Προγραμματισμός ΙΙ (ECE116)

#17 βασικές λειτουργίες διεργασιών (δημιουργία, τερματισμός, εκτέλεση νέου κώδικα)

Σπύρος Λάλης
lalis@uth.gr

Δημιουργία νέας διεργασίας

- `pid_t fork()`

- Η `fork` **δεν** έχει παραμέτρους
  - π.χ. δεν δέχεται κάποιο κώδικα προς εκτέλεση

- Δημιουργεί μια νέα διεργασία

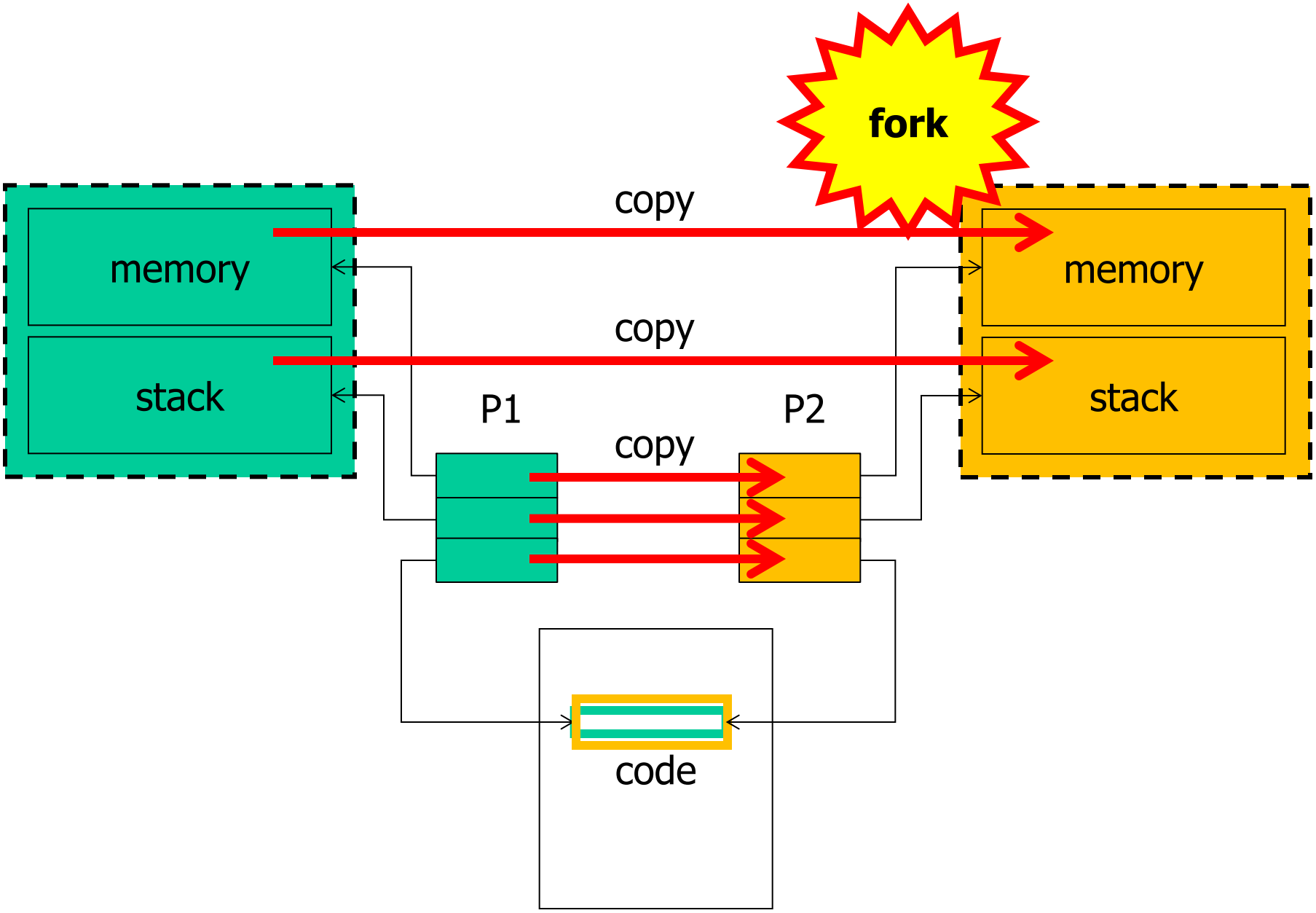
- Η νέα διεργασία ονομάζεται **«παιδί»**
  - ή «θυγατρική»
- Αυτή που την δημιούργησε ονομάζεται **«γονιός»**
  - ή «γονική»

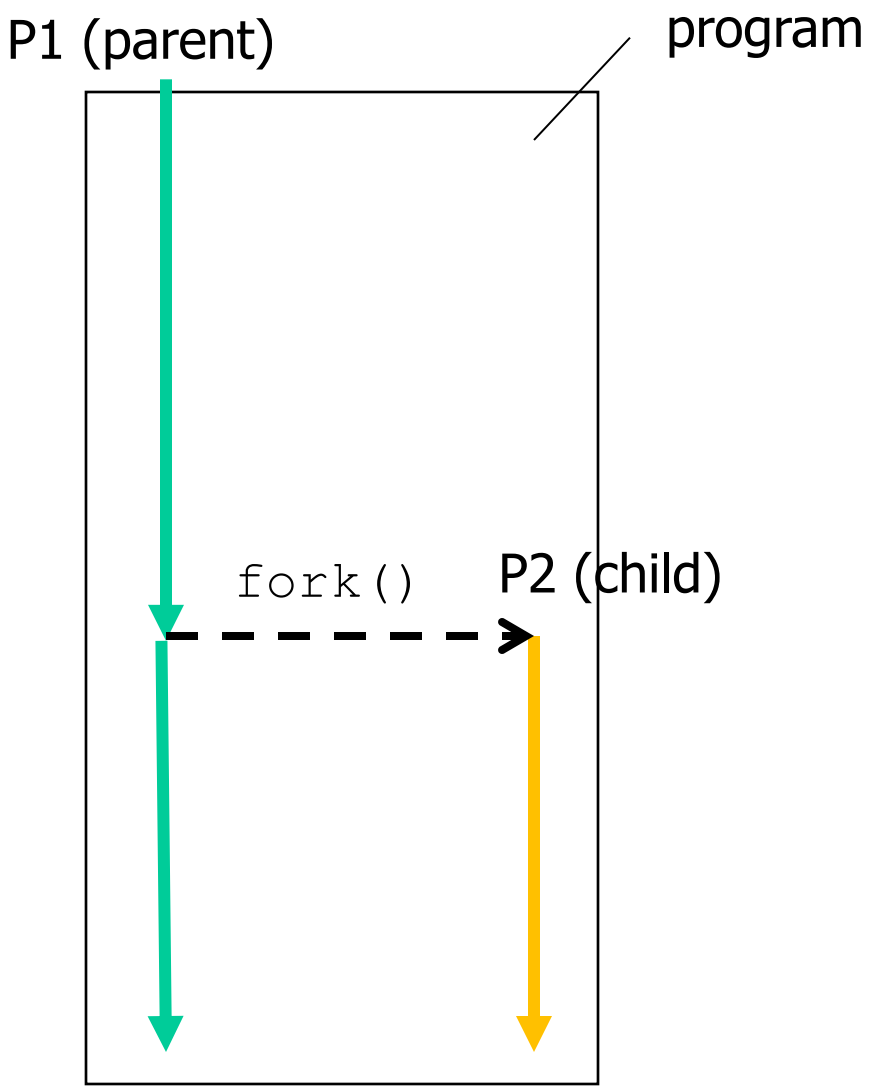
Κληρονομιά

- Το παιδί είναι ένα **αντίγραφο** του γονιού
  - αρχικοποιείται «κληρονομώντας» την κατάσταση του

- Το παιδί εκτελεί τον **ίδιο κώδικα** με τον γονιό
- Η **μνήμη** του παιδιού είναι **αντίγραφο** της μνήμης του γονιού, με τα ίδια περιεχόμενα
  - στατική μνήμη, δυναμική μνήμη, στοίβα

- Το παιδί «κληρονομεί» (μεταξύ άλλων) και τον **program counter** του γονιού
  - όπως ο γονιός του, το παιδί συνεχίζει την εκτέλεση του κώδικα **μετά** την κλήση της `fork` (όχι με την πρώτη εντολή της `main`)


fork
copy
memory
memory
copy
stack
stack
P1
P2
copy
code


P1 (parent)
program
fork()
P2 (child)

Διαχωρισμός γονιού-παιδιού

- Πως ξεχωρίζουμε, στο επίπεδο του κώδικα, αν μετά την κλήση της `fork` βρισκόμαστε στο πλαίσιο εκτέλεσης του γονιού ή του παιδιού;
  - συνηθισμένη περίπτωση: θέλουμε το παιδί να εκτελέσει διαφορετικό κώδικα από τον γονιό του
- Ελέγχουμε την τιμή που επιστρέφει η `fork`
  - `= 0`: πλαίσιο εκτέλεσης του **παιδιού**
  - `> 0`: πλαίσιο εκτέλεσης του **γονιού** για επιτυχία (επιστρέφεται το αναγνωριστικό του παιδιού)
  - `< 0`: πλαίσιο εκτέλεσης του **γονιού** για αποτυχία (δεν δημιουργήθηκε νέα διεργασία)
- Μπορεί να χρησιμοποιηθεί για να γίνει κατάλληλη **διακλάδωση** μέσα στον κώδικα του προγράμματος
  - εφόσον επιθυμούμε κάτι τέτοιο

```
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);
}
```

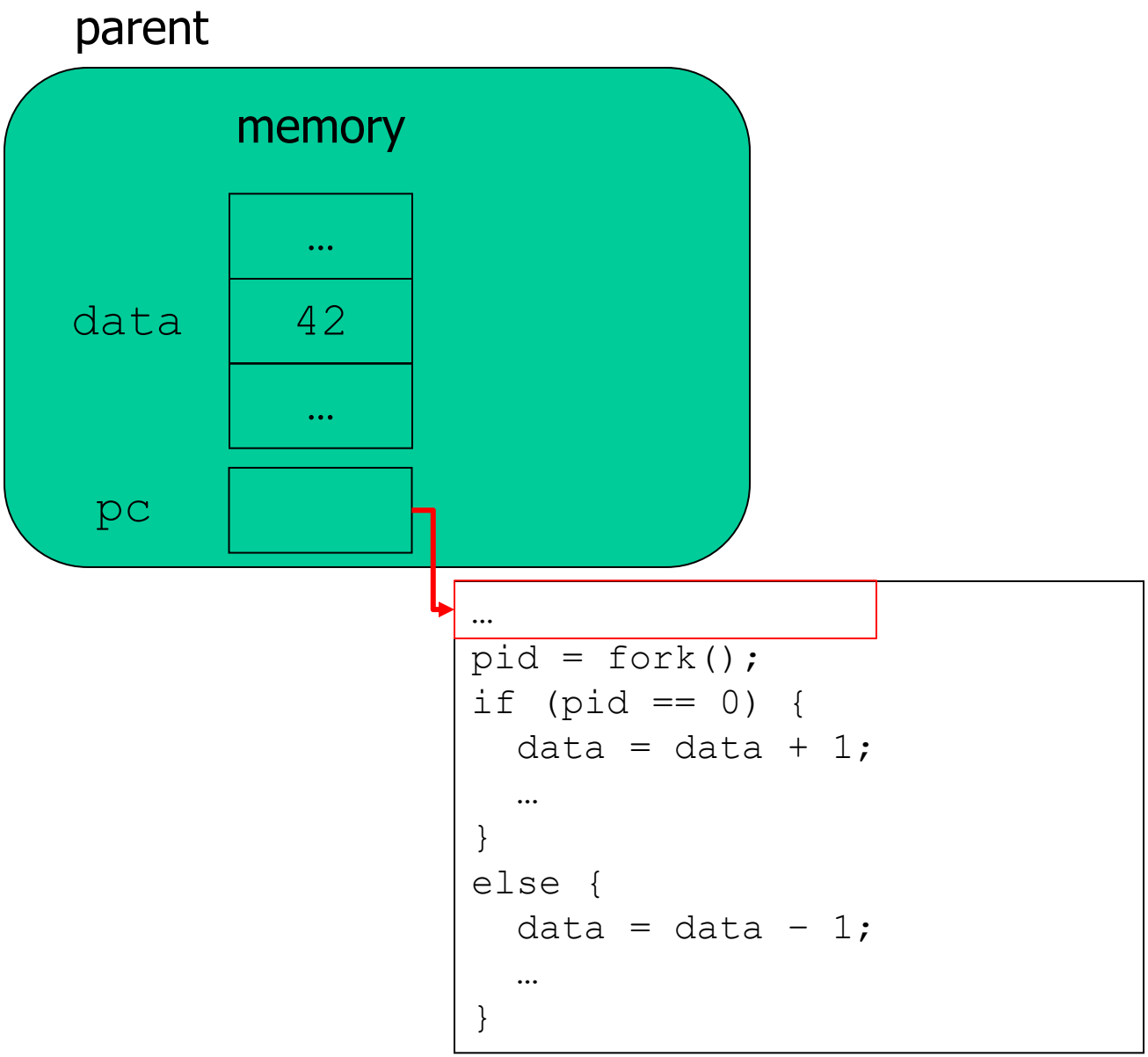
```
int main(int argc, char *argv[]) {
  int pid;

  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);
}
```

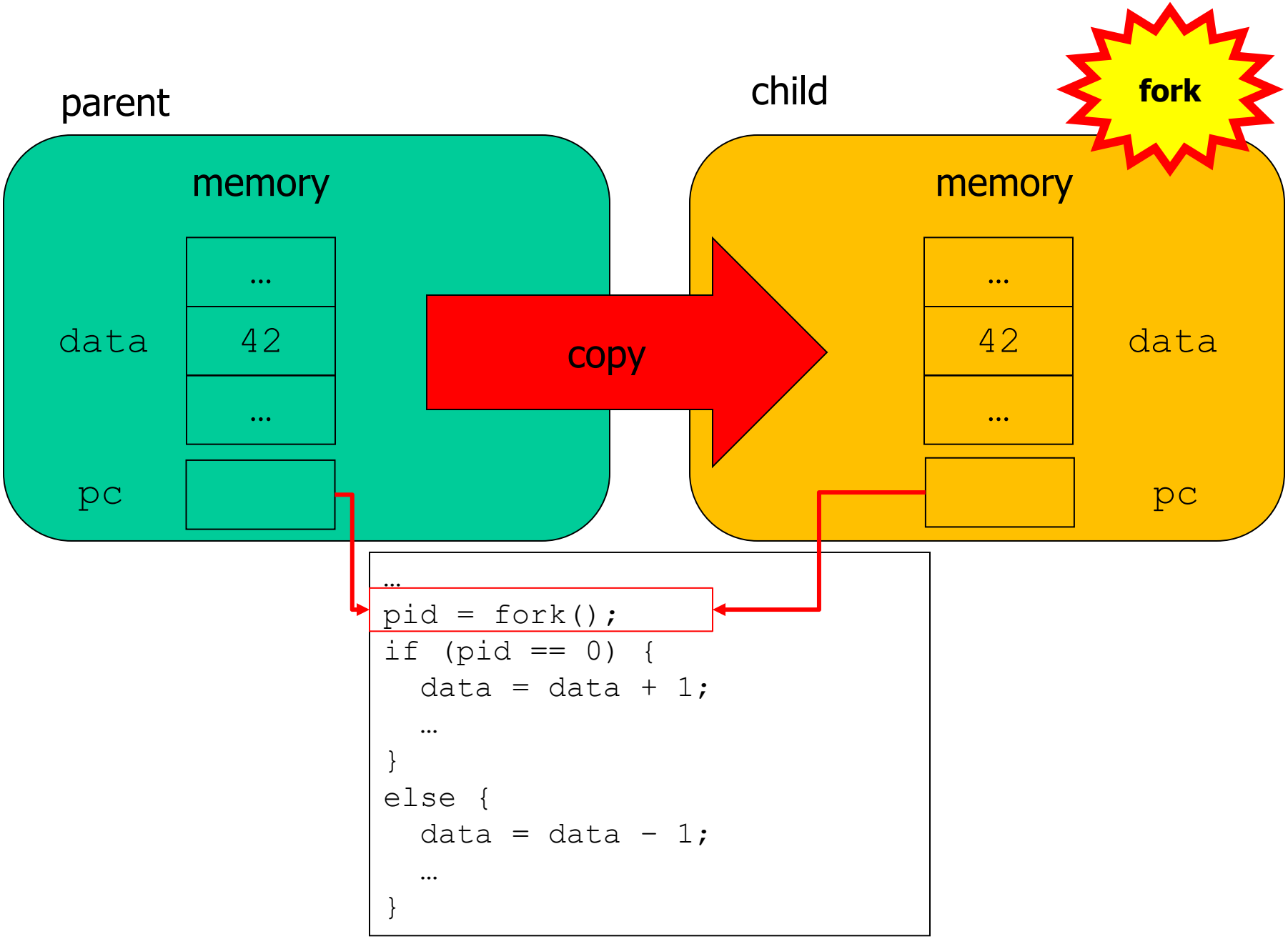
Μια πρώτη μορφή «επικοινωνίας»

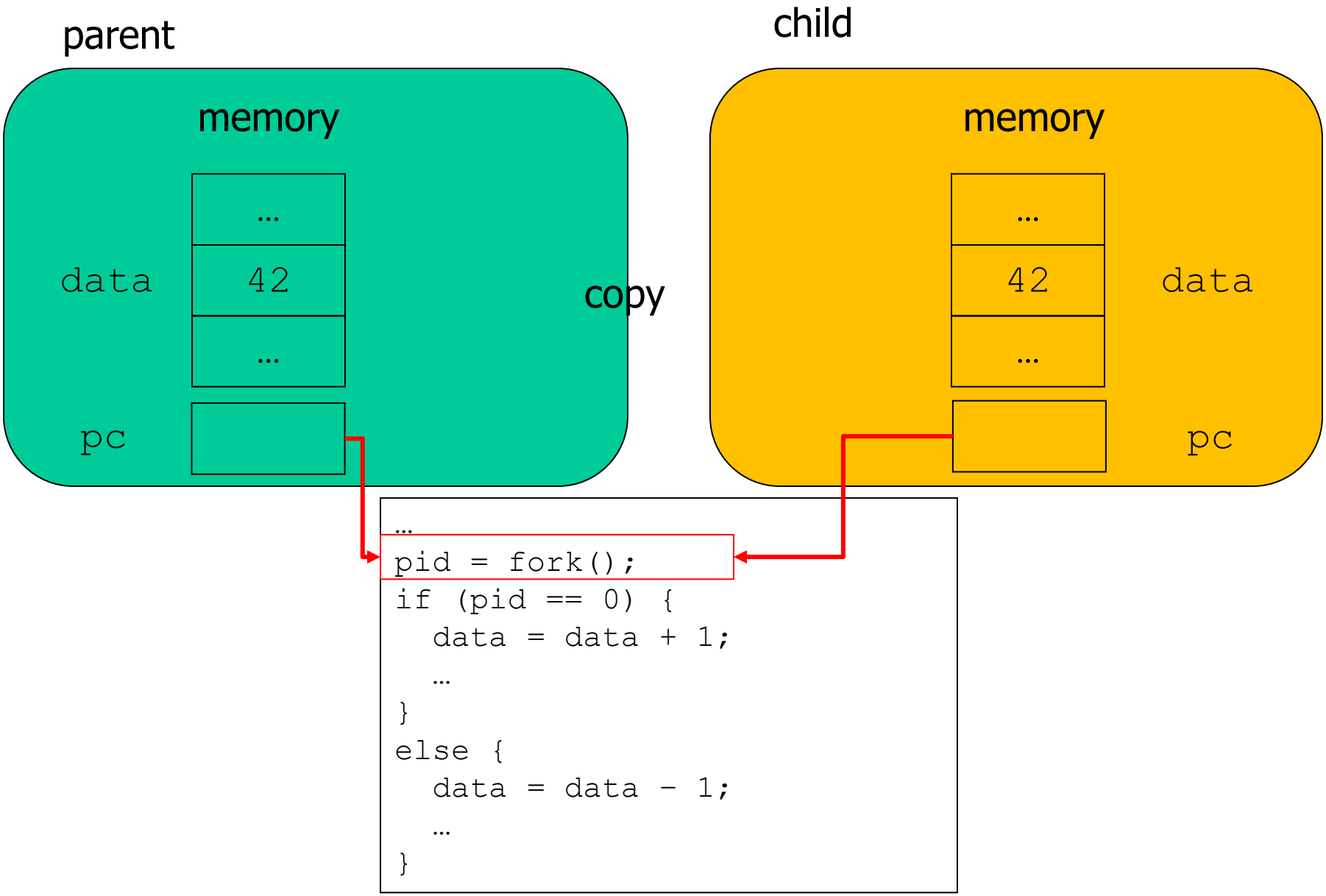
- Η `fork` αντιγράφει στο παιδί την μνήμη του γονιού
- Ο γονιός μπορεί να αναθέσει σε κάποιες μεταβλητές (θέσεις μνήμης) τιμές / δεδομένα που επιθυμεί να «περάσει» στο παιδί, **προτού** δημιουργήσει το παιδί
- Μετά την κλήση της `fork`, όλες οι μεταβλητές του προγράμματος που εκτελεί το παιδί θα έχουν τις ίδιες τιμές με αυτές του γονιού

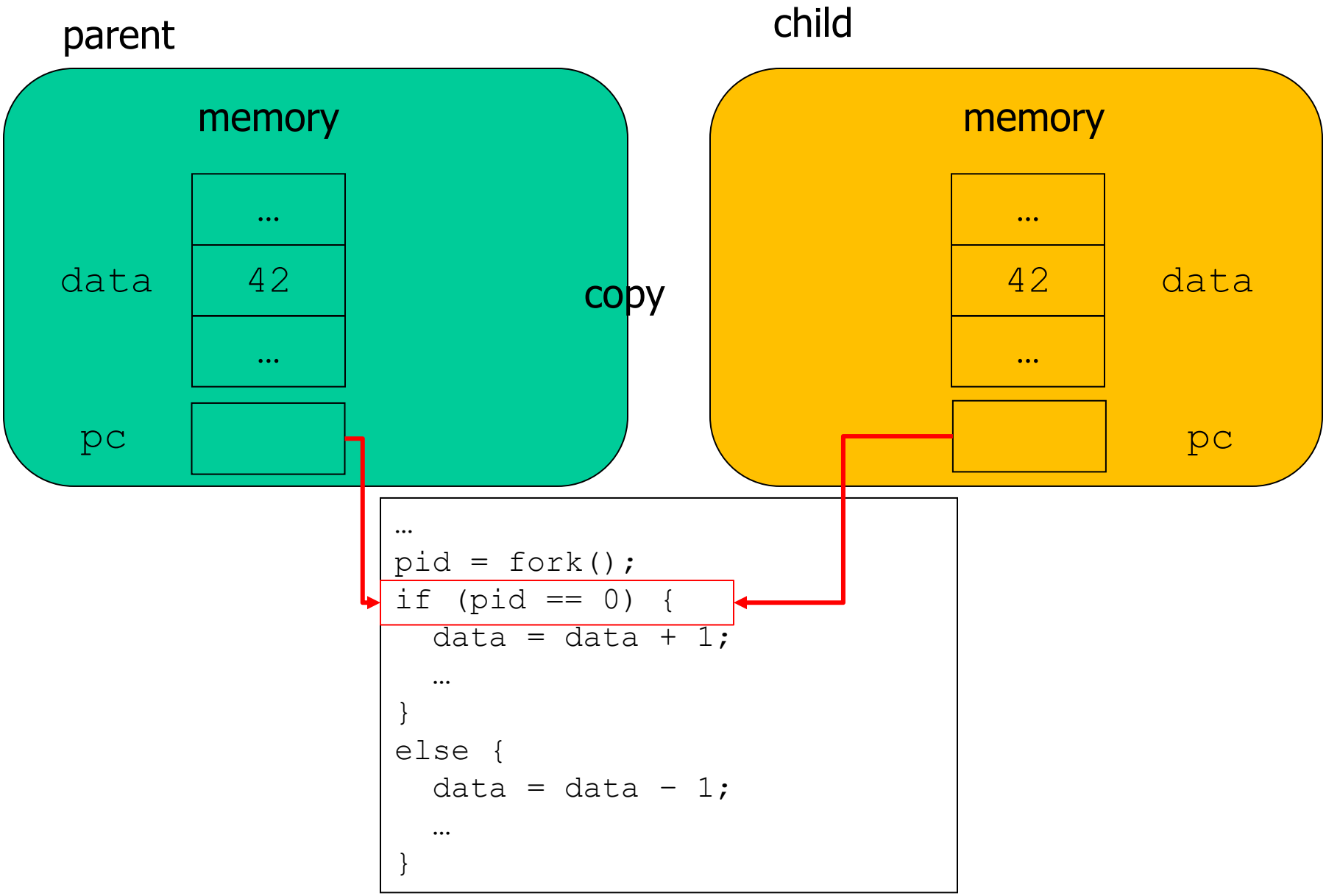
- Στην συνέχεια, αφού κάθε διεργασία έχει ξεχωριστή μνήμη, οι όποιες αλλαγές γίνονται στην μνήμη μιας διεργασίας είναι ορατές **μόνο** στην ίδια
  - δεν υπάρχει κάποια «κρυφή» μετάδοση δεδομένων ανάμεσα στην διεργασία γονιό και την διεργασία παιδί

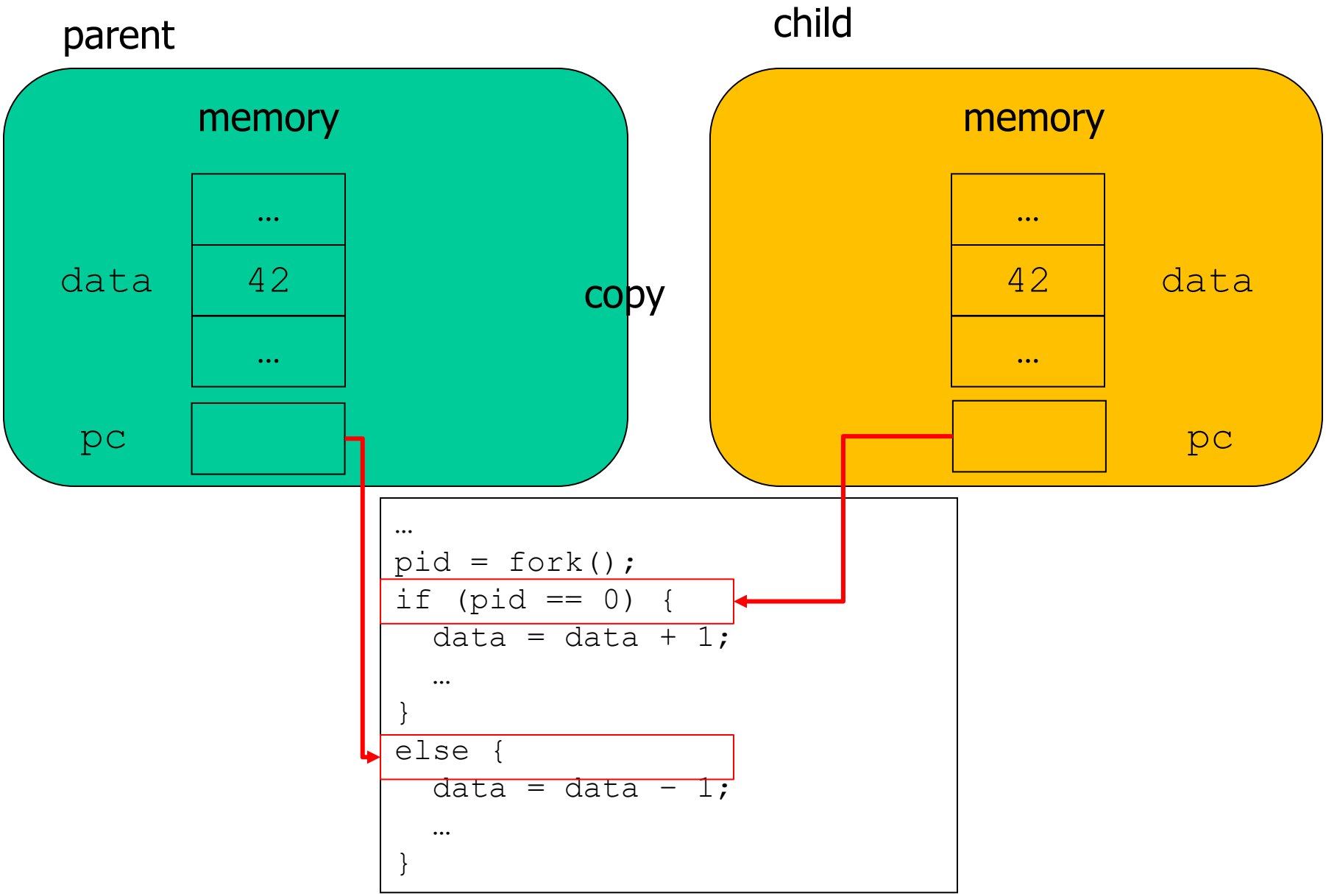
```
int main(int argc, char *argv[]) {
  int pid;
  int data = 42;

  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);
  }
}
```

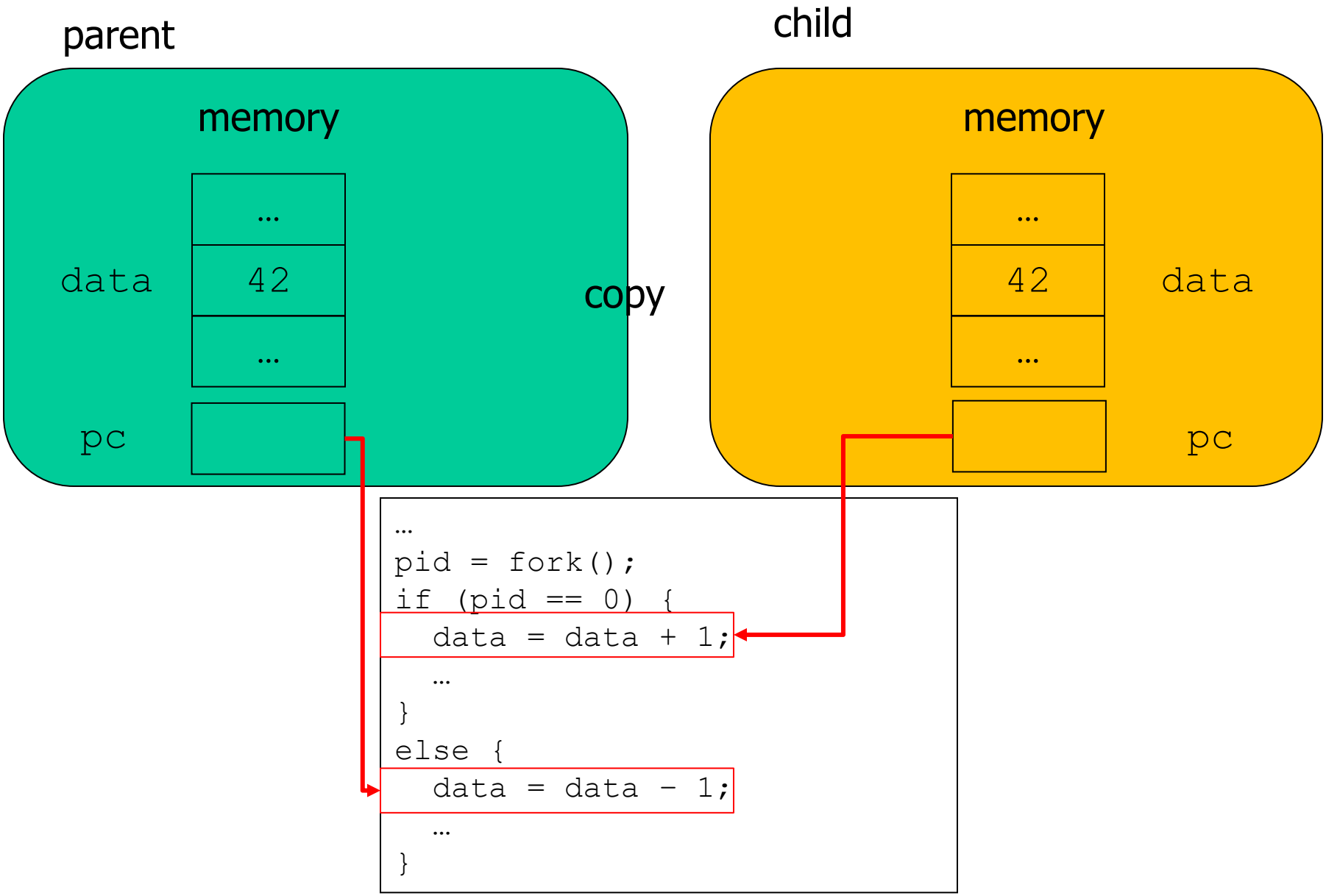

parent
memory
...
data
42
...
pc
...
pid = fork();
if (pid == 0) {
data = data + 1;
...
}
else {
data = data - 1;
...
}

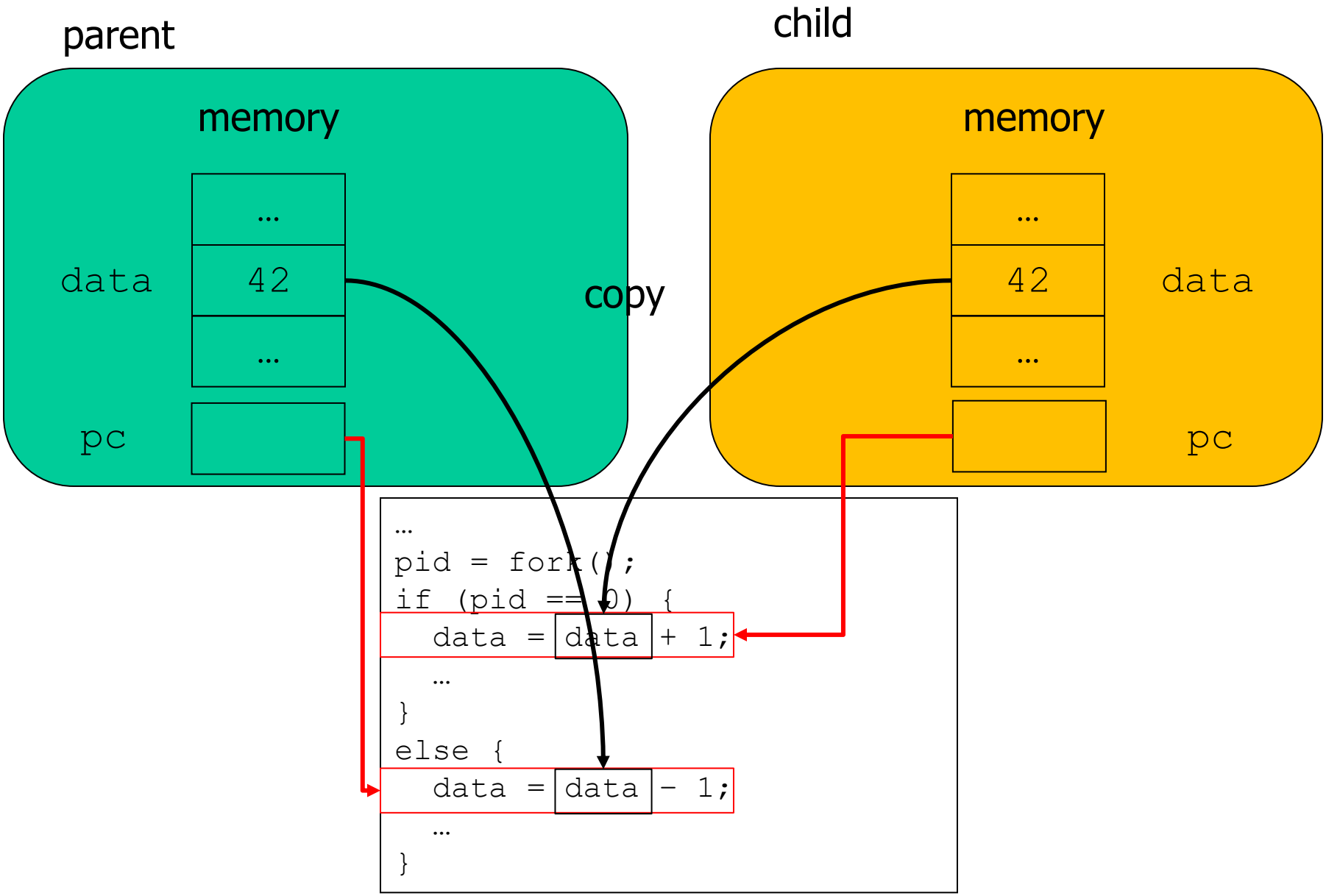

parent
child
fork
memory
memory
...
data
42
...
copy
...
42
data
...
pc
pc
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}

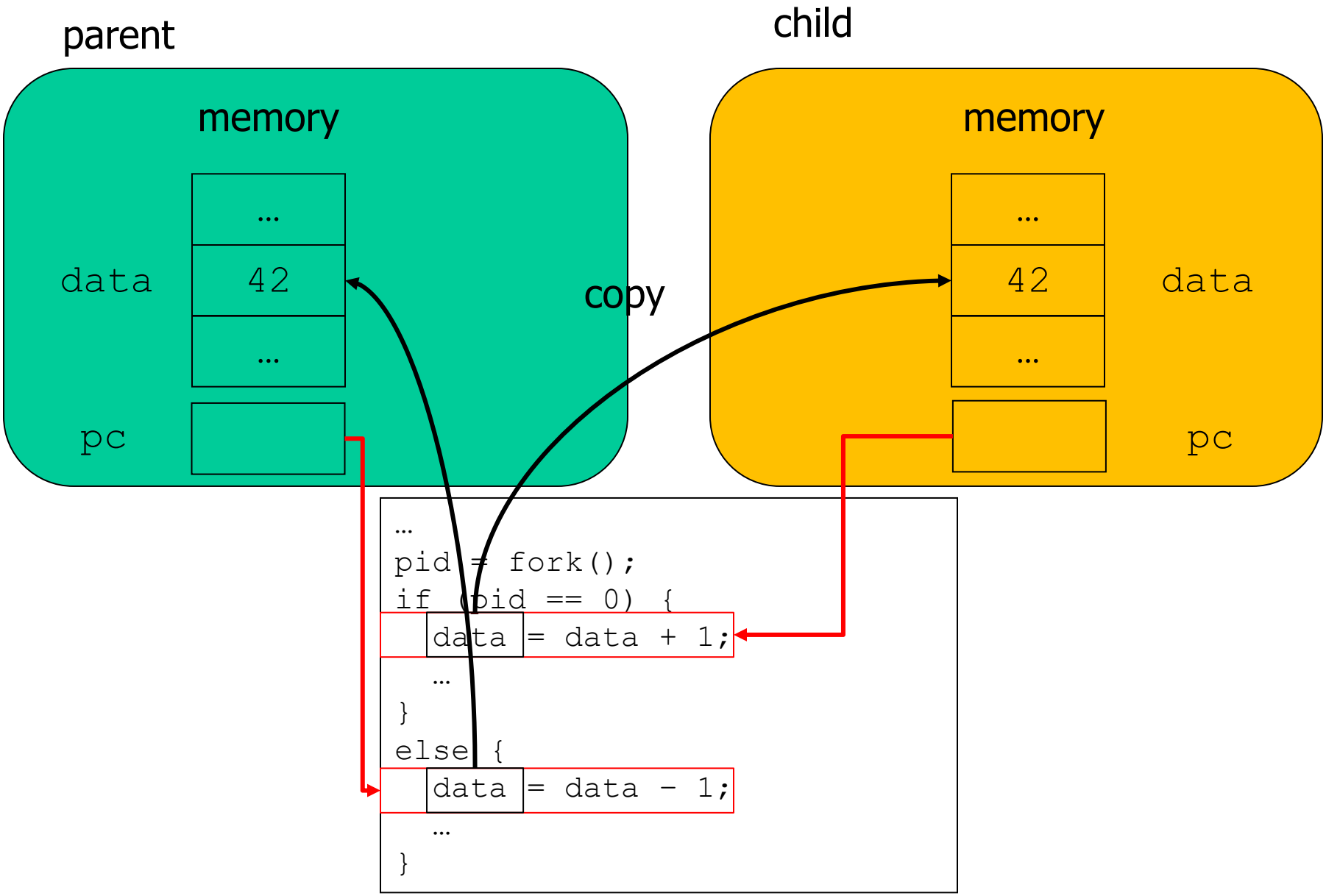

parent
child
memory
memory
...
data
42
...
copy
...
42
data
...
pc
pc
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}

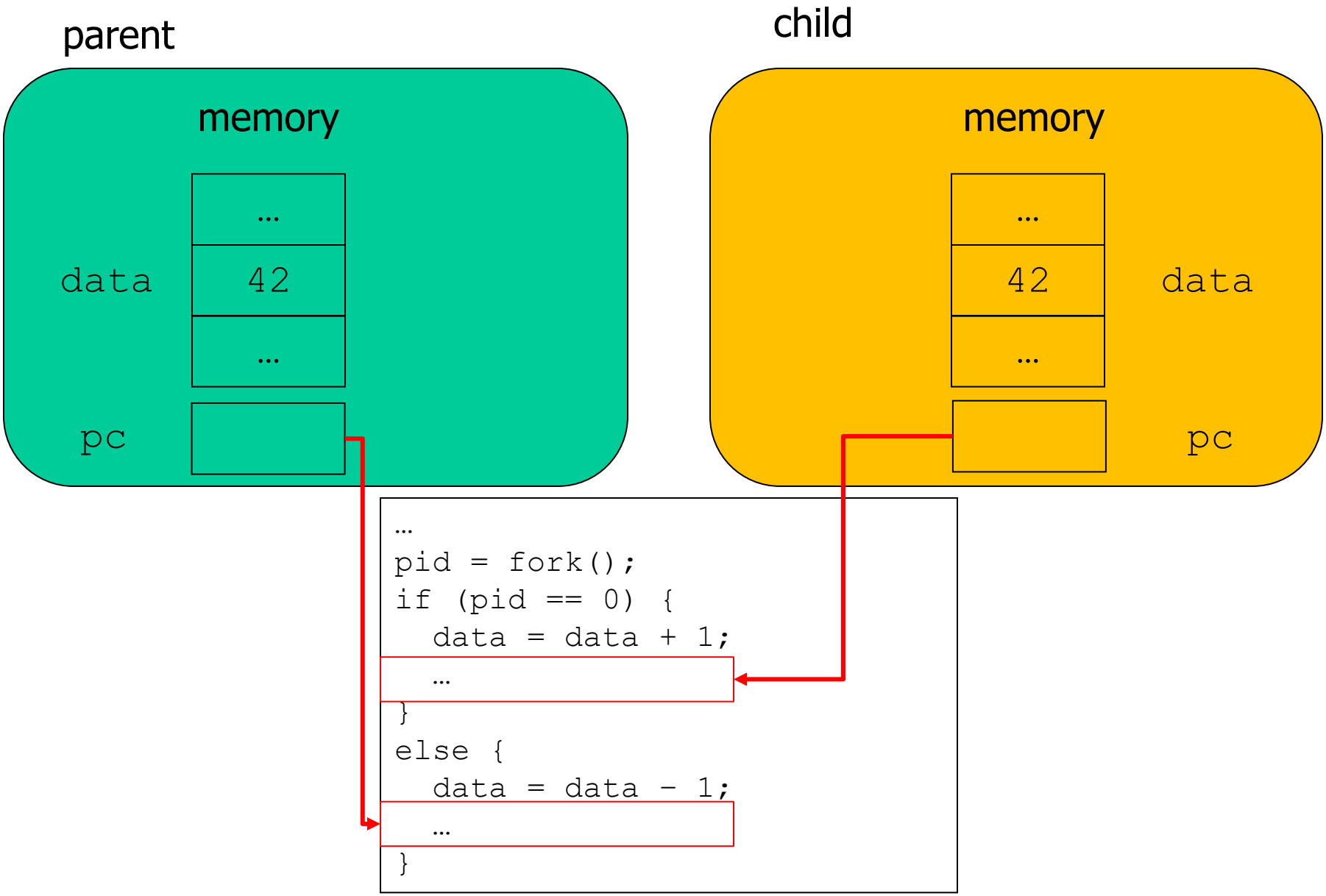

parent
child
memory
memory
...
data
42
...
copy
...
42
data
...
pc
pc
…
pid = fork();
if (pid == 0) {
data = data + 1;
…
}
else {
data = data - 1;
…
}


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


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


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


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


parent
child
memory
memory
...
42
...
data
data
pc
pc
…
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 καθολικών μεταβλητών, ενδιάμεσες αποθήκες δεδομένων, ...

- Αυτό μπορεί να οδηγήσει σε «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
  - κλήση επιπλέον διαδικασιών «καθαρισμού» του χρήστη
- Μπορεί να κληθεί από **οποιοδήποτε** σημείο και μέσα από οποιαδήποτε συνάρτηση του προγράμματος
  - ισοδυναμεί με κλήση της `return` από την `main`
- Η τιμή επιστροφής `status` πρέπει να είναι μια τιμή (`0..255`) ώστε να χωράει σε ένα byte
  - συνήθως `EXIT_SUCCESS` (=`0`) ή `EXIT_FAILURE` (=`1`)
- Το λειτουργικό κρατά αυτή την τιμή
  - 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 για την οποία η πληροφορία τερματισμού αποθηκεύεται στην διεύθυνση που δείχνει ο δείκτης `status`

Ερμηνεία πληροφορίας τερματισμού

- Η πληροφορία τερματισμού κωδικοποιείται σε έναν ακέραιο και πρέπει να **ερμηνευθεί** κατάλληλα
  - συμπεριλαμβάνει την τιμή επιστροφής της διεργασίας + άλλα

Αυτό γίνεται μέσω ειδικών μακροεντολών

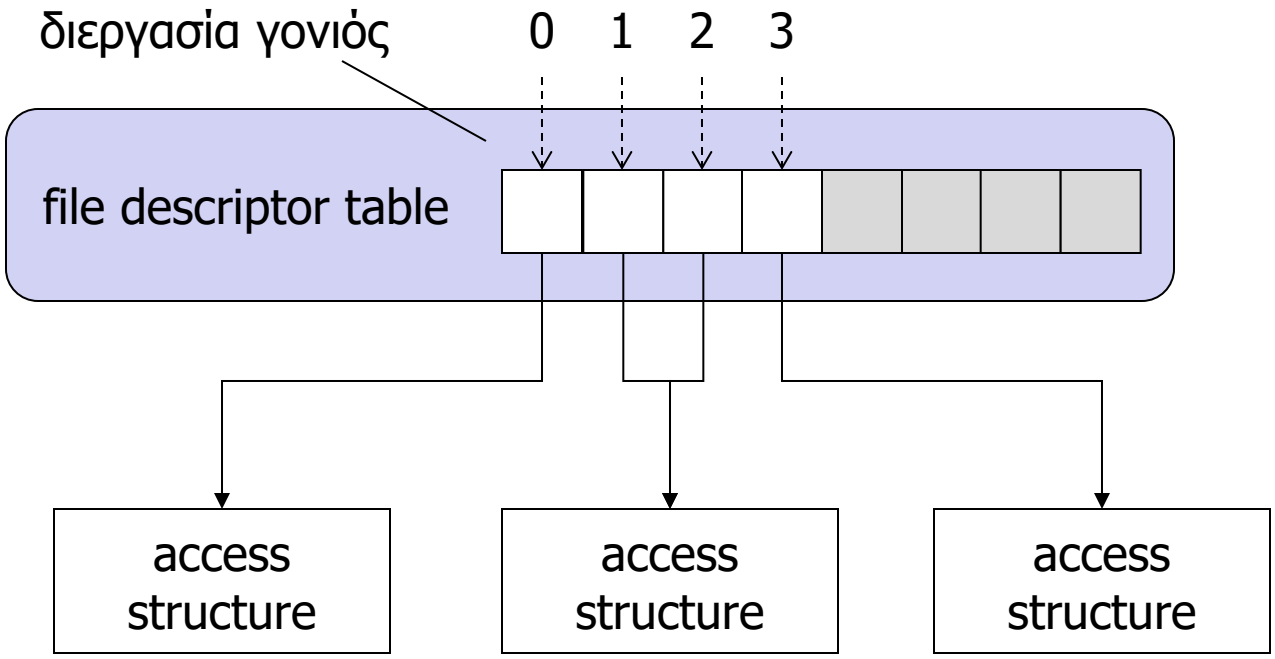
- `WIFEXITED()`: `true` αν το παιδί τερματίστηκε κανονικά (με `return` ή `_exit` ή `exit`)
- `WEXITSTATUS()`: η τιμή (1 byte) που επέστρεψε το παιδί μέσω `return` ή `_exit` ή `exit`
- `WIFSIGNALED()`: `true` αν το παιδί δεν τερματίστηκε κανονικά (αλλά μέσω σήματος)
- `WTERMSIG()`: ο αριθμός του σήματος που προκάλεσε τον τερματισμό του παιδιού

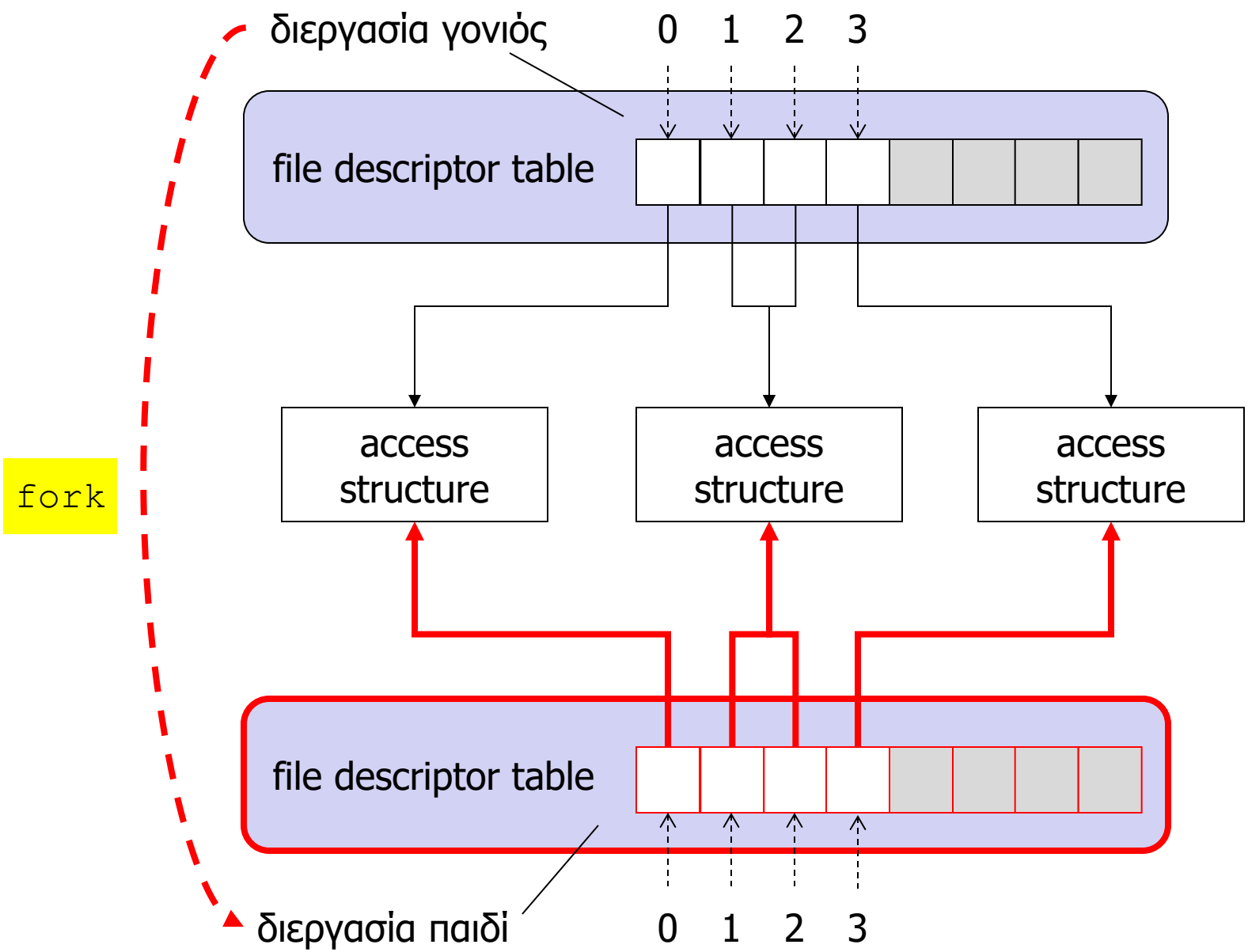
```
int main(int argc, char *argv[]) {
  int pid;
  int status;

  pid = fork();

  if (pid == 0) {
    sleep(60);
    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);
}
```

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;
  int 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 termination */
  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

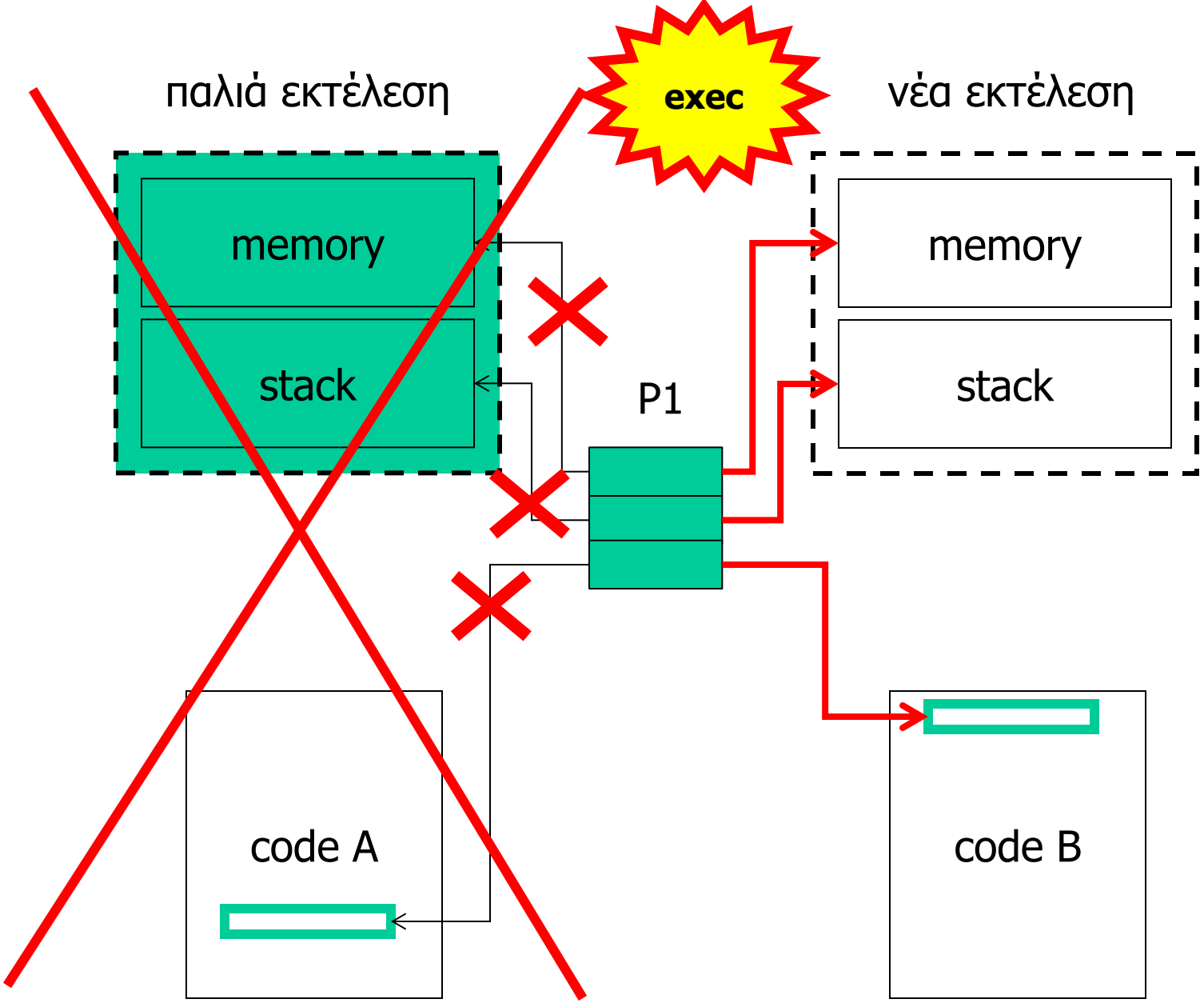
- Ένα παιδί μπορεί να «αυτονομηθεί» πλήρως και να εκτελέσει **διαφορετικό** κώδικα από τον γονιό του
  - συνηθισμένος λόγος για την δημιουργία νέων διεργασιών

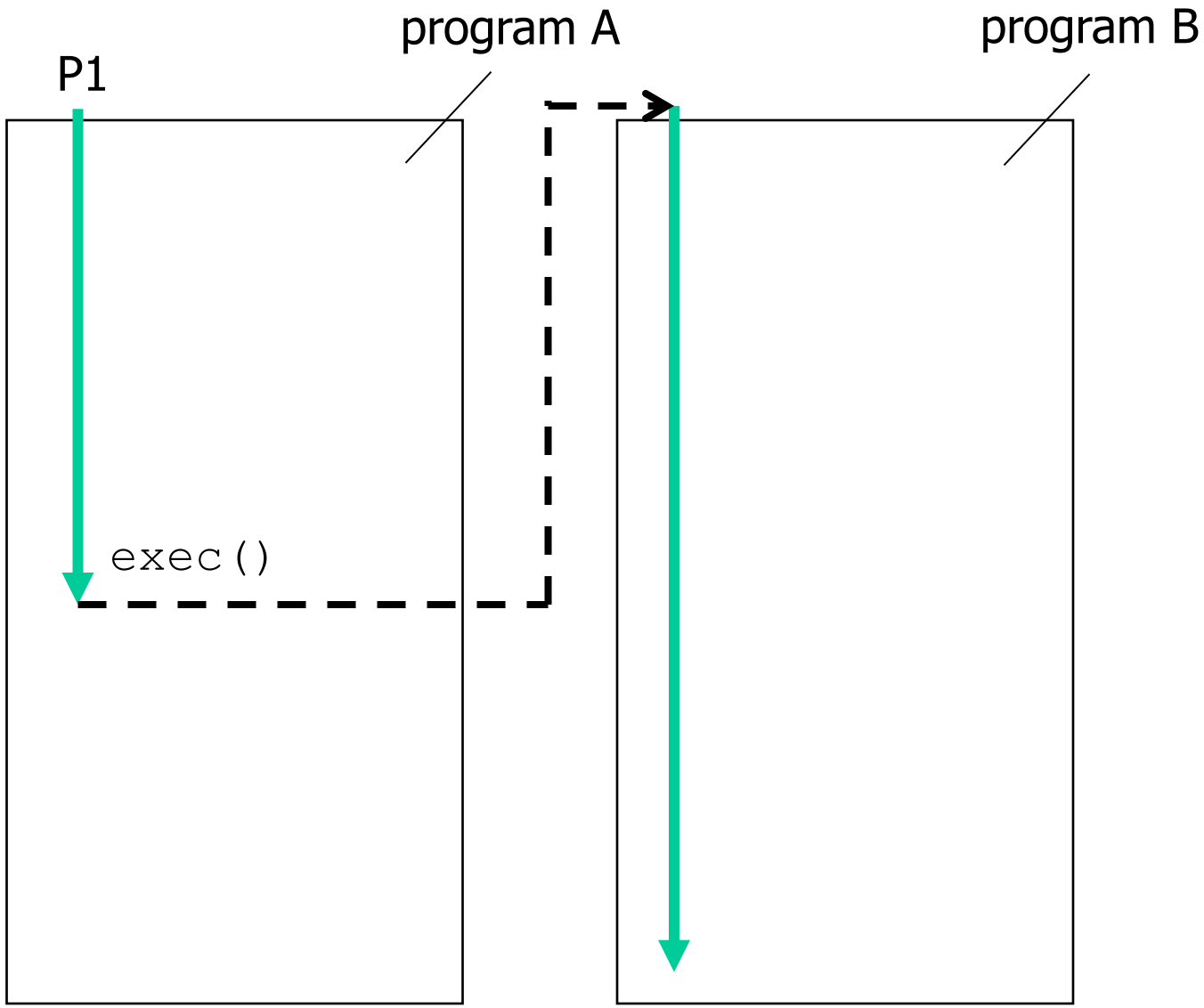
- Το σύστημα είναι **ανοιχτό**/**επεκτάσιμο**
- Μπορεί να γραφτεί κώδικας που κάποια στιγμή θα εκτελέσει κώδικα που **δεν** έχει γραφτεί ακόμα

Εκτέλεση προγράμματος

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

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


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


program A
program B
P1
exec()

Κλασικό μοτίβο εκτέλεσης fork-exec

- Ένα πρόγραμμα Α επιθυμεί να **εκτελέσει** ένα άλλο (διαφορετικό) πρόγραμμα Β έτσι ώστε:
  - να **μην** σταματήσει η εκτέλεση του τρέχοντος προγράμματος Α αλλά μπορεί να **συνεχιστεί** παράλληλα ή αφού πρώτα ολοκληρωθεί η εκτέλεση του προγράμματος Β
  - τα αποτελέσματα της εκτέλεσης του προγράμματος Β να **χρησιμοποιηθούν** στο πρόγραμμα Α
  - το πρόγραμμα Α να μπορεί να **διακόψει** την εκτέλεση του προγράμματος Β ανά πάσα στιγμή
- Αυτό μπορεί να επιτευχθεί πολύ εύκολα αν το πρόγραμμα Α αναθέσει την εκτέλεση του Β σε μια **ξεχωριστή διεργασία** παιδί με `fork-exec`
- Είναι αυτό που κάνει το πρόγραμμα του shell που τρέχει αυτόματα όταν ανοίγουμε ένα τερματικό

πρόγραμμα add

```
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;

  if (!(pid=fork())) {
    printf("child: execute ./add 123 456\n");
    execl("./add", "add", "123", "456", NULL);
    perror("execl");
    return(1);
  }

  waitpid(pid, NULL, 0);
  printf("parent: child terminated\n");
  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**
  - καθώς αρχίζει μια εντελώς καινούργια εκτέλεση κώδικα

- **Προσοχή:** η διεργασία παραμένει η **ίδια**

Περιγραφείς αρχείων μετά το `exec`

- Οι περιγραφείς αρχείων που έχει η διεργασία τη στιγμή που κάνει `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 προτού αρχίσει η εκτέλεση του μέσω `exec`
  - **εν αγνοία** του προγράμματος που εκτελείται στην συνέχεια

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

  if (!(pid=fork())) {
    printf("child: execute ./add 123 456 with redirection\n");
    fd = open("test", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU);
    dup2(fd, STDOUT_FILENO); // redirect stdout
    close(fd);
    execl("./add", "add", "123", "456", NULL);
    perror("execlp");
    return(1);
  }

  waitpid(pid, NULL, 0);
  printf("parent: child terminated\n");
  return(0);
}
```

Ορφανές διεργασίες και ζόμπι

- Αν τερματίσει ο γονιός μιας διεργασίας, η διεργασία παιδί συνεχίζει την εκτέλεση της αλλά μένει **ορφανή**
  - οι ορφανές διεργασίες υιοθετούνται από τον στενότερο συγγενή τους
- Αν τερματίσει ένα παιδί μιας διεργασίας, παραμένει σε κατάσταση **ζόμπι**
  - το λειτουργικό **συνεχίζει** να κρατά πληροφορία για την διεργασία (λόγος τερματισμού, τιμή επιστροφής κτλ), για να μπορεί να παραλάβει αυτή την πληροφορία ο γονιός
- Μια διεργασία παραμένει σε κατάσταση ζόμπι μέχρι ο γονιός της (α) να καλέσει την λειτουργία αναμονής για τερματισμό του παιδιού ή (β) να τερματίσει

Ομάδες διεργασιών (process groups)

- Κάθε διεργασία ανήκει σε μια **ομάδα** (process group)
- Το αναγνωριστικό ομάδας (process group ID – pgid) είναι το ίδιο με το αναγνωριστικό του πρώτου μέλους της (process group leader)

- Τα παιδιά **κληρονομούν** την ομάδα του γονιού
- Το αναγνωριστικό της ομάδας μιας διεργασίας μπορεί να ληφθεί/αλλάξει μέσω `getpgid`/`setpgid`

- Η ομαδοποίηση **απλοποιεί την «μαζική» διαχείριση** των διεργασιών
  - αποστολή σήματος σε μια ομάδα διεργασιών

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

  pid = fork();

  if (!pid) {
    printf("child: PID=%d, PPID=%d, PGID=%d\n",
                   getpid(), getppid(), getpgrp());
    while (1) {
      printf("child: ping\n");
      sleep(3);
    }
    return(0);
  }

  printf("parent: PID=%d, PPID=%d, PGID=%d\n",
                  getpid(), getppid(), getpgrp());
  while (1) {
    printf("parent: ping\n");
    sleep(3);
  }
  return(0);
}
```

και οι 2 διεργασίες ανήκουν στην ίδια ομάδα (αυτή του γονιού)

θα τερματιστούν και οι 2 με Ctrl+C

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

  pid = fork();

  if (!pid) {
    setpgid(0,getpid()); /* create my own group */
    printf("child: PID=%d, PPID=%d, PGID=%d\n",
                   getpid(), getppid(), getpgrp());
    while (1) {
      printf("child: ping\n");
      sleep(3);
    }
    return(0);
  }

  printf("parent: PID=%d, PPID=%d, PGID=%d\n",
                  getpid(), getppid(), getpgrp());
  while (1) {
    printf("parent: ping\n");
    sleep(3);
  }
  return(0);
}
```

το παιδί δημιουργεί την δική του ομάδα

δεν τερματίζεται με Ctrl+C