ΔΙΕΡΓΑΣΙΕΣ (PROCESSES)

Λειτουργικά Συστήματα - Εργαστήριο

Πάνος Παπαδόπουλος

Τμήμα Πληροφορικής με Εφαρμογές στη Βιοϊατρική
Πανεπιστήμιο Θεσσαλίας

ΘΕΩΡΙΑ

ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ (1/2)

- Τα **processes** είναι βασικές μονάδες για την δέσμευση/κατανομή πόρων του συστήματος.
- Το **process** εκτελεί ένα πρόγραμμα.
- Κάθε **process** έχει ένα μοναδικό id, το **process id**.
- Επίσης κάθε **process** ανήκει σε μια ομάδα από processes που περιγράφεται μοναδικά από το **process group id**.
- Κάθε **process** έχει το δικό του **address space** και διαχειρίζεται (συνήθως) 1 thread.

ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ (2/2)

- Μπορούμε να έχουμε πολλαπλά **processes** που εκτελούν το **ίδιο** πρόγραμμα, αλλά κάθε proccess έχει το **δικό του αντίγραφο** του προγράμματος μέσα **στο δικό του address space** και **εκτελείται ανεξάρτητα** από τα άλλα.

- Κάθε process έχει ένα **parent process** που είναι αυτό που το δημιούργησε.

- Το process που δημιουργήθηκε ονομάζεται **child process**.

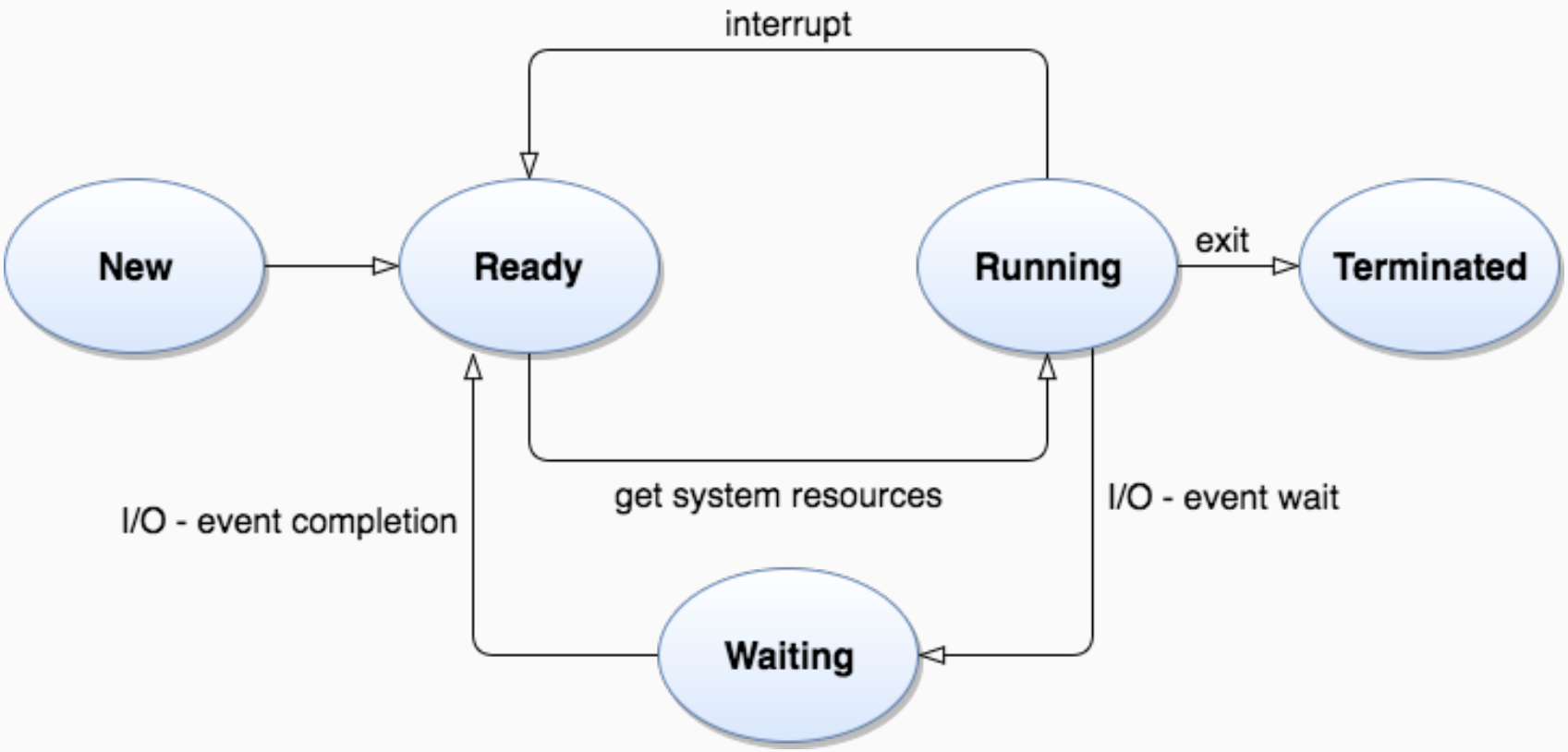
- Το **child process** κληρονομεί πολλά από τα χαρακτηριστικά του parent process.

PROCESS STATES (1/2)

Μετά τη δημιουργία του, το **process** μπορεί να βρίσκεται σε **μία** από τις παρακάτω καταστάσεις :

- **Ready :** το process είναι **έτοιμο** να λάβει πόρους του συστήματος όταν αυτοί γίνουν διαθέσιμοι.
- **Running :** το process **έχει** λάβει πόρους του συστήματος.
- **Waiting/Blocked :** το process **περιμένει** να λάβει πόρους του συστήματος.

PROCESS STATES (2/2)



ΓΙΑΤΙ ΠΟΛΛΑ PROCESSES?

- **Πολλά προγράμματα** που θέλουμε να τρέχουν την ίδια στιγμή.
- **Πολλοί χρήστες** που μοιράζονται το ίδιο μηχάνημα και θέλουν να τρέχουν τα δικά τους προγράμματα ανεξάρτητα από όλους τους άλλους.
- **Παραλληλισμός** που αφορά την εκτέλεση διαφορετικών τμημάτων μιας επεξεργασίας ή ενός υπολογισμού από ξεχωριστές διεργασίες και αποφυγή μπλοκαρίσματος εν' αναμονή κάποιου γεγονότος.
- **Καλύτερη δόμηση προγράμματος** γιατί μια πολύπλοκη διαδικασία ίσως μπορεί να μοντελοποιηθεί πιο απλά αν δομηθεί ως ένα σύστημα από διεργασίες.

SYSTEM CALL FORK() (1/2)

- #include <sys/types.h> (Δήλωση του τύπου δεδομένων pid_t)
- Prototype : pid_t fork()
- Η fork **δε** δέχεται παραμέτρους.
- Είναι ο μόνος τρόπος δημιουργίας ενός νέου process σε περιβάλλον UNIX.
- Δημιουργεί ένα νέο process που είναι **αντίγραφο** του process που κάλεσε τη fork.

SYSTEM CALL FORK() (2/2)

- **Η fork() εκτελείται 1 φορά αλλά επιστρέφει 2 τιμές!!!**
- Μια επιστροφή γίνεται στο parent process και μια στο child process.
- Επιστρέφει **αρνητικό αριθμό** αν **δε** δημιουργήθηκε child process.
- Στο **child process** επιστρέφει **0**.
- Στο **parent process** επιστρέφει **θετικό αριθμό** που αντιστοιχεί στο **process id** του child process που δημιουργήθηκε (το process id είναι τύπου pid_t).

CHILD PROCESS

- Το **child process** αρχικοποιείται **"κληρονομώντας"** την κατάσταση του parent process.
- Η μνήμη του **child process** (αν και ξεχωριστή), έχει αρχικά, τα **ίδια περιεχόμενα (static memory, dynamic memory, stack...)** με τη μνήμη του parent process.
- Το **child process** "κληρονομεί" επίσης και τον **program counter** του parent process. Αυτό σημαίνει ότι το child process αρχίζει την εκτέλεσή του από την εντολή που ακολουθεί την κλήση της fork (και όχι απο την αρχή του προγράμματος).
- Το **child process** εκτελεί τον **ίδιο κώδικα** με το parent process (αν και όπως θα δούμε αργότερα, αυτό μπορεί να αλλάξει).

PROCESS IDENTIFICATION

- #include <unistd.h> (Δήλωση των **getpid()** και **getppid()**)
- #include <sys/types.h> (Δήλωση του τύπου δεδομένων pid_t)
- Prototypes : pid_t **getpid()** και pid_t **getppid()**
- Οι συναρτήσεις **getpid()** και **getppid()** **δε** δέχονται παραμέτρους.
- Η **getpid()** επιστρέφει το **process id** του **process** που την κάλεσε.
- Η **getppid()** επιστρέφει το **process id** του **parent process** του process που την κάλεσε.

CODE 1

```
1  int main(int argc, char *argv[])
2  {
3      pid_t pid;
4
5      pid = fork();
6
7      printf("pid = %d\n", pid);
8
9      return 0;
10 }
```

CODE 2

```
1  int main (int argc, char *argv[])
2  {
3      pid_t pid;
4
5      pid = fork();
6
7      printf("pid = %d, getpid=%d, getppid=%d\n", pid, getpid(), getppid());
8
9      return 0;
10 }
```

CODE 3

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

    pid = fork();

    if (pid == 0) {
        printf("child process\n");
    }
    else {
        printf("parent process\n");
    }

    return 0;
}
```

PROCESS MEMORY

- Το parent process μπορεί να αναθέσει σε κάποιες μεταβλητές τις τιμές και τα δεδομένα που επιθυμεί να ”περάσει” στο child process, **προτού** το δημιουργήσει.

- Επειδή το child process έχει αντίγραφο της μνήμης του parent process όλες οι μεταβλητές του μετά τη κλήση της fork(), θα έχουν τις ίδιες τιμές με αυτές του parent process.

- Στη συνέχεια όμως, οποιαδήποτε αλλαγή στις μεταβλητές γίνεται **τοπικά** για το κάθε process, είναι δηλαδή **ορατή μόνο στο ίδιο**.

CODE 4

```c
int main(int argc, char *argv[])
{
    pid_t pid;
    int x = 10;

    pid = fork();
    if (pid == 0) {
        sleep(2); // wait for parent process to modify the data
        printf("child process\n");
        x = x * x; // x *= x;
        printf("x = %d\n", x);
    }
    else {
        printf("parent process\n");
        x = x - 4; // x -= 4;
        sleep(5); // wait for child process to modify the data
    }
    return 0;
}
```

EXIT() FUNCTION

- #include <stdlib.h> (Δήλωση της συνάρτησης **exit()**)
- Prototype : void exit(int status)
- Η **exit()** τερματίζει το process που την κάλεσε.
- Μπορεί να κληθεί σε από οποιοδήποτε σημείο του κώδικα του προγράμματος.
- Το **status** πρέπει να είναι μία τιμή από 0 έως 255.
- Το **status** αποθηκεύεται έτσι ώστε να μπορεί να παραληφθεί από το parent process.

ORPHAN VS ZOMBIE PROCESS

- Αν ένα parent process έχει τερματίσει ενώ το child process του συνεχίζει την εκτέλεσή του, τότε το child process γίνεται **orphan process**.
- Το **orphan process** ”υιοθετείται” από ένα ειδικό system process που ονομάζεται **init**.
- Αν ένα child process έχει τερματίσει ενώ το parent process του συνεχίζει την εκτέλεσή του, τότε το child process γίνεται **zombie process**.
- **Zombie process** είναι το process που ενώ έχει τελειώσει την εκτέλεσή του, υπάρχει ακόμη εγγραφή για αυτό στον **process table**.

ZOMBIE PROCESS

- Σε αντίθεση με τα ”κανονικά” processes, το zombie process **δεν** μπορεί να τερματιστεί χρησιμοποιώντας την εντολή **kill**.
- Υπεύθυνο για τον τερματισμό του είναι **μόνο** το parent process που του δημιουργήσε.
- Το λειτουργικό συνεχίζει να κρατά πληροφορία για το child(zombie) process, έτσι ώστε να μπορεί να την παραλάβει το parent process αργότερα.
- Δεσμεύονται πόροι του συστήματος χώρις αυτό να είναι αναγκαίο.

CODE 5

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

    pid = fork();

    if (pid == 0) {
        sleep(5);
        printf("child process : %d, parent : %d\n", getpid(), getppid());
        exit(0);
    }

    printf("parent process : %d\n", getpid());

    return 0;
}
```

CODE 6

```
1  int main(int argc, char *argv[])
2  {
3      pid_t pid;
4
5      pid = fork();
6
7      if (pid == 0) {
8          printf("child process : %d, parent : %d\n", getpid(), getppid());
9          exit(0);
10     }
11
12     while(1) {
13         printf("parent process : %d\n", getpid());
14         sleep(2);
15     }
16
17     return 0;
18 }
```

WAITPID() FUNCTION (1/2)

- #include <sys/wait.h> (Δήλωση της **waitpid()**)
- #include <sys/types.h> (Δήλωση του τύπου δεδομένων pid_t)
- Prototype : pid_t **waitpid(pid_t pid, int *status, int options)**
- Η **waitpid()** περιμένει ένα process **να αλλάξει κατάσταση**:
  - **Παύση** του child process από ένα signal.
  - **Συνέχεια** του child process από ένα signal.
  - **Τερματισμός** του child process.
- Η **waitpid()** επιστρέφει το **process id** του process που τερματίστηκε.

WAITPID() FUNCTION (2/2)

- Το **pid** μπορεί να πάρει τις παρακάτω τιμές :
  - **< -1 :** η waitpid() περιμένει για οποιοδήποτε child process του οποίου το **process group id** είναι ίδιο με το |**pid**|.
  - **-1 :** η waitpid() περιμένει για **οποιοδήποτε** child process.
  - **0 :** η waitpid() περιμένει για οποιοδήποτε child process του οποίου το **process group id** είναι ίδιο με αυτό του process που την κάλεσε.
  - **> 0 :** η waitpid() περιμένει για οποιοδήποτε child process του οποίου το **process id** είναι ίδιο με το **pid**.
- Στο ***status** αποθηκεύεται πληροφορία σχετικά με το process που τερματίστηκε.

CODE 7

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

    pid = fork();

    if (pid == 0) {
        sleep(5);
        printf("child process : %d, parent : %d\n", getpid(), getppid());
        exit(0);
    }

    printf("parent process : %d\n", getpid());
    waitpid(pid, NULL, 0);
    printf("child process terminated\n");

    return 0;
}
```

CODE 8

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

    pid = fork();

    if (pid == 0) {
        sleep(5);
        printf("child process : %d, parent : %d\n", getpid(), getppid());
        exit(0);
    }

    printf("parent process : %d\n", getpid());
    waitpid(-1, NULL, 0);
    printf("child process terminated\n");

    return 0;
}
```

ΑΛΛΑΓΗ ΚΩΔΙΚΑ ΤΟΥ PROCESS

- Ένα process μπορεί να **αλλάξει** το πρόγραμμα που εκτελεί μέχρι εκείνη τη στιγμή και να αρχίσει να εκτελει ένα **διαφορετικό** πρόγραμμα.

- **Δεν αλλάζει το process, αλλάζει το πρόγραμμα.**

- Για μπορέσει το process να αλλάξει το πρόγραμμα που εκτελεί χρησιμοποιούμε τις **exec()** family συναρτήσεις.

ΕΚΤΕΛΕΣΗ ΠΡΟΓΡΑΜΜΑΤΟΣ

- #include <unistd.h> (Δήλωση των **execlp()** και **execvp()**)
- Prototype : int **execlp**(const char *file, const char *arg0, const char *arg1, ...)
- Prototype : int **execvp**(const char *file, const char *argv[])
- Η **execlp()** εκτελεί το πρόγραμμα **file** με ορίσματα const char *arg0, const char *arg1, ...
- Η **execvp()** εκτελεί το πρόγραμμα **file** με ορίσματα const char *argv[].
- Οι συναρτήσεις **execlp()** και **execvp()** επιστρέφουν **μόνο** αν υπάρξει κάποιο πρόβλημα.
- Αν πετύχει η εκτέλεση, **δεν επιστρέφουν ποτέ στον κώδικα!**

CODE 9

```
1  int main(int argc, char *argv[])
2  {
3      pid_t pid;
4
5      pid = fork();
6
7      if (pid == 0) {
8          execlp("pwd", "pwd", NULL);
9          printf("execlp error\n");
10         exit(1);
11     }
12
13     waitpid(-1, NULL, 0);
14
15     return 0;
16 }
```

CODE 10

```
1  // program : add
2  int main (int argc, char *argv[])
3  {
4      int x, y;
5
6      if (argc != 3) {
7          printf("Invalid arguments\n");
8          return 1;
9      }
10
11     x = atoi(argv[1]);
12     y = atoi(argv[2]);
13
14     printf("%d + %d = %d\n", x, y, x+y);
15
16     return 0;
17 }
```

CODE 11

```
1  int main(int argc, char *argv[])
2  {
3      pid_t pid;
4
5      pid = fork();
6
7      if (pid == 0) {
8          execlp("./add", "./add", argv[1], argv[2], NULL);
9          printf("execlp error\n");
10         exit(1);
11     }
12
13     waitpid(-1, NULL, 0);
14
15     return 0;
16 }
```

ΑΣΚΗΣΕΙΣ

ΑΣΚΗΣΕΙΣ (1/2)

1. Να γραφεί πρόγραμμα σε C που θα δημιουργεί **ένα νέο process**. Το νέο process θα πρέπει να τυπώνει το δικό του process id και το process id του process που το δημιούργησε.

2. Να γραφεί πρόγραμμα σε C που θα δημιουργεί **δύο νέα processes**. Τα νέα processes θα πρέπει να τυπώνουν το καθένα το δικό του process id και το process id του process που τα δημιούργησε (το parent process id που θα τυπώνεται από τα 2 νέα processes θα πρέπει να είναι το ίδιο).

3. Να γραφεί πρόγραμμα σε C που θα τυπώνει τα περιεχόμενα του καταλόγου στον οποίο βρίσκεται.

4. Να γραφεί πρόγραμμα σε C που θα τυπώνει τα περιεχόμενα του καταλόγου στον οποίο βρίσκεται **με μορφή λίστας**.

ΑΣΚΗΣΕΙΣ (2/2)

5. Να γραφεί πρόγραμμα σε C που θα τυπώνει τα περιεχόμενα ενός καταλόγου. Το όνομα (ή το μονοπάτι) του καταλόγου θα δίνεται από τον χρήστη.

6. Να γραφεί πρόγραμμα σε C που θα μετονομάζει ένα αρχείο. Το όνομα του αρχείου θα δίνεται από τον χρήστη **ως όρισμα**. Θα πρέπει να γίνεται έλεγχος εισόδου και να εμφανίζεται κατάλληλο μήνυμα αν δε δοθεί κάποιο όρισμα.

ΑΣΚΗΣΕΙΣ (SHELL)

Να γραφεί πρόγραμμα σε C για τη δημιουργία ενός απλού shell. Το shell θα δέχεται από τον χρήστη μια συμβολοσειρά η οποία θα αποτελείται απο **μία εντολή (και τα ορίσματά της)**. Οι εντολές που θα υποστηρίζει θα πρέπει να αντιστοιχούν σε εφαρμογές που υπάρχουν ήδη στο Unix και βρίσκονται στο φάκελο **/bin**. Το shell τερματίζει όταν δοθεί το **exit**.

ΣΧΟΛΙΟ : ΜΗ χρησιμοποιήσετε την **system()**.

Bonus 1 : Εκτέλεση(εκτός των εφαρμογών του Unix) δικών σας εφαρμογών που βρίσκονται στον **ίδιο φάκελο** με το shell σας.

Bonus 2 : Εκτέλεση(εκτός των εφαρμογών του Unix) δικών σας εφαρμογών που βρίσκονται **οπουδήποτε**.