Δημιουργία & Τερματισμός Διεργασιών

Δημιουργία νέας διεργασίας

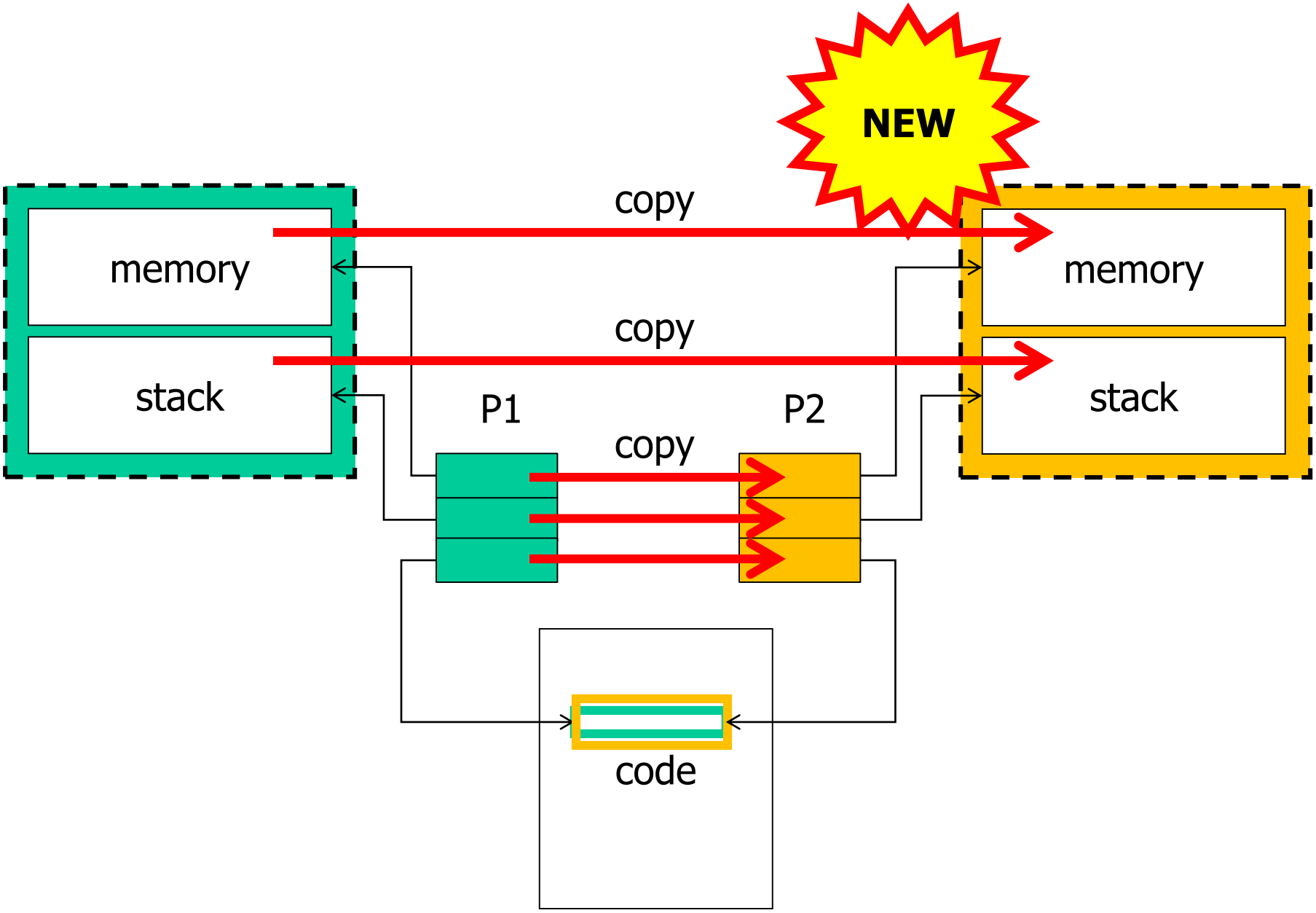
- `pid_t fork()`
- Η `fork` **δεν** έχει παραμέτρους
- Δημιουργεί μια νέα διεργασία που είναι ένα **αντίγραφο** της διεργασίας που καλεί την `fork`
- Η νέα διεργασία ονομάζεται **«παιδί»** ή «θυγατρική», και αυτή που την δημιούργησε **«γονιός»** ή «γονική»

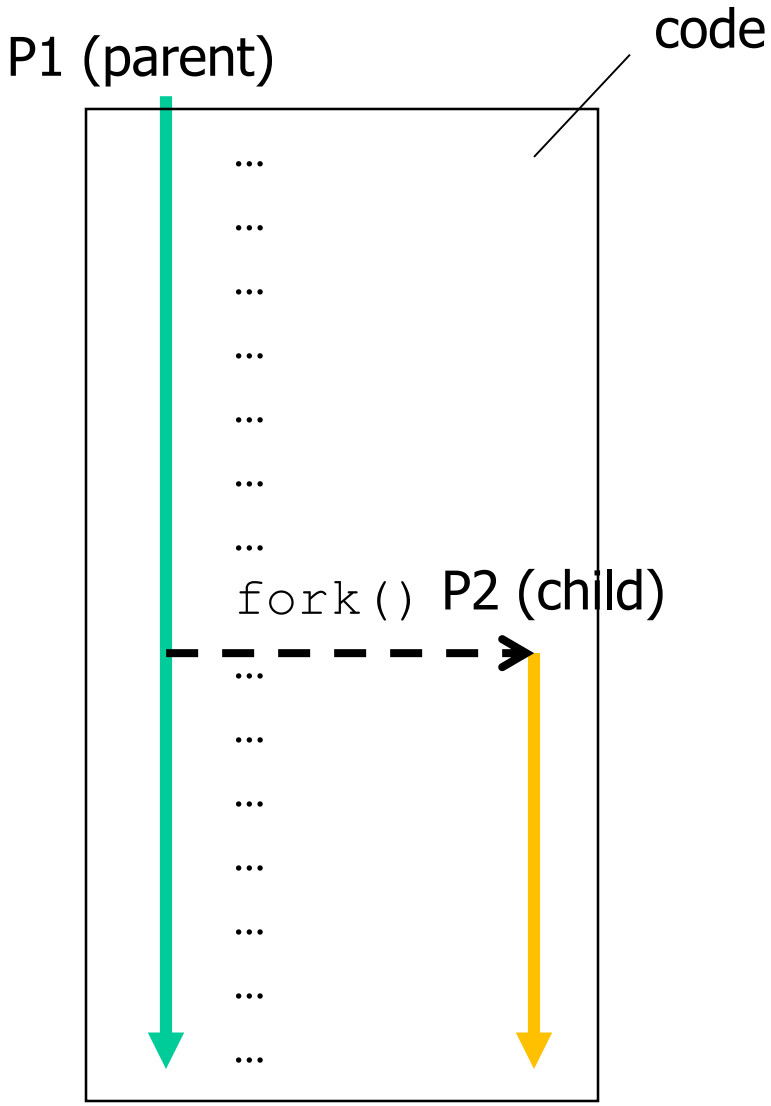
Κληρονομιά

- Το παιδί (ως αντίγραφο του γονιού) αρχικοποιείται **«κληρονομώντας»** την κατάσταση του γονιού

- Το παιδί εκτελεί τον **ίδιο κώδικα** με τον γονιό
- Η **μνήμη** του παιδιού (αν και ξεχωριστή) έχει, αρχικά, τα **ίδια περιεχόμενα** με αυτήν του γονιού
  - αντίγραφα στατικής μνήμης, δυναμικής μνήμης, στοίβας

- Ως μέρος της κατάστασης του γονιού, το παιδί «κληρονομεί» και τον program counter
  - συνεπώς, το παιδί αρχίζει την εκτέλεση του από την εντολή που ακολουθεί την κλήση της `fork` (και **όχι** από την πρώτη εντολή της `main`)


NEW
copy
memory
memory
copy
stack
stack
P1
P2
copy
code


P1 (parent)
code
...
...
...
...
...
...
...
fork()
P2 (child)
...
...
...
...
...
...
...

Διαχωρισμός γονιού-παιδιού

- Πως ξεχωρίζουμε, μέσα στον κώδικα, αν βρισκόμαστε στο πλαίσιο εκτέλεσης του γονιού ή του παιδιού;

- Ελέγχουμε την τιμή που επιστρέφει η `fork`
  - **= 0: για το παιδί**
  - **> 0: για τον γονιό** (αναγνωριστικό του παιδιού)
  - < 0: αποτυχία (δεν δημιουργήθηκε νέα διεργασία)

- Η τιμή επιστροφής χρησιμοποιείται συνήθως για να γίνει μια **διακλάδωση** μέσα στον κώδικα
  - έτσι ώστε ο γονιός και το παιδί να εκτελέσουν **διαφορετικό** τμήμα του κώδικα του προγράμματος

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

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

  pid = fork();

  printf("%d: fork returned %d\n",getpid(),pid);

  return(0);
}
```

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

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

  pid = fork();

  if (pid == 0) {
    printf("%d: hi from child\n",getpid());
  }
  else {
    printf("%d: hi from parent\n",getpid());
  }

  return(0);
}
```

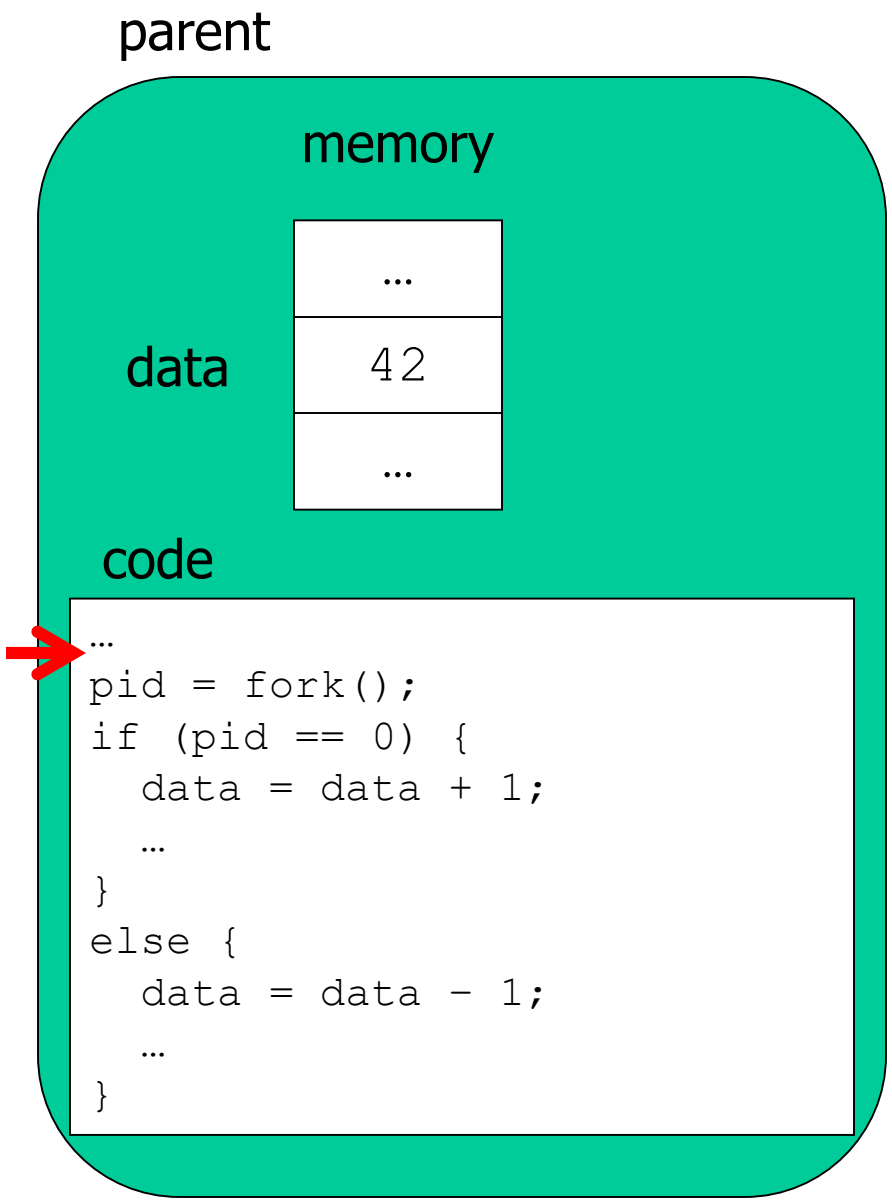
Μια πρώτη μορφή «επικοινωνίας»

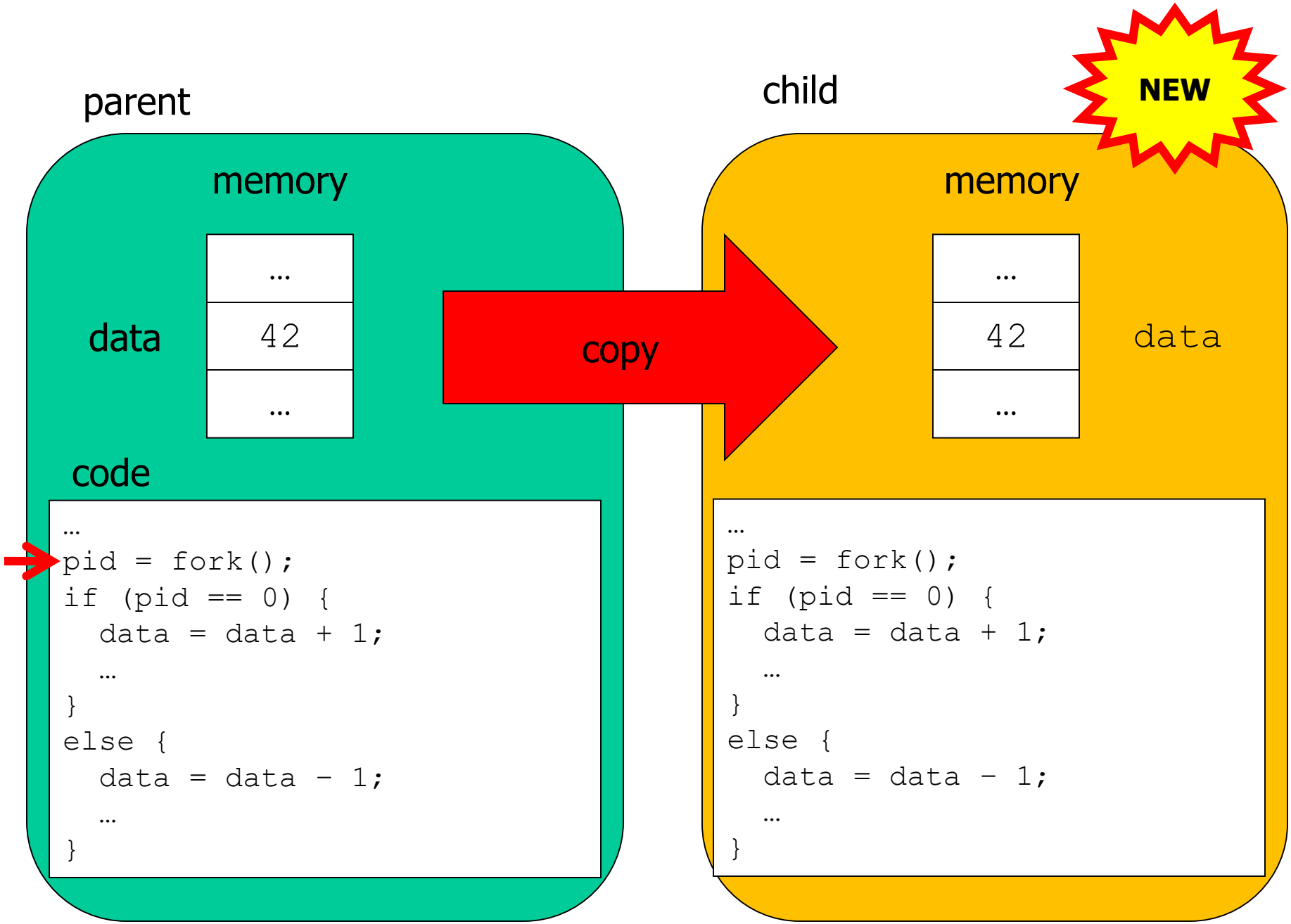
- Ο γονιός μπορεί να αναθέσει σε κάποιες μεταβλητές (θέσεις μνήμης) τιμές / δεδομένα που επιθυμεί να «περάσει» στο παιδί, **προτού** δημιουργήσει το παιδί

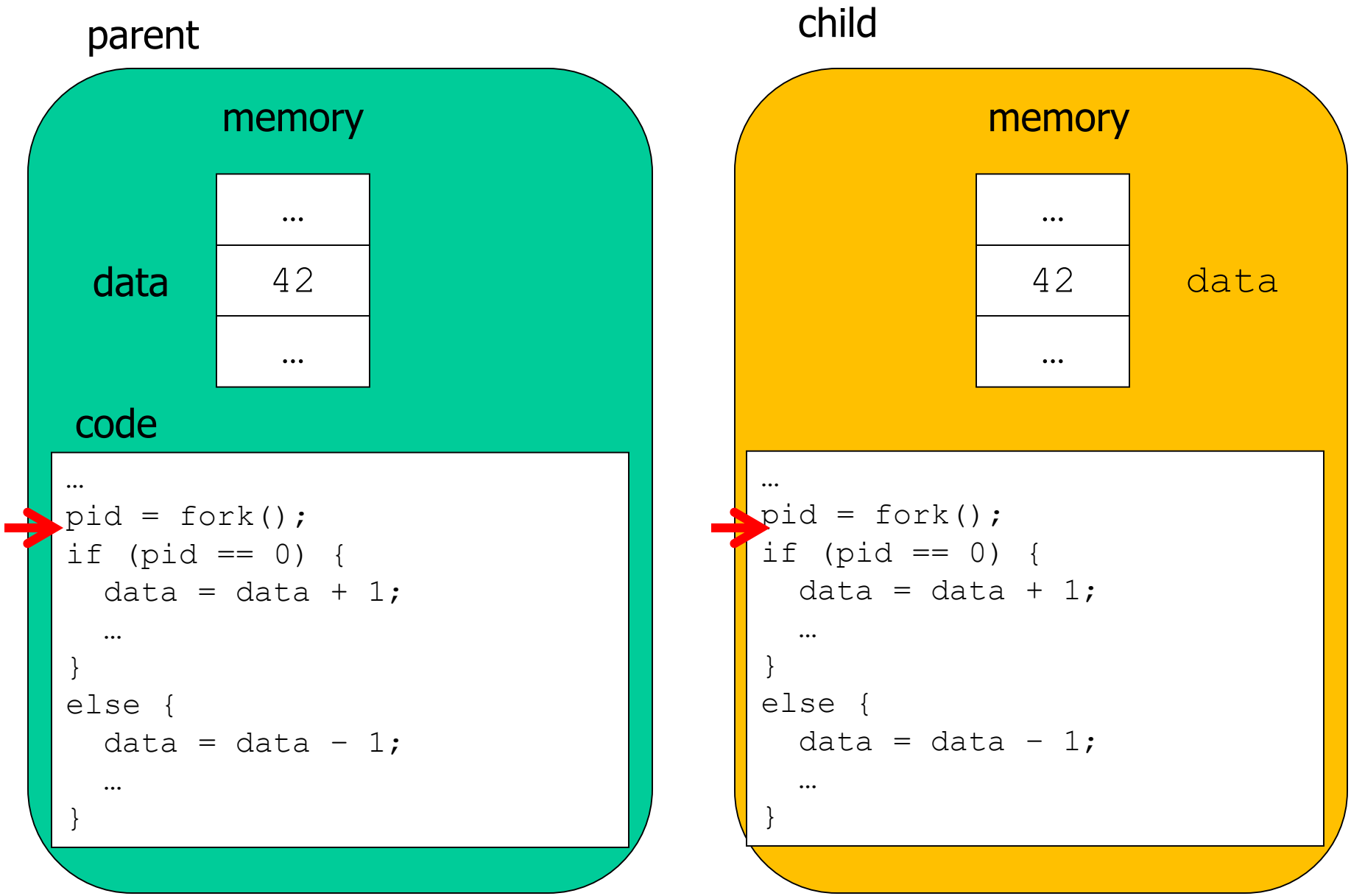
- Μετά την `fork`, το παιδί έχει ένα αντίγραφο της μνήμης του γονιού – άρα οι μεταβλητές του παιδιού έχουν (αρχικά) τις ίδιες τιμές με αυτές του γονιού

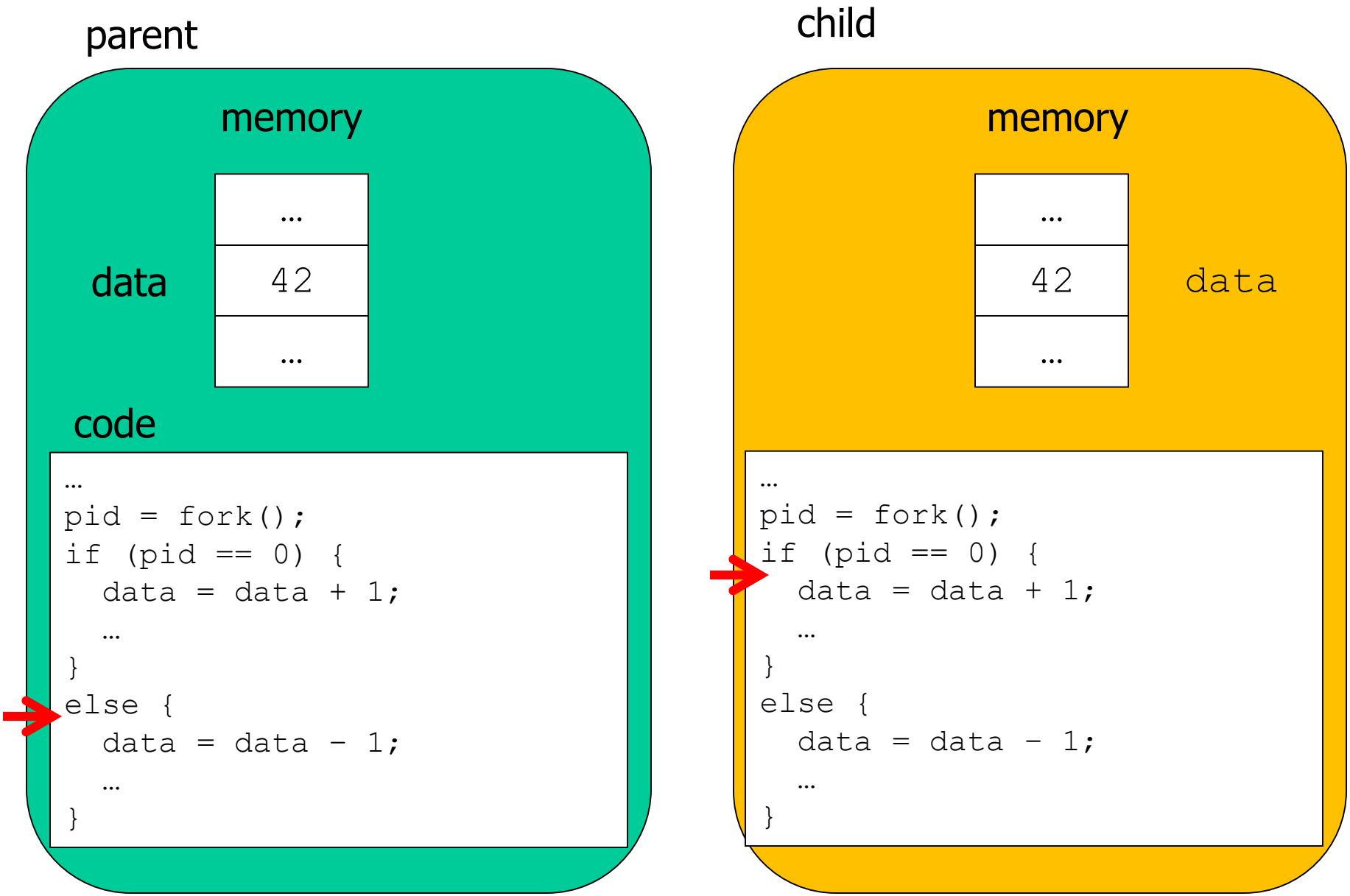
- Στην συνέχεια, αφού κάθε διεργασία έχει ξεχωριστή μνήμη, οι όποιες αλλαγές γίνονται στην μνήμη μιας διεργασίας είναι ορατές **μόνο** στην ίδια
  - δεν υπάρχει κάποια «κρυφή» μετάδοση δεδομένων ανάμεσα στην διεργασία γονιό και την διεργασία παιδί

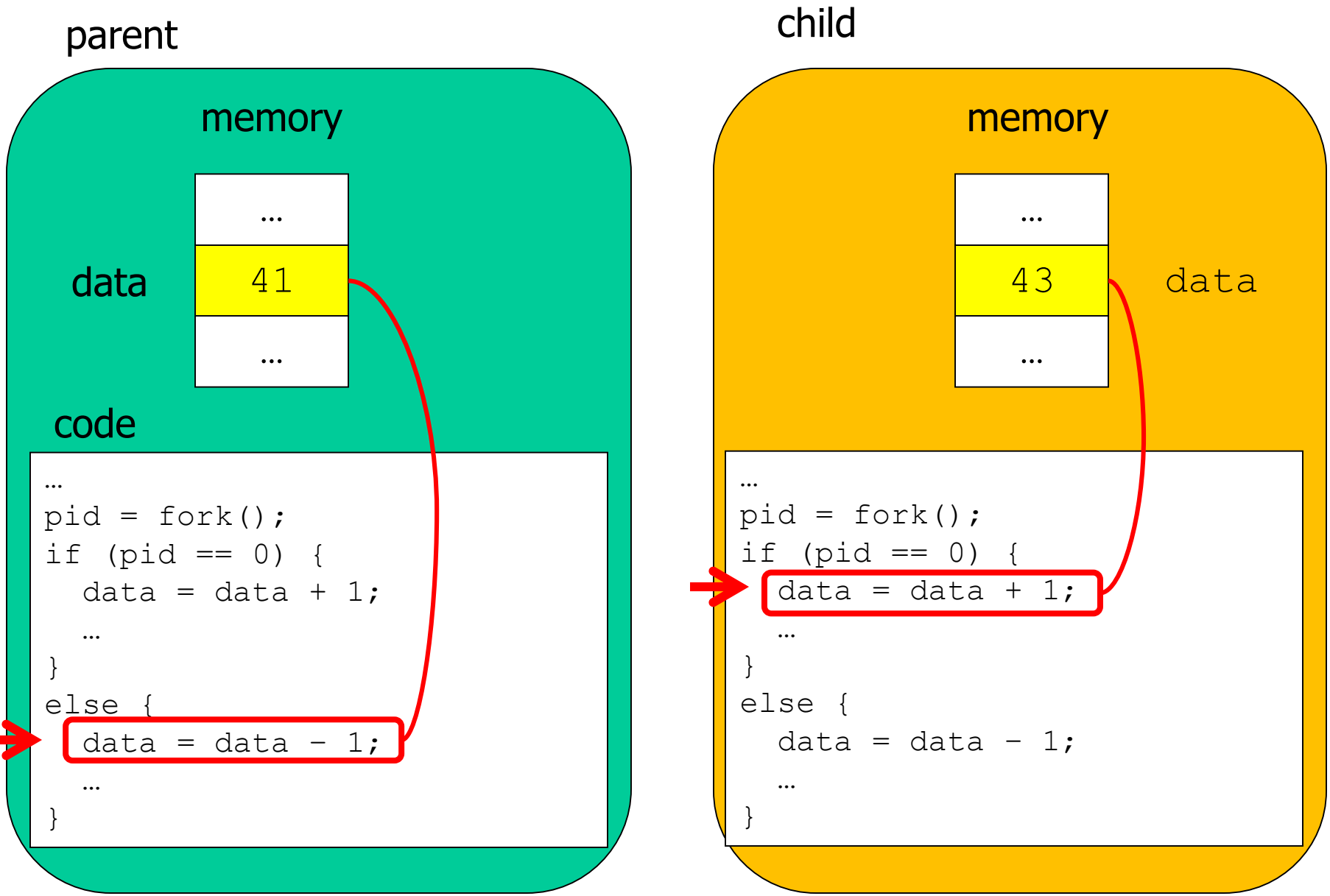
```
int main(int argc, char *argv[]) {
  int pid;
  int data = 42;

  pid = fork();

  if (pid == 0) {
    data = data + 1;
    printf("child: %d\n",data);
    return(0);
  }
  else {
    data = data - 1;
    printf("parent: %d\n",data);
    return(0);
  }
}
```


parent
memory
...
data
42
...
code
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data – 1;
…
}


parent
memory
…
data
42
…
code
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}
copy
child
NEW
memory
…
42
data
…
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}


parent
memory
…
data
42
…
code
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}
child
memory
…
42
data
…
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}


parent
memory
…
data
42
…
code
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data – 1;
…
}
child
memory
…
42
data
…
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data – 1;
…
}


parent
memory
...
data
41
...
code
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}
child
memory
...
43
data
...
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}

Κρυφή κατάσταση (βιβλιοθηκών)

- Μέρος της κατάστασης που κληρονομείται στο παιδί είναι και η **εσωτερική κατάσταση** των βιβλιοθηκών που χρησιμοποιεί ο γονιός
  - τιμές μεταβλητών, ενδιάμεσες αποθήκες δεδομένων, ...

- Αυτό μπορεί να προκαλέσει «περίεργα» φαινόμενα

- Για παράδειγμα, τα περιεχόμενα της εσωτερικής αποθήκης της `stdio` στον γονιό αντιγράφονται στην εσωτερική αποθήκη της `stdio` στο παιδί (όπως όλα τα περιεχόμενα της μνήμης του γονιού)

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

  printf("hello ");

  pid = fork();

  if (!pid) {
    printf("from child\n");
    return(0);
  }
  else {
    printf("from parent\n");
    return(0);
  }
}
```

λόγω line-buffered πολιτικής οι χαρακτήρες αποθηκεύονται **εσωτερικά** στην `stdio`

και τελικά γράφονται στην καθιερωμένη έξοδο **όταν** ολοκληρωθεί η γραμμή με `\n`

Τερματισμός διεργασίας

- `void _exit(int status)`
  - τερματίζει την διεργασία άμεσα, κλείνει περιγραφείς αρχείων
- `void exit(int status)`
  - κλήση **επιπλέον** διαδικασιών «καθαρισμού» του χρήστη
  - ισοδυναμεί με κλήση της `return` από την `main`
- Η `_exit/exit` μπορεί να κληθεί από οποιοδήποτε σημείο του κώδικα του προγράμματος
- Ο κωδικός κατάστασης `status` πρέπει να είναι μια τιμή (0..255) ώστε να χωράει σε ένα byte
  - συνήθως `EXIT_SUCCESS` (`0`) ή `EXIT_FAILURE` (`1`)
- Ο κωδικός κατάστασης **αποθηκεύεται** έτσι ώστε να μπορεί να παραληφθεί από την γονική διεργασία

Αναμονή για τερματισμό παιδιού

- `pid_t waitpid(pid_t pid,`
  `int *status,`
  `int options)`

- `pid > 0`: αναμονή για την συγκεκριμένη διεργασία
- `pid == 0`: αναμονή για οποιαδήποτε διεργασία βρίσκεται στην ίδια ομάδα με την γονική διεργασία
- `pid == -1`: αναμονή για οποιαδήποτε διεργασία
- `pid < -1`: αναμονή για οποιαδήποτε διεργασία βρίσκεται στην ομάδα με αναγνωριστικό `|pid|`
- Επιστρέφει το αναγνωριστικό της διεργασίας ή μιας από όλες τις διεργασίες που έχει/έχουν τερματιστεί

Ερμηνεία της τιμής επιστροφής

- Στην διεύθυνση `status` αποθηκεύεται πληροφορία σχετικά με την διεργασία που τερματίστηκε
- Η τιμή αυτή **ερμηνεύεται** μέσω μακροεντολών

- `WIFEXITED()`: `true` αν η θυγατρική διεργασία τερματίστηκε κανονικά (με `_exit` ή `exit`)
- `WEXITSTATUS()`: το λιγότερο σημαντικό byte της τιμής που επεστράφη μέσω `_exit` ή `exit`
- `WIFSIGNALED()`: `true` αν η θυγατρική διεργασία δεν τερματίστηκε κανονικά (αλλά μέσω σήματος)
- `WTERMSIG()`: ο αριθμός του σήματος που προκάλεσε τον τερματισμό της διεργασίας

```
int main(int argc, char *argv[]) {
  int pid; // child process id
  int status; // child status

  pid = fork();

  if (pid == 0) {
    sleep(5);
    exit(42); // return(42);
  }

  waitpid(pid,&status,0);
  if (WIFEXITED(status)) {
    printf("child returned %d\n",WEXITSTATUS(status));
  }
  else if (WIFSIGNALED(status)) {
    printf("child terminated with %d\n",WTERMSIG(status));
  }

  return(0);
}
```

Λίγα περισσότερα για την `waitpid`

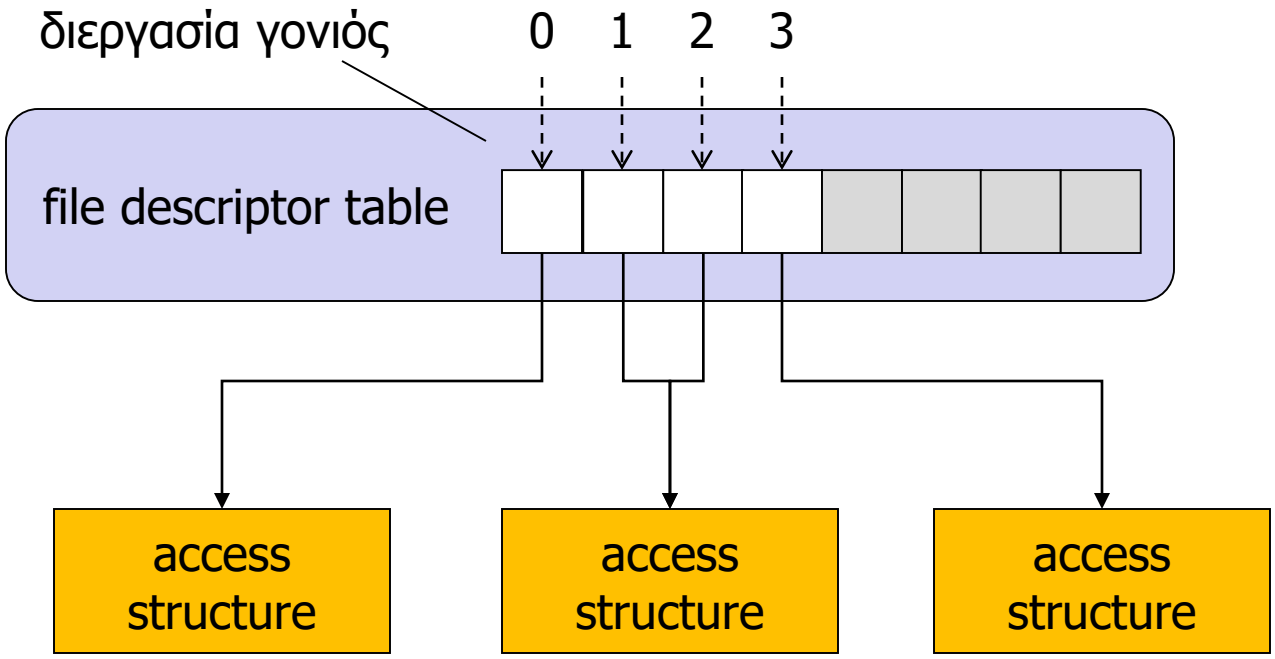
- Στην πραγματικότητα η `waitpid` επιστρέφει όταν γίνεται μια **αλλαγή κατάστασης** του παιδιού
- Υπάρχουν 3 περιπτώσεις: τερματισμός διεργασίας, σταμάτημα μέσω σήματος, ξεκίνημα μέσω σήματος
  - λίγο περισσότερα για τα σήματα αργότερα ...

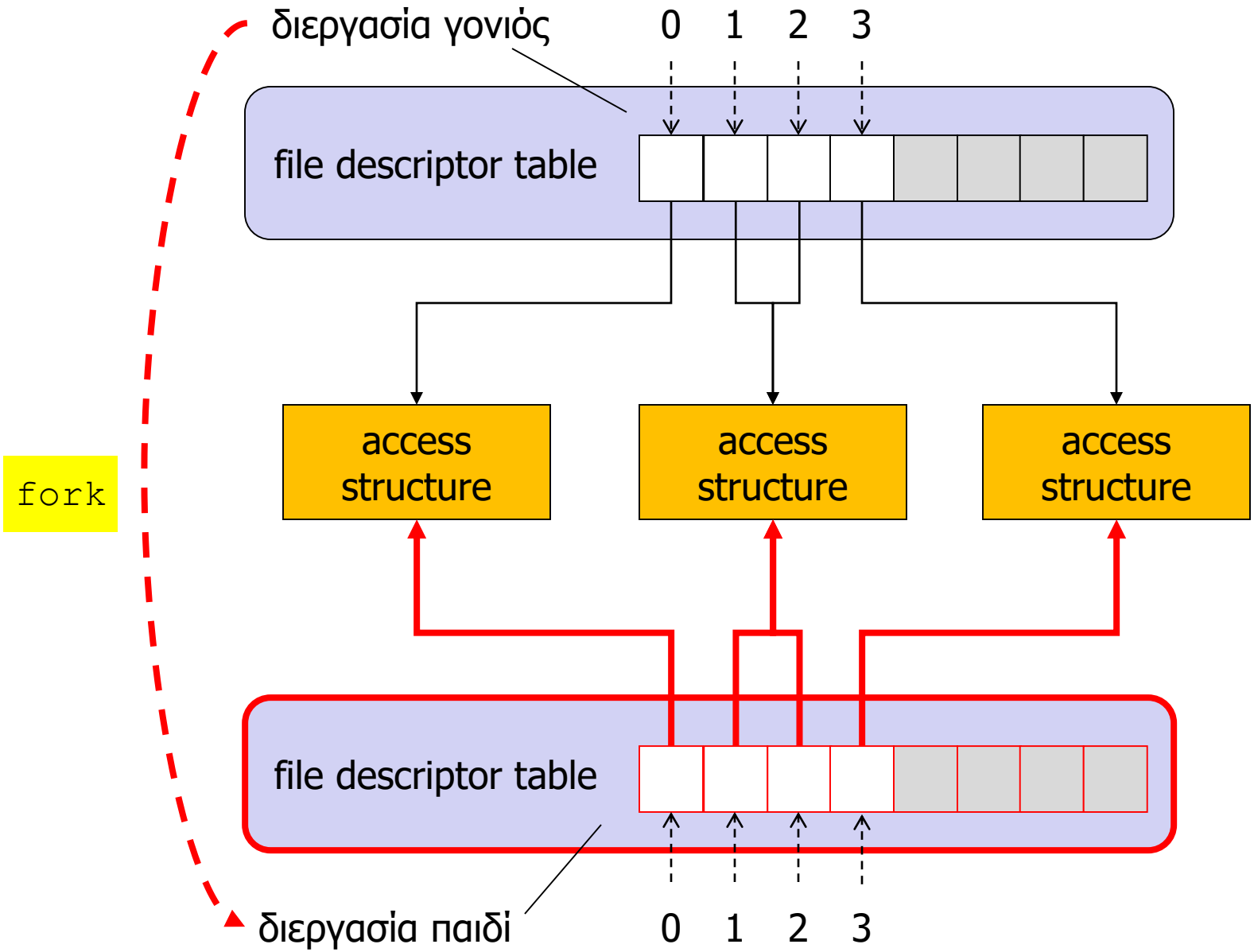
- Αν ο γονιός δεν περιμένει για τον τερματισμό ενός παιδιού, αυτό παραμένει σε μια κατάσταση **zombie**
- Το λειτουργικό **συνεχίζει** να κρατά πληροφορία για το παιδί (λόγος τερματισμού, τιμή επιστροφής κλ), για να μπορεί να την παραλάβει ο γονιός αργότερα
  - η πληροφορία κρατιέται μέχρι να τερματιστεί ο γονιός

Κληρονομιά περιγραφέων αρχείων

- Μαζί με την κατάσταση εκτέλεσης του γονιού, το παιδί κληρονομεί **αντίγραφα** των περιγραφέων αρχείων που έχει δημιουργήσει ο γονιός
  - στο πνεύμα της `dup`
- Η ανάγνωση/εγγραφή μέσω των περιγραφέων αρχείων της μιας διεργασίας **αλλάζουν** την θέση ανάγνωσης/εγγραφής των περιγραφέων της άλλης

- Ο γονιός πρέπει να κλείνει τους περιγραφείς που δεν θέλει να κληρονομήσει το παιδί, και αντίστοιχα το παιδί πρέπει να κλείνει τους κληρονομημένους περιγραφείς αρχείων που δεν χρειάζεται


διεργασία γονιός
0
1
2
3
file descriptor table
access structure
access structure
access structure


διεργασία γονιός
0 1 2 3
file descriptor table
access structure
access structure
access structure
fork
file descriptor table
διεργασία παιδί
0 1 2 3

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

  fd=open(argv[1],O_RDWR|O_CREAT|O_TRUNC,S_IRWXU);

  if (!(pid=fork())) {
    write(fd,"hello from child ",17);
    close(fd);
    return(0);
  }

  waitpid(pid,NULL,0); /* wait for child to write */
  write(fd,"and hello from parent",21);
  close(fd);

  return(0);
}
```

Αλλαγή κώδικα διεργασίας

- Μια διεργασία μπορεί να **αλλάξει** το πρόγραμμα που εκτελεί, και να αρχίσει την εκτέλεση ενός (εντελώς) **διαφορετικού** προγράμματος
  - η διεργασία μένει η ίδια

- Ένα παιδί μπορεί να «αυτονομηθεί» και να εκτελέσει **διαφορετικό** κώδικα από την γονική διεργασία
  - συνηθισμένος λόγος για την δημιουργία νέων διεργασιών

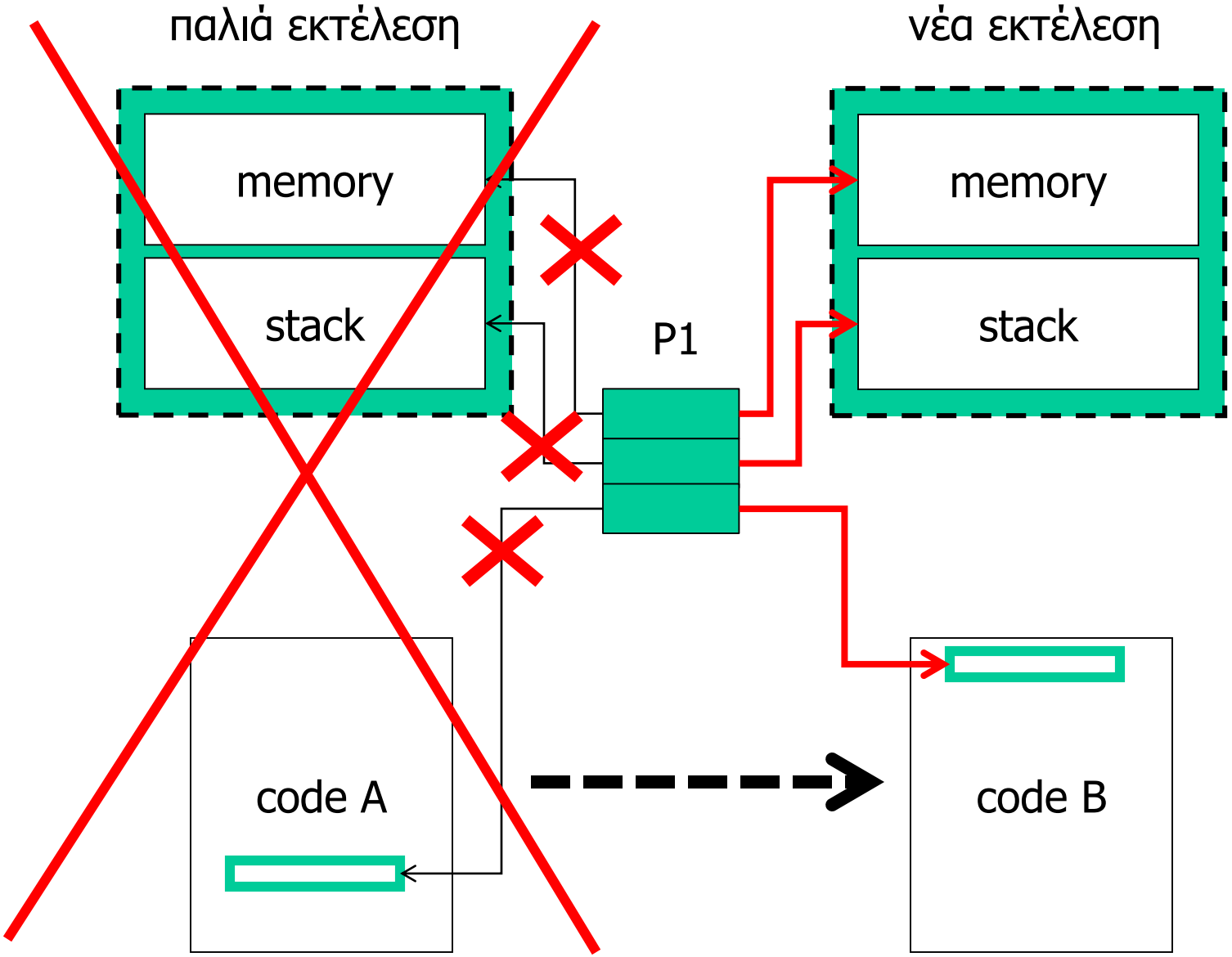
- Το σύστημα είναι **ανοιχτό**/**επεκτάσιμο**
- Μπορεί να γραφτεί κώδικας που κάποια στιγμή «θα» εκτελέσει κώδικα που **δεν** έχει γραφτεί ακόμα

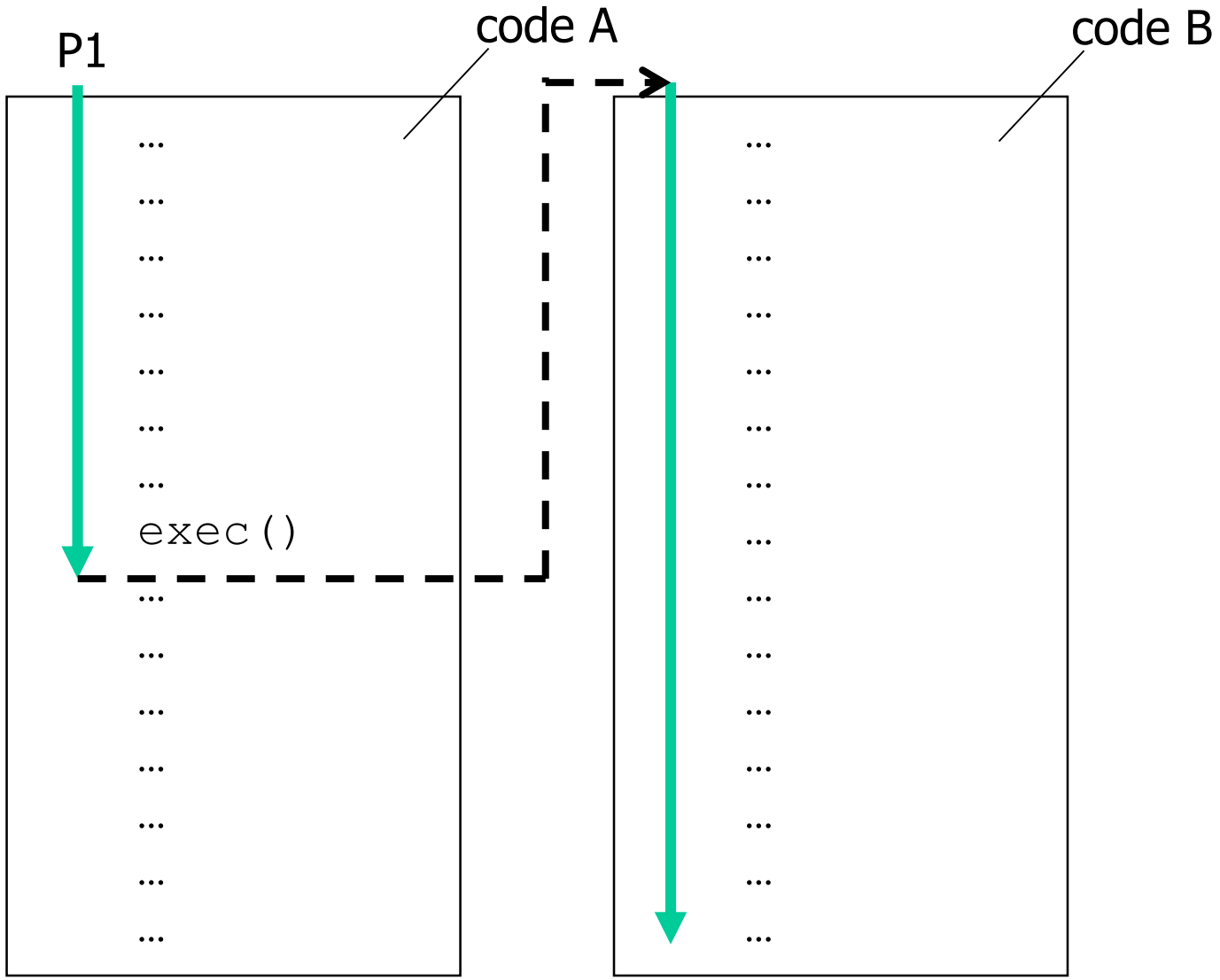
...

Εκτέλεση προγράμματος

- ```
  int execlp(const char *file,
             const char *arg0,
             const char *arg1,
             ...,
             NULL)
  ```
- Εκτελεί το πρόγραμμα `file` (με βάση το `PATH`) περνώντας τα ορίσματα `arg0`, `arg1`, ...
- Αν πετύχει, η κλήση **δεν επιστρέφει ποτέ**

- ```
  int execvp(const char *file,
             char *const argv[])
  ```
- Όπως η `execlp`, με τα ορίσματα του προγράμματος να περνιούνται ως διάνυσμα, όπως τα δέχεται η `main`


παλιά εκτέλεση
νέα εκτέλεση
memory
stack
P1
memory
stack
code A
code B


P1
code A
code B
exec()

πρόγραμμα add.c

```
int main(int argc, char *argv[]) {
  int v1,v2;

  v1=atoi(argv[1]);
  v2=atoi(argv[2]);

  printf("%d plus %d equals %d\n",v1,v2,v1+v2);
  return(0);
}
```

```
$ gcc add.c -o add
$ add 123 456
123 plus 456 equals 579
$
```

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

  if (!(pid=fork())) {
    printf("trying to execute add\n");
    execlp("./add","add",argv[1],argv[2],NULL);
    perror("execlp");
    return(1);
  }

  waitpid(pid,&status,0);
  if (WIFEXITED(status)) {
    printf("child returned %d\n",WEXITSTATUS(status));
  }
  else if (WIFSIGNALED(status)) {
    printf("child terminated with %d\n",WTERMSIG(status));
  }

  return(0);
}
```

Κατάσταση διεργασίας μετά το `exec`

- Όταν μια διεργασία αλλάζει κώδικα μέσω `exec`, η κατάσταση εκτέλεσης **αρχικοποιείται** εκ νέου

- Η διεργασία αποκτά **καινούργια** μνήμη
- Η διεργασία αποκτά **καινούργια** στοίβα

- Οι βιβλιοθήκες σε επίπεδο χρήστη ξαναφορτώνονται

- Είναι **σχεδόν** σα να δημιουργείται μια καινούργια διεργασία που εκτελεί το πρόγραμμα

Περιγραφείς αρχείων μετά το `exec`

- Οι περιγραφείς αρχείων που διατηρεί η διεργασία μέχρι τη στιγμή που αλλάζει κώδικα μέσω `exec` **παραμένουν** ανοιχτοί

- Το πρόγραμμα που εκτελεί η διεργασία μπορεί να χρησιμοποιήσει αυτούς τους περιγραφείς, σαν να τους είχε δημιουργήσει το ίδιο

- **Κλασική «εφαρμογή»:** ανακατεύθυνση της συμβατικής εισόδου/εξόδου ενός προγράμματος (**εν αγνοία** του ίδιου του προγράμματος), προτού αρχίσει η εκτέλεση του μέσω `exec`

```
int main(int argc, char *argv[]) {
  int fd,pid,status;

  if (!(pid=fork())) {
    fd=open(argv[3],O_RDWR|O_CREAT|O_TRUNC,S_IRWXU);
    dup2(fd,STDOUT_FILENO); // redirect stdout
    close(fd);
    execlp("./add","add",argv[1],argv[2],NULL);
    perror("execlp");
    return(1);
  }

  waitpid(pid,&status,0);
  if (WIFEXITED(status)) {
    printf("child returned %d\n",WEXITSTATUS(status));
  }
  else if (WIFSIGNALED(status)) {
    printf("child terminated with %d\n",WTERMSIG(status));
  }
  return(0);
}
```