Προηγμένα θέματα λειτουργικών συστημάτων

Άσκηση

Περί τίνος πρόκειται

- Μια δυναμικά εξελισσόμενη άσκηση η οποία θα αναπτύσσεται βάσει της θεωρίας του μαθήματος
- Στόχος είναι να εξερευνήσουμε κάποιες από τις έννοιες του λειτουργικού συστήματος σε τεχνικό επίπεδο
- Καθότι είναι μια agile διαδικασία απαιτείται η χρήση κάποιων εργαλείων όπως του **git**.
- Καθότι έχουμε variance μηχανημάτων απαιτείται η χρήση εικονικών μηχανών

Ανάπτυξη δέντρου διεργασιών

- Να γραφτεί πρόγραμμα σε γλώσσα προγραμματισμού C και περιβάλλον Linux στο οποίο η διεργασία πατέρας (F) δημιουργεί N διεργασία (C1, .. Cn).

- Οι διεργασίες F και Cs γράφουν από ένα μήνυμα σ’ ένα αρχείο.

- Το όνομα του αρχείου προσδιορίζεται από τον χρήστη ως όρισμα από την γραμμή εντολών κατά την εκτέλεση του προγράμματος

- Το πλήθος των διεργασιών παιδιών προσδιορίζεται από το χρήστη ως όρισμα από την γραμμή εντολών κατά την εκτέλεση του προγράμματος

Δομή του αρχείου εξόδου

[PARENT] —> <PID>

[CHILD] —> <PID>

[CHILD] —> <PID>

[CHILD] —> <PID>

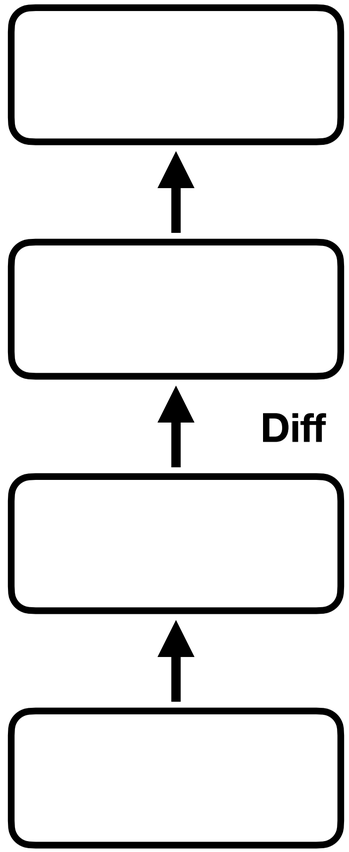
[CHILD] —> <PID>

[CHILD] —> <PID>

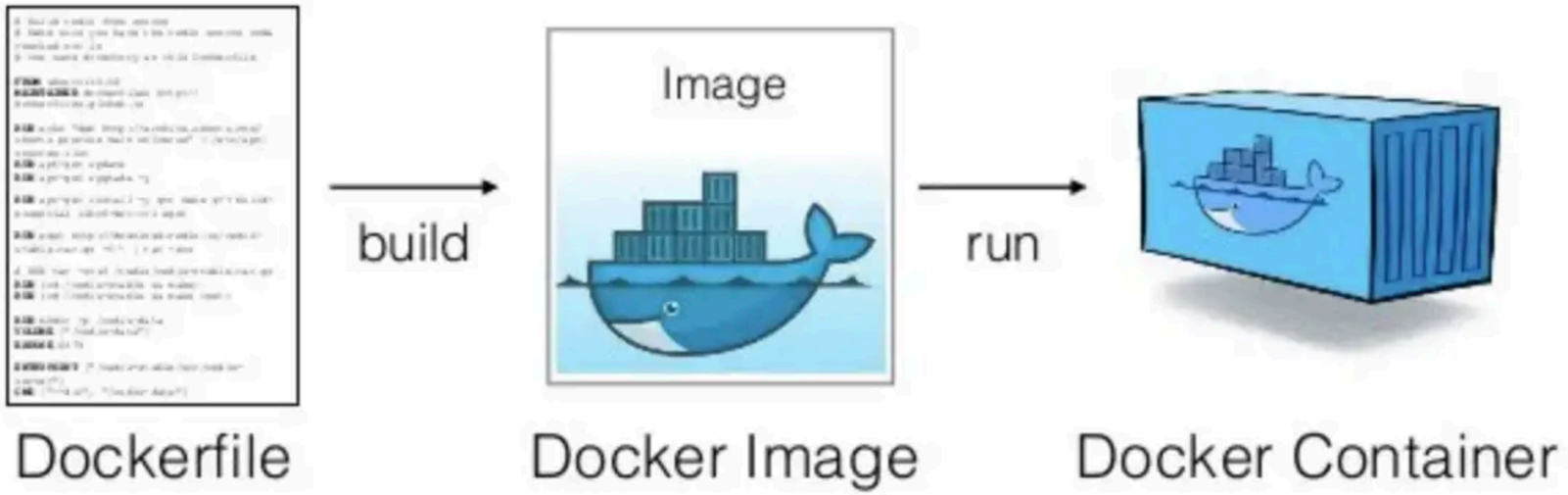
Git basics

```
commit e3b9e3aa69c3d1d7d343cf9f9b5f2703bf9d8f4f
Author: Christos Andrikos <candrikos@Christoss-MacBook-Air.local>
Date:   Fri Mar 26 15:06:30 2021 +0200

    feat(pipelines): generic pipeline for
```



Containerization …



Επικοινωνία πατέρα-παιδιού

- Επέκταση του κώδικα που έχουμε ήδη αναπτύξει στο προηγούμενο στάδιο προκειμένου να υποστηρίζεται επικοινωνία πατέρα παιδιού μέσω pipes.
- Ο πατέρας διατηρεί ανοιχτά pipes για να επικοινωνήσει με τα n παιδιά του. Εδώ θα πρέπει να δημιουργηθεί μια έξυπνη **δομή αναφοράς*** στα παιδιά.
- Ο πατέρας στέλνει στα παιδιά του ένα μήνυμα της μορφής:
  - “Hello child, I am your father and I call you: <childName>.”
- Το παιδί εκτυπώνει πλέον στο αρχείο τη γραμμή:
  - <childPID> —> <childName>
- Το παιδί ενημερώνει τον πατέρα για το πέρας της εκτύπωσης με το μήνυμα “done”.

Τεχνικές λεπτομέρειες

- Έξυπνη δομή αναφοράς: Αν σκεφτούμε ότι τα ο πατέρας πρέπει να μπορεί να στέλνει μνήματα σε n παιδιά, και προφανώς όχι σειριακά, τι struct πρέπει να δημιουργήσουμε;

- Μπορούμε να επιτύχουμε πραγματικά ασύγχρονη επικοινωνία και από ποια πλευρά μας ενδιαφέρει αυτό;

  - https://linux.die.net/man/2/select

N “done”s from the same child!

- Μάλλον η εύκολη λύση που υλοποιήθηκε ως τώρα εξυπηρετεί τη “σύγχρονη” αποστολή μηνυμάτων (αν όχι μπορείτε να παραλείψετε αυτό το βήμα)
- Τι συμβαίνει μόλις ο πατέρας λάβει ένα μήνυμα “done” από ένα συγκεκριμένο παιδί; Μπορεί να του στείλει ακόμη ένα μήνυμα;
- Ζητείται η επέκταση του κώδικα προκειμένου να υποστηρίζει αυτή τη λειτουργικότητα

Worker Pool

- Έχοντας ένα πατέρα με n παιδιά, μπορούμε να υποθέσουμε το worker pool paradigm όπου ένας πατέρας ζητά από τα παιδιά που είναι idle, να εκτέλέσουν μια εργασία για λογαριασμό του
- Τα παιδιά μπορούν εκτελούν την εργασία και να απαντούν το αποτέλεσμα στο αντίστοιχο channel όπως πριν απαντούσαν done
- Έτσι καταλήγουμε να έχουμε Worker Pool το οποίο προαφανώς εκκινεί σωστά αλλά θα πρέπει και να τερματίζει ορθά!

Gracefull shutdown

- Τα παιδιά ενός πατέρα θεωρούνται από προγραμματική πλευρά τμήμα “κτήμα” του.
- Ο προγραμματιστής οφείλει να προσθέσει κώδικα τέτοιο ώστε τερματισμός του πατέρα να τερματίζει όλα τα παιδιά προκειμένου να απελευθερώνονται ιεραρχικά όλοι οι πόροι του συστήματος.
- Παράδειγμα:
  - έστω ότι προσθέτω ένα busy loop στον κώδικα πατέρα.
  - Στη συνέχεια στέλνω ctrl-c (sigkill) ενώ υπάρχουν ζωντανά παιδιά
  - Τι πρέπει να γίνει; —> το ctrl-c (sigkill) πρέπει να διαδοθεί σε όλα τα παιδιά, εκείνα να τερματίσουν και μετά να τερματίσει ο πατέρας. Σε περίπτωση που δεν γίνει αυτό αφήνουμε το λειτουργικό να λάβει δράση και προφανώς αφήνουμε το σύστημά μας σε “unknown” state!

Gracefull shutdown - explained

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

pid_t child_pid = -1 ; //Global

void kill_child(int sig){
    kill(child_pid,SIGTERM);
}

void on_sig_term(int sig){
  printf("I am child, my father just terminated me!\n");
  exit(0);
}

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


    child_pid = fork();

    if (child_pid > 0) {
     /*PARENT*/
      signal(SIGALRM,(void (*)(int))kill_child);
      alarm(10);
      wait(NULL);
      printf("Child has been gracefully shut down\n");
    }
    else if (child_pid == 0){
      /*CHILD*/
      signal(SIGTERM, (void (*)(int))on_sig_kill);
      printf("This is a test\n");
      sleep(60);
    }
}
```

Signal handler definitions

SIGALARM schedule

Signal handler registrations

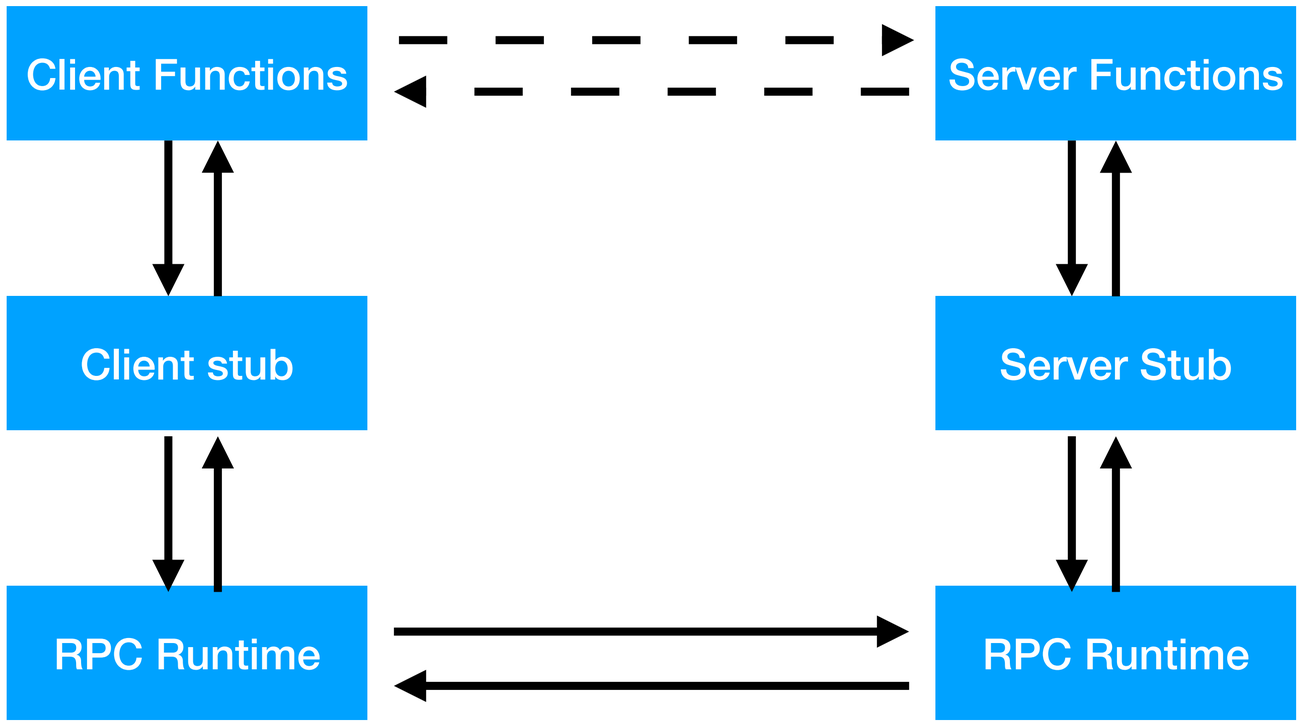
RPC

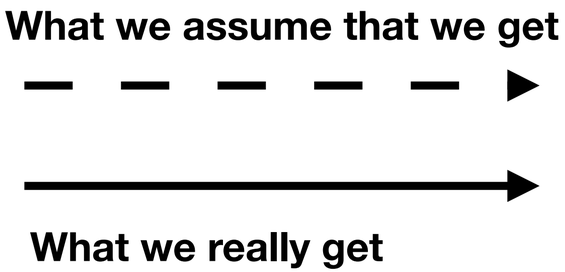
- Έχοντας υλοποιήσει τα ακόλουθα:
    1. Έναν πατέρα που γενιά Ν παιδιά
    2. Επικοινωνεί με τα παιδιά του μέσω pipes
    3. Τερματίζει σωστά τα παιδιά του σε περίπτωση τερματισμού
- Έχουμε όπως είπαμε ένα Worker Pool σύστημα…
- Αλλά πώς θα μπορούσαμε να αξιοποιήσουμε αυτό το σύστημα?

RPC basics

- Client-Server communication protocol
- Client performs remote procedure calls
- Remote procedures are part of the code of the server
- Remote procedures are exposed to the code of the client (transparency)

RPC


Client Functions
Server Functions
Client stub
Server Stub
RPC Runtime
RPC Runtime


What we assume that we get
What we really get

Tech Recipe

- Stack:
  - gcc —> c compiler
  - rpcgen —> protocol compiler
  - rpcbind —> the rpc runtime, a daemon.

RPC ADD example

add.x

Arguments

```
struct numbers{
        int a;
        int b;
};

program ADD_PROG{
    version ADD_VERS{
        int add(numbers)=1;
    }=1;
}=0x23459911;
```

RPC program definition

The file to describe the protocol

- Let's build some boilerplate code:

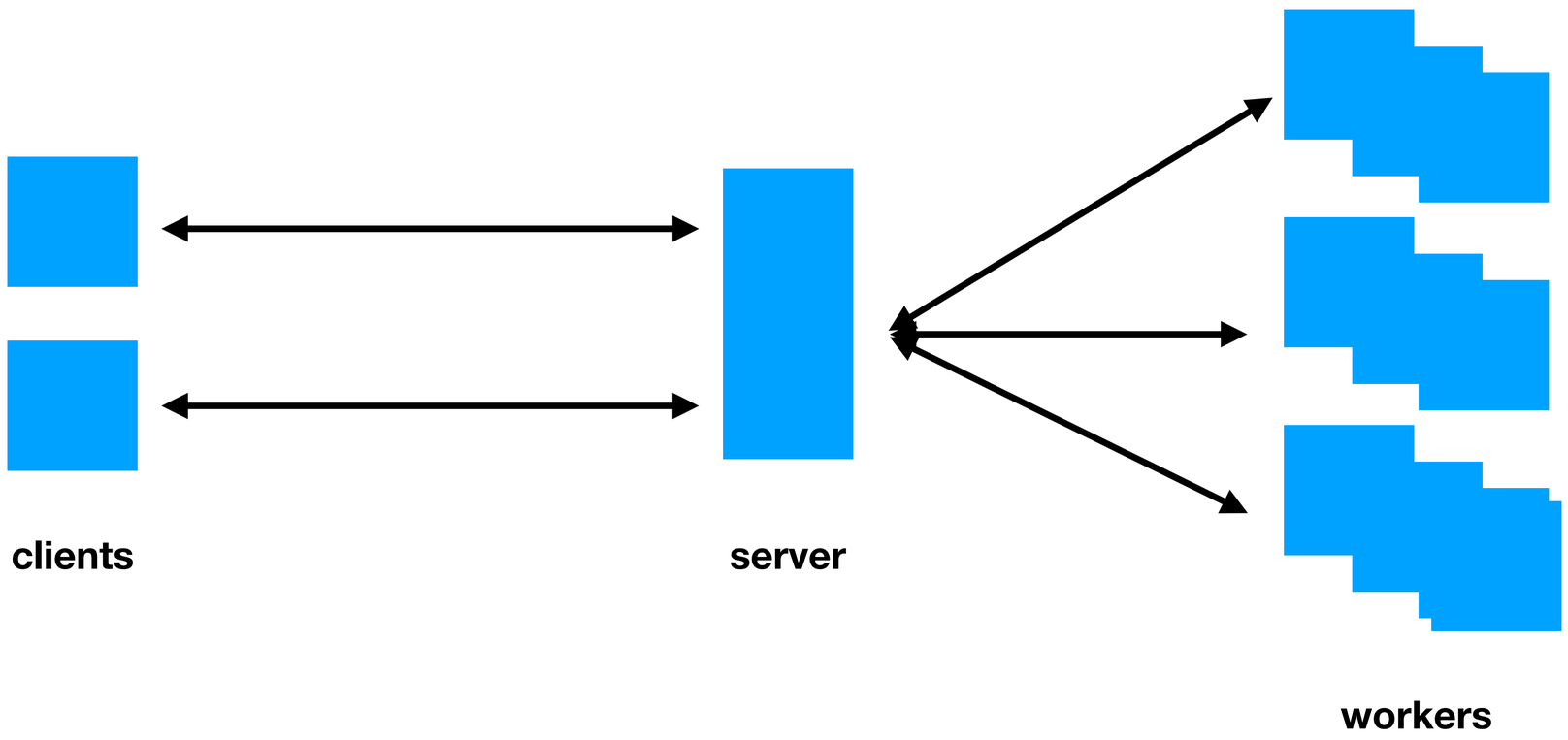
  rpcgen -a -C add.x

- Checkout the generated files:
  - add_server —> the server code
  - add_client —> the client code
- Let's build:

  make -f Makefile.add

- Run:
  - ./add server (start the server)
  - ./add_client localhost <args>
- Hint: Nothing happens! We need to update both add_server.c and add_client.c to get the expect behaviour ….

Η τελική αρχιτεκτονική



Τι κάνει όμως το σύστημα?

- Ως εδώ τίποτα καθότι έχουμε στήσει (περίπου) ένα framework!
- Δηλαδή ένα πολύ βασικό μηχανισμό να κάνουμε κατανεμημένους υπολογισμούς
- Δηλαδή ποιος είναι ο κώδικας που θα τρέχει στους workers?
  - Για να μπορέσουμε να αξιολογήσουμε την υλοποίησή μας αρκεί οι workers να εκτυπώνουν κάποια μηνύματα για την έναρξη εργασιών τους και τα αντίστοιχα για τη λήξη αυτών και ενδιάμεσα να κοιμούνται για όσο χρόνο τους ορίζει ένα όρισμα n.

Δηλαδή

```
struct duration{
        int seconds;
};

program SLEEP_PROG{
    version SLEEP_VERS{
        void add(duration)=1;
    }=1;
}=0x31599123;
```

- Αυτό θα μπορούσε να είναι ένα πρωτόκολλο
- Και αρκεί ο κώδικας πελάτη να έτρεχε ένα μεγάλο αριθμό από requests … που θα έθεταν όλοιυς τους workers σε κατάσταση sleep
- Πως θα αντιμετωπιστεί η περίπτωση request flood?

Επεκτάσεις:

- Πώς μπορεί να επεκταθεί η υποδομή για να έχουμε workers με διαφορετικό profile?
    - Ετερογενή workloads
- Πως θα μπορούσε να επεκταθεί ή αρχιτεκτονική για να εισάγουμε έννοιες προτεραιότητας: tasks υψηλής και χαμηλής προτεραιότητας …