Διαχείριση διεργασιών

- Κάθε διεργασία στο Unix έχει έναν αριθμό ταυτότητας (PID), τον κώδικά της, τα δεδομένα της, μία στοίβα καθώς και κάποια άλλα χαρακτηριστικά
- Η αρχική διεργασία είναι η *init (PID=1)*
- Ο μόνος τρόπος να δημιουργηθεί μία διεργασία είναι κάποια άλλη να αναπαραγάγει τον εαυτό της, δηλαδή μία διεργασία-πατέρας να γεννήσει μία διεργασία-παιδί
- Όλες οι διεργασίες είναι απόγονοι της *init*
- Ο κώδικας, τα δεδομένα και η στοίβα της διεργασίας-παιδιού είναι αντίγραφα αυτών της διεργασίας-πατέρα
- Η ταυτότητα της διεργασίας-παιδιού είναι διαφορετική από την ταυτότητα της διεργασίας-πατέρα
- Μία διεργασία-παιδί μπορεί να αντικαταστήσει τον κώδικα, τα δεδομένα και τη στοίβα της με αυτά ενός εκτελέσιμου αρχείου, διαφοροποιώντας έτσι τον εαυτό της από τον πατέρα της

- Κλήση συστήματος **exit**
  - void **exit**(int *status*)
  - Τερματίζει την εκτέλεση μίας διεργασίας
  - Ο ακέραιος *status* ονομάζεται κωδικός εξόδου και είναι διαθέσιμος στη διεργασία-πατέρα
  - Κατά σύμβαση, η τιμή 0 για το *status* δείχνει επιτυχή τερματισμό
  - Τοπική μεταβλητή *status* του κελύφους C
- Χρήση της κλήσης **exit**

```
/* File: exit_code.c */
#include <stdio.h>   /* For printf */
main()
{  printf("I am going to terminate with exit code 23\n");
   exit(23); }
```

```
% exit_code
I am going to terminate with exit code 23
% echo $status
23
%
```

- Κλήση συστήματος **fork**
  - int **fork**( )
  - Αναπαράγει μία διεργασία, δημιουργώντας μία άλλη πανομοιότυπη
  - Στη διεργασία-πατέρα επιστρέφει την ταυτότητα της διεργασίας-παιδιού και στη διεργασία-παιδί επιστρέφει 0
  - Επιστρέφει λάθος (-1) στη διεργασία-πατέρα αν δεν είναι δυνατόν να δημιουργηθεί η νέα διεργασία-παιδί
- Κλήσεις συστήματος **getpid** και **getppid**
  - int **getpid**( )
  - int **getppid**( )
  - Επιστρέφουν τις ταυτότητες μίας διεργασίας και του πατέρα της αντίστοιχα

- Χρήση των κλήσεων **fork**, **getpid** και **getppid**

```
/* File: fork_a_process.c */
#include <stdio.h>   /* For printf */
main()
{  int pid;
   printf("Original process: PID = %d, PPID = %d\n",
          getpid(), getppid());
   pid = fork();
   if (pid == -1) {    /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {     /* The parent process */
      printf("Parent process: PID = %d, PPID = %d, CPID = %d\n",
             getpid(), getppid(), pid); }
   else {              /* The child process */
      printf("Child process: PID = %d, PPID = %d\n",
             getpid(), getppid()); }
   printf("Process with PID = %d terminates\n",
          getpid()); } /* Executed by both processes */
```

```
% fork_a_process
Original process: PID = 24079, PPID = 19518
Parent process: PID = 24079, PPID = 19518, CPID = 24080
Child process: PID = 24080, PPID = 24079
Process with PID = 24079 terminates
Process with PID = 24080 terminates
%
```

- Αν μία διεργασία τερματίσει πρίν από κάποιο παιδί της, τότε το τελευταίο, σαν ορφανή (orphan) διεργασία, υιοθετείται από την *init*

```
/* File: orphan_process.c */
#include <stdio.h>   /* For printf */
main()
{  int pid;
   printf("Original process: PID = %d, PPID = %d\n",
          getpid(), getppid());
   pid = fork();
   if (pid == -1) {    /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {     /* The parent process */
      printf("Parent process: PID = %d, PPID = %d, CPID = %d\n",
             getpid(), getppid(), pid); }
   else {              /* The child process */
      sleep(2);        /* Delay child in order to ensure
                          that the parent terminates first */
      printf("Child process: PID = %d, PPID = %d\n",
             getpid(), getppid()); }
   printf("Process with PID = %d terminates\n",
          getpid()); } /* Executed by both processes */
```

```
% orphan_process
Original process: PID = 24321, PPID = 19518
Parent process: PID = 24321, PPID = 19518, CPID = 24322
Process with PID = 24321 terminates
% Child process: PID = 24322, PPID = 1
Process with PID = 24322 terminates
```

- Κλήση συστήματος **wait**
  - int **wait**(int **status*)
  - Προκαλεί την αναμονή μίας διεργασίας μέχρις ότου κάποιο παιδί της τερματίσει
  - Αποδέχεται τον κωδικό εξόδου του παιδιού, δηλαδή τον ακέραιο της εντολής **exit**, στο αριστερό byte του *status*, ενώ το δεξιό byte είναι 0
  - Αν το παιδί τερματίσει εξ αιτίας κάποιου σήματος, τότε τα 7 τελευταία bits του *status* αντιπροσωπεύουν το χαρακτηριστικό αριθμό αυτού του σήματος
  - Επιστρέφει την ταυτότητα του παιδιού που τερμάτισε ή λάθος (-1) αν η διεργασία δεν έχει παιδιά

- Χρήση της κλήσης **wait**

```
/* File: wait_usage.c */
#include <stdio.h>   /* For printf */
main()
{  int pid, status;
   printf("Original process: PID = %d\n", getpid());
   pid = fork();
   if (pid == -1) {                /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {                 /* The parent process */
      printf("Parent process: PID = %d\n", getpid());
      while (wait(&status) != pid); /* Wait for child to exit */
      printf("Child terminated: PID = %d, exit code = %d\n",
             pid, status >> 8); }
   else {                          /* The child process */
      printf("Child process: PID = %d, PPID = %d\n",
             getpid(), getppid());
      exit(72); }              /* Exit with a silly number */
   printf("Process with PID = %d terminates\n",
          getpid()); }         /* Executed by parent only */
```

```
% wait_usage
Original process: PID = 25569
Parent process: PID = 25569
Child process: PID = 25570, PPID = 25569
Child terminated: PID = 25570, exit code = 72
Process with PID = 25569 terminates
%
```

- Μία διεργασία που τερματίζει δεν εγκαταλείπει το σύστημα μέχρις ότου ο πατέρας της αποδεχθεί τον κωδικό εξόδου της και είναι όλο αυτό το χρονικό διάστημα μία ζωντανή-νεκρή (zombie) διεργασία

```
/* File: zombie_process.c */
main()
{  int pid;
   pid = fork();
   if (pid == -1) {    /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {     /* The parent process */
      while (1)        /* Never terminate */
         sleep(1000); }
   else {              /* The child process */
      exit(37); } }    /* Exit with a silly number */
```

```
% zombie_process &
[1] 24409
% ps
  PID TT STAT   TIME COMMAND
24410 p1 Z      0:00 <exiting>
19518 p1 S      0:00 -csh (csh)
24409 p1 S      0:00 zombie_process
24411 p1 R      0:00 ps
% kill 24409

[1]    Terminated           zombie_process
% ps
  PID TT STAT   TIME COMMAND
19518 p1 S      0:00 -csh (csh)
24412 p1 R      0:00 ps
%
```

- Κλήσεις συστήματος της ομάδας **exec**
  - int **execl**(char **path*, char **arg0*, char **arg1*, ... , char **argn*, NULL)
  - int **execv**(char **path*, char **argv[ ]*)
  - int **execlp**(char **path*, char **arg0*, char **arg1*, ... , char **argn*, NULL)
  - int **execvp**(char **path*, char **argv[ ]*)
  - Αντικαθιστούν μία διεργασία (κώδικα, δεδομένα, στοίβα) με αυτά του εκτελέσιμου αρχείου που δίνεται στο *path*
  - Οι **execl** και **execv** απαιτούν απόλυτα ή σχετικά ονόματα-μονοπάτια στο *path*
  - Οι **execlp** και **execvp** χρησιμοποιούν τη μεταβλητή περιβάλλοντος PATH για να βρουν το εκτελέσιμο αρχείο
  - Οι **execl** και **execlp** θέλουν στο *arg0* το όνομα του εκτελέσιμου αρχείου, στα *arg1*, ... , *argn* τα ορίσματά του και το τελευταίο όρισμά τους NULL
  - Οι **execv** και **execvp** θέλουν το όνομα του εκτελέσιμου αρχείου και τα ορίσματά του στα *argv[0]*, *argv[1]*, ... , *argv[n]* και NULL στο *argv[n+1]*
  - Όλες οι κλήσεις της ομάδας **exec** επιστρέφουν λάθος (-1) αν δεν βρεθεί το εκτελέσιμο αρχείο, ενώ σε επιτυχία ποτέ δεν επιστρέφουν

- Χρήση της κλήσης **execl**

```
/* File: execl_demo.c */
#include <stdio.h>   /* For printf */
main()
{  int ret;
   printf("I am process %d and I will execute an 'ls -l ..'\n",
          getpid());
   ret = execl("/bin/ls", "ls", "-l", "..", NULL);
   if (ret == -1) {  /* Always true because of failure
                        to execute /bin/ls -l .. */
      perror("execl"); } }
```

```
% execl_demo
I am process 25637 and I will execute an 'ls -l ..'
total 4
drwxr-x--x  2 takis         512 Sep 15 13:31 bin
drwxr-xr-x  2 takis         512 Nov 15 11:14 c_progs
drwxr-xr-x  2 takis         512 Nov 10 15:09 mails_sent
drwxr-xr-x  2 takis         512 Nov 10 15:08 sh_scripts
%
```

- Χρήση της κλήσης **execvp**

```
/* File: execvp_demo.c */
#include <stdio.h>   /* For printf */
main()
{  int pid, status; char *argv[2];
   if ((pid = fork()) == -1) {      /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {                  /* Parent process */
      printf("I am parent process %d\n", getpid());
      while (wait(&status) != pid); /* Wait for child */
      printf("Child terminated with exit code %d\n",
             status >> 8); }
   else {                           /* Child process */
      argv[0] = "date";
      argv[1] = NULL;
      printf("I am child process %d", getpid());
      printf(" and I will replace myself by 'date'\n");
      execvp("date", argv);         /* Execute date */
      perror("execvp");
      exit(1); } }
```

```
% execvp_demo
I am parent process 25825
I am child process 25826 and I will replace myself by 'date'
Mon Nov 15 11:37:39 WET 1993
Child terminated with exit code 0
%
```

- *Να γραφεί ένα πρόγραμμα C που να δημιουργεί ένα δυαδικό δέντρο από διεργασίες το οποίο να έχει δεδομένο βάθος N. Κάθε διεργασία που δεν είναι φύλλο του δέντρου να εκτυπώνει την ταυτότητά της, την ταυτότητα του πατέρα της καθώς και έναν αύξοντα αριθμό σύμφωνα με μία πρώτα κατά πλάτος διάσχιση του δέντρου.*

```
/* File: process_tree.c */
#include <stdio.h>   /* For printf */
main(int argc, char *argv[])
{  int i, depth, numb, pid1, pid2, status;
   if (argc > 1) depth = atoi(argv[1]);    /* Make integer */
   else          exit(0);
   if (depth > 5) {                    /* Avoid deep trees */
      printf("Depth should be up to 5\n"); exit(0); }
   numb = 1;           /* Holds the number of each process */
   for(i=1 ; i<=depth ; i++) {
      printf("I am process no %2d with PID %5d and PPID %5d\n",
             numb, getpid(), getppid());
      switch (pid1 = fork()) {
         case 0:           /* Left child code */
            numb = 2*numb; break;
         case -1:          /* Error creating left child */
            perror("fork"); exit(1);
         default:          /* Parent code */
            switch (pid2 = fork()) {
               case 0:     /* Right child code */
                  numb = 2*numb+1; break;
               case -1:    /* Error creating right child */
                  perror("fork"); exit(1);
               default:    /* Parent code */
                  wait(&status); wait(&status);
                  exit(0); } } } }
```

```
% process_tree 1
I am process no  1 with PID  4025 and PPID 25507
% process_tree 2
I am process no  1 with PID  4028 and PPID 25507
I am process no  2 with PID  4029 and PPID  4028
I am process no  3 with PID  4030 and PPID  4028
% process_tree 3
I am process no  1 with PID  4036 and PPID 25507
I am process no  2 with PID  4037 and PPID  4036
I am process no  5 with PID  4040 and PPID  4037
I am process no  3 with PID  4038 and PPID  4036
I am process no  6 with PID  4043 and PPID  4038
I am process no  7 with PID  4044 and PPID  4038
I am process no  4 with PID  4039 and PPID  4037
% process_tree 4
I am process no  1 with PID  4051 and PPID 25507
I am process no  2 with PID  4052 and PPID  4051
I am process no  3 with PID  4053 and PPID  4051
I am process no  4 with PID  4054 and PPID  4052
I am process no  5 with PID  4055 and PPID  4052
I am process no  8 with PID  4056 and PPID  4054
I am process no  9 with PID  4057 and PPID  4054
I am process no 10 with PID  4058 and PPID  4055
I am process no 11 with PID  4059 and PPID  4055
I am process no  6 with PID  4068 and PPID  4053
I am process no  7 with PID  4069 and PPID  4053
I am process no 12 with PID  4070 and PPID  4068
I am process no 13 with PID  4071 and PPID  4068
I am process no 14 with PID  4072 and PPID  4069
I am process no 15 with PID  4073 and PPID  4069
%
```