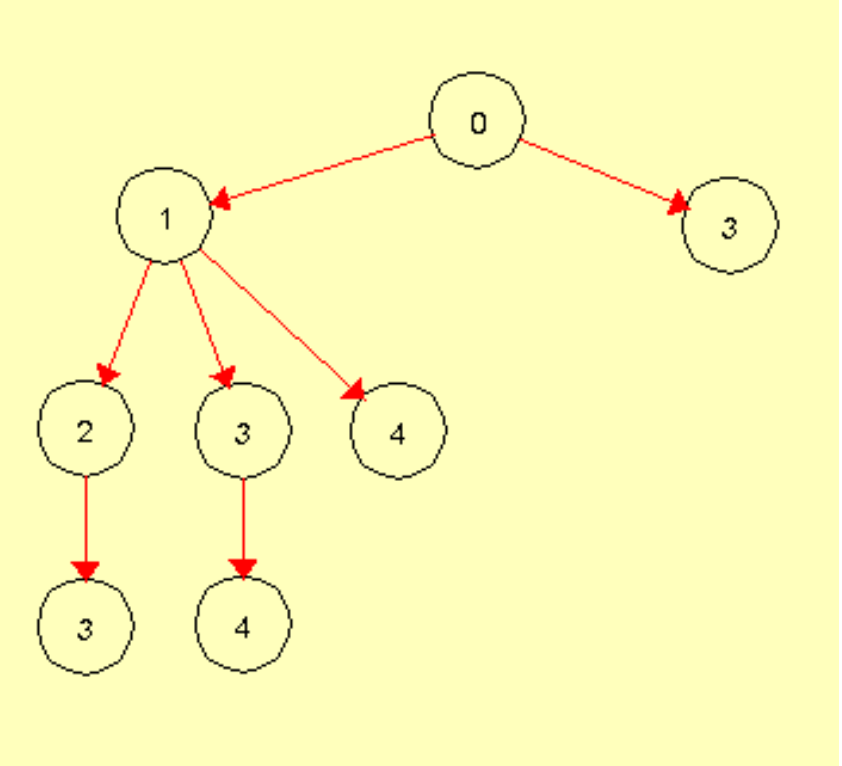


Exercise 1

- Trace the following program segment and determine how many processes are created.

```
c2 = 0;
c1 = fork(); /* fork number 1 */
if (c1 == 0)
c2 = fork(); /* fork number 2 */
fork(); /* fork number 3 */
if (c2 > 0)
fork(); /* fork number 4 */
```

Answer



The wait() Function

- When a process creates a child, both parent and child proceed with execution from the point of the fork
- The parent process can run the **wait()** or **waitpid()** functions to block its execution until the child process finishes
- The **wait()** function causes the caller (i.e., the parent) to suspend execution until a child's status becomes available

    - A process status most commonly becomes available after process termination

      #include <sys/wait.h>

      pid_t **wait**(int *status_location);

- It takes one parameter, a pointer to the location for returning the status of the process
- The function returns either the process ID that terminated or –1 (and sets `errno`)
- If a parent terminates without waiting for a child, the child becomes an ***orphan*** and is adopted by a special system process
- If a child process terminates and its parent does not wait for it, it becomes a ***zombie*** in UNIX terminology
    - Zombies stay in the system until they are waited for

- Traditionally, this process is called **init** and has process ID of 1; it periodically waits for children, so eventually orphaned zombies are removed

Wait Example #1 (ch03/parentwaitpid.c)

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

int main (void) {
   pid_t childpid;

  childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)
      fprintf(stderr, "I am child %ld\n", (long)getpid());
   else if (wait(NULL) != childpid)
      fprintf(stderr, "A signal must have interrupted the wait!\n");
   else
      fprintf(stderr, "I am parent %ld with child %ld\n", (long)getpid(),
          (long)childpid);
   return 0;
}
```

How to Compile:

gcc parentwaitpid.c –o parentwaitpid.out

How to Run:

./parentwaitpid.out

Wait Example #2 (ch03/fanwait.c)

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

int main(int argc, char *argv[]) {
  pid_t childpid;
  int i, n;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s n\n", argv[0]);
    return 1;
  }
  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) <= 0)
      break;

  while(wait(NULL) > 0) ;    /* wait for all of your children */
  fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```

How to Compile:

gcc fanwait.c –o fanwait.out

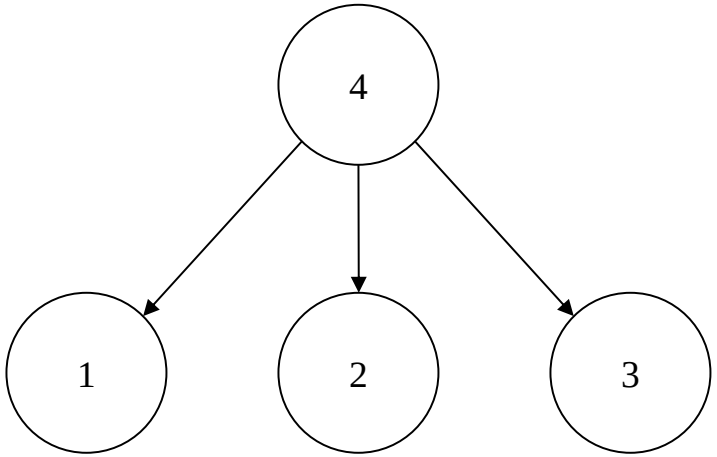
How to Run:

./fanwait.out 4

wait() Example (using fan code)

Sample Run

```
% ./fanwait.out  4
 i:1  process ID: 2736  parent ID:  120  child ID:    0
 i:2  process ID: 3488  parent ID:  120  child ID:    0
 i:3  process ID:  512  parent ID:  120  child ID:    0
 i:4  process ID:  120  parent ID:   40  child ID:  512
```



The waitpid() Function

- The **`waitpid()`** function allows a parent to wait for a *particular* child to terminate
  - It also allows a parent process to check whether a child has terminated without blocking

```
#include <sys/wait.h>

pid_t waitpid ( pid_t pid,  int *status_location,  int options );
```

- The function takes three paremeters: a pid, a pointer to the location for returning a status, and a flag specifying options
- There are several variations on the **pid** parameter and the resulting actions of the **`waitpid()`** function
  - `pid > 0` waits for the specific child whose process ID is pid
  - `pid == -1` waits for any child
- By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.
- The **options** parameter is the bitwise inclusive OR of one or more flags
  - WNOHANG option causes the function to return even if the status of a child is not immediately available
  - It returns 0 to report that there are possibly unwaited-for children but that their status is not available

waitpid() Example

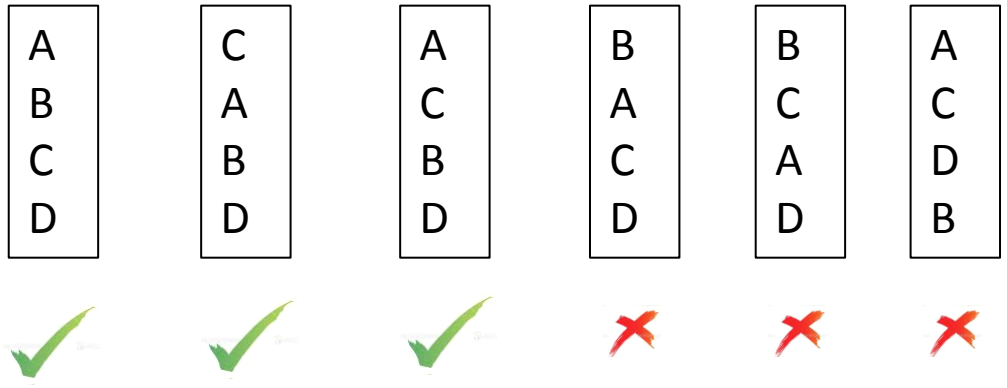
- The following code segment waits for all children that have finished but avoids blocking if there are no children whose status is available

```
pid_t childpid;

// Wait for any child
childpid = waitpid(-1, NULL, WNOHANG);
while (childpid != 0)
   {
   if ((childpid == -1) && (errno != EINTR))
      break;
   else
      childpid = waitpid(-1, NULL, WNOHANG);
   } // End while
```

Exercise 2

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

int main(void) {
   pid_t childpid;

   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {
        printf("A\n");
        printf("B\n");
   }
   else {
        printf("C\n");
        wait(NULL);
        printf("D\n");
   }
   return 0;
}
```

Output:



The exec Family of Functions

- The **`fork`**`()` function creates a copy of the calling process, but many applications require the child process to execute code that is different than that of the parent
- The **exec** family of functions provides a facility of overlaying the process image of the calling process with a new image
  - Usually the parent continues running the same code after the **`fork`**`()` call, while the child process runs the new program (by means of an exec function call)
- There are six variations of the **exec** function
  - Each differ in the way command-line arguments and the environment are passed
  - They also differ in whether a full pathname must be given for the executable
- All exec functions returns –1 if unsuccessful
  - If any of the exec functions return at all, the call was unsuccessful
- The **`execl`**`,` **`execlp`**`,` and **`execle`** functions pass the command-line arguments in an explicit list and are useful if the programmer knows the number of command line arguments at compile time
- The **`execv`**`,` **`execvp`**`,` and **`execve`** functions pass the command-line arguments in an argument array

Exec Functions

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg0, ... /*, char *(0) */);
int execle (const char *path, const char *arg0, ... /*, char *(0), char *const envp[] */);
int execlp (const char *file, const char *arg0, ... /*, char *(0) */);

int execv(const char *path, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
int execvp (const char *file, char *const argv[]);
```

execl() Example (ch03/execls.c)

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

int main (void)
{
pid_t childpid;

childpid = fork();
if (childpid == -1)
   {
   perror("Fork failed");
   return 1;
   } // End if

if (childpid == 0)
   {
   execl("/bin/ls", "ls", "-l", NULL);
   perror("Child failed when running execl");
   return 1;
   } // End if

if (childpid != wait(NULL))
   {
   perror("Parent failed to wait due to signal or error");
   return 1;
   } // End if

return 0;
}  // End main
```

This program creates a child process that runs the ls utility using the "-l" option. It displays a long listing of the contents of the current working directory

• "/bin/ls"	const char *path
• "ls"	const char *arg0
• "-1"	const char *arg1
• NULL	NULL pointer

How to Compile:

gcc execls.c –o execls.out

How to Run:

./execls.out

execvp() Example (ch03/execcmd.c)

```
int main (int argc, char *argv[])
{
pid_t childpid;

if (argc < 2)
   {
   fprintf(stderr, "Usage: a.out command arg1 arg2 ...\n");
   return 1;
   } // end if

childpid = fork();
if (childpid == -1)
   {
   perror("Fork failed");
   return 1;
   } // End if

if (childpid == 0)  /* Child code */
   {
   execvp(argv[1], &argv[1]);
   perror("Child failed upon running execvp function");
   return 1;
   }

if (childpid != wait(NULL)) /* Parent code */
   {
   perror("Parent failed to wait due to signal or error");
   return 1;
   }

return 0;
}  // End main
```

This program creates a child process that runs the command or program submitted on the command line.

• argv[1]	const char *file
• &argv[1]	const char *argv[]

How to Compile:

gcc execcmd.c –o execcmd.out

How to Run:

./execcmd.out ls

./execcmd.out ls -l

Background Processes

- A **shell** in UNIX terminology is a command interpreter that provides a prompt for a command, reads the command from standard input, forks a child to execute the command and waits for the child to finish
- When standard input and output come from a terminal type or device, a user can terminate an executing command by entering the interrupt character (commonly Ctrl-C)
- Most command shells interpret an input line ending with **&** (i.e., ampersand) as a command that should be executed as a background process
  - Example: **./a.out &**
  - When a shell creates a background process, it does not wait for the process to complete before issuing a prompt and accepting additional commands
  - Also, Ctrl-C from the keyboard does not terminate a background process
- A **daemon** is a background process that normally runs indefinitely
- UNIX relies on many daemon processes to perform routine tasks
  - For example, the Solaris `pageout` daemon handles paging for memory management
  - The `in.rlogind` daemon handles remote login requests
  - Other daemons handle mail, file transfer, statistics and printer requests