OS Project 2

Βησσαρίων Μουτάφης sdi1800119@di.uoa.gr
1115201800119

Notes για τους διορθωτές:

- Έχω αλλάξει τον τρόπο εκτύπωσης τών πρώτων για την διευκόλυνση του parsing στο IPC.
- Έχω δημιουργήσει ένα subprocess στα workers για να εκτελώ τον αντίστοιχο prime algorithm και ταυτόχρονα να θέτω ένα protocol για signal handling/sending (δες στο αντίστοιχο section).
- Το IPC γίνεται με την μεταβίβαση των μηνυμάτων σε string-batches.
- Γίνεται χρηση της sigaction και της sigqueue για πιο reliable signal handling.
- Ορίζεται πρωτόκολλο για το signal handling, αλλά το default version δεν το χρησιμοποιεί γιατί σε ορισμένα program runs προσθέτει overhead.
- Όλα τα processes που γράφουν σε pipe-write-end, τα έχω βάλει να εκτυπώνουν στο stdout και για να μπορεί να ανακατευθύνεται το output στο κατάλληλο file descriptor, χρησιμοποιώ την κλήση dup2 για να κάνω duplicate το pipe[WRITE] στο STDOUT_FILENO.

Compilation and Run

Για το compilation και run της εφαρμογής παρέχω ένα Makefile. Οι πιθανές εντολές είναι:

```
~$ make # compile the app
~$ make run # compile & run the app with no special protocol to capture signals
~$ make [run] PROTOCOL=1 # compile (and run) the program with a specific protocol established
~$ make clean # uninstall the app
~$ make re-run [PROTOCOL=1 (optional)] # clean install and run with or without the protocol
```

Χρησιμοποιούμε separate compilation με objective files ώστε να μην χρειάζεται να ξαναφτιάχνουμε τα παντα όταν αλλάζουμε ένα μόνο αρχείο. Το sig-protocol ενεργοποιείται με την χρήση του macro ifdef, στα files *root.c* και *workers.c*. Δες παρακάτω για specifics.

Abstract

Η εφαρμογή βασίζεται σε 4 code bases:

- root
- internal
- workers
- primes, utilities, main function (Others).

Το καθένα από τα 3 πρώτα περιλαμβάνει βασικές και process-specific functions, οι οποίες χρησιμεύουν στην διευθέτηση του process behaviour των διάφορων processes. Επίσης υπάρχει ένα file, ονόματι *process-utils.c*, το οποίο περιέχει γενικές συναρτήσεις που χρησιμεύουν σε 2 τουλάχιστον process-types. Με αυτό τον τρόπο πετυχαίνουμε code reduction και ταυτόχρονα simpler process API. Τα utilities προσφέρουν κάποιες χρήσιμες συναρτήσεις για το manipulation strings και για άλλες χρήσιμες λειτουργίες, πολλές από αυτές είναι κοινές με το OS-Prj1 (ο φάκελος είναι ίδιος απλά εμπλουτισμένος). Τέλος όλες οι main συναρτήσεις, για τα διαφορετικά code bases, τρέχουν μία συνάρτηση που ελέγχει το argv array, για καθαρότερο error checking, κατά την διάρκεια της κατασκευής της δομής της εφαρμογής.

Data Structures Used

Έγινε χρήση μιας Priority Queue, υλοποιημένο με Min Heap (pointer based), για το prime sorting στα ενδιάμεσα level και στο root process.

Η εισαγωγή έγινε σύμφωνα με τον αλγόριθμο heapify-up και η διαγραφή με τον αλγόριθμο heapify-down. Παρέχουμε και διαγραφή σε $O(1)$, καθώς κρατάμε ένα pointer στον previous last node στο complete binary tree, που είναι το heap μας. Ταυτόχρονα παρέχουμε υλοποίηση με void*, κάτι που κάνει τον κώδικα generic, και delete, compare, visit, create του item που θα προσθέσουμε, όπου όλες πέρα από την create αποθηκεύονται στο PQ header. Τέλος, κάνοντας χρήση αναδρομικών κλήσεων καθαρίζουμε το pq καλώντας την αντίστοιχη συνάρτηση, εξασφαλίζοντας ότι δεν θα έχουμε memory leaks.

Common Process Utilities

Οι συναρτήσεις που περιέχονται στο *process-utils.c*, όπως αναφέραμε είναι κοινές για 2 ή περισσότερα process code bases.

Αρχικά έχουμε τις συναρτήσεις για τα internal process nodes, που είναι τα *root* και I_k. Οι 2 process types καλούν την ρουτίνα *internal_process_behaviour*, η οποία καθορίζει τα ranges και τον αριθμό των παιδιών του parent process, δημιουργέι τα παιδιά καλώντας μια process-type specific ρουτίνα *create_children* (δες στη συνέχεια). Τα διαστήματα καθορίζονται βάσει της ρουτίνας *divide_ranges*, η οποία καθορίζει ένα βήμα σύμφωνα με τον αριθμό των παιδιών, αρχίζει να σπάει το αρχικό διάστημα και σταματάει μόνο όταν όλα τα παιδιά έχουν ένα range ή όταν η ανάθεση των range τελείωνει, ότι συμβεί νωρίτερα. Με τον τρόπο αυτό γλιτώνουμε resources που θα γίνονταν allocate σε processes που δεν θα έκαναν τίποτα. Τέλος, η ρουτίνα περιμένει για τα παιδιά του current process, με ένα wait loop.

Άλλη μία σημαντική συνάρτηση, κοινή για root internal processes, αποτελεί η *internal_read_from_child*, η οποία κάνει χρήση polling, για να κάνει monitor τα file descriptor των παιδιών του ώστε να **μην κάνει block-on-read**. Επίσης διαβάζει τους primes και τους εισάγει στο priority queue και τέλος εκτυπώνει τα πάντα με την χρήση batching (δες παρκάτω). Στην περίπτωση που το process είναι το root, το καθορίζουμε με ένα special flag, το οποίο εισάγουμε ως παράμετρο και απλά καθορίζει το πως θα εκτυπωθούν τα primes και τα timestamps.

Τέλος παρέχεται μία συνάρτηση που κλείνει όλα τα sibling-pipes και την χρησιμοποιούμε όταν δημιουργούμε τα un-named pipes.

Root Code Base

Το αρχείο root.c περιέχει όλο τον root-process-specific code. Αρχικά, κάνει argv-error-checking, κάνει install ένα signal handler (δες στο Signal Handling section), κάνει parse τα arguments και αφού τα καθορίσει όλα, καλέι την ρουτίνα *internal_process_behaviour*, στην οποία παρέχει και μία υπορουτίνα της, *create_internals*.

Η *create_internals*, δημιουργεί τα pipes, τα προσθέτει στο pipe board, δημιουργεί τα παιδιά με $fork()$ κλήσεις, κλείνει τα pipes που δεν θα χρειαστεί το παιδί (sibling pipes) και, με *execv* κλήση στέλνει τα παιδιά να εκτελέσουν την internal behaviour. Η root process, θα εκτυπώσει specifics της ρίζας και θα μπει στην $internal\_read\_from\_child$, για να διαβάσει τους πρώτους και να εκτυπώσει ότι χρειάζεται.

Internal Code Base

Οι internals ακολουθούν, όπως είναι φυσικό, την ίδια συμπεριφορά με την ρίζα, εκτός από τις ακόλουθες λεπτομέρειες.

Εφόσον δεν θα δεχτούν κανένα SIGUSR1 signal, δεν εγκαθιδρύουν κανένα signal handler και η κλήση *execv* για τα child-processes, προορίζονται για το worker code-base. Παράλληλα περνούν και το κατάλληλο algorithm index (έχουμε $prime\{1, 2, 3\}$) στα παιδιά για να ξέρουν ποιόν να επιλέξουν και δίνουν και το αντίστοιχο $k \in \{0, 1, ...number\_of\_children\}$ στον W_k, για το worker profiling (δεν το αντιστοιχο section).

Workers Code Base

Οι workers, στο δικό τους code base, στο αρχείο *workers.c*, έχουν όλο τον χρήσιμο κώδικά τους στην main. Αρχικά, ελέγχουν το argv array, παίρνουν το root pid από το τελευταίο και τελούν την λειτουργία τους:
Φτιάχνουν ένα παιδί το οποίο θα εκτελέσει, με κλήση execl, το αντίστοιχο prime* αλγόριθμο για να βρεί τους πρώτους στο δοσμένο διάστημα. Ο worker, στο τέλος αφού κάνει wait για το παιδί του, εκτυπώνει το k του (process worker id) στέλνει το USR1 στην ρίζα με διαφορετικό τρόπο, αναλόγως αν έχει οριστεί η σταθερά PROTOCOL ή όχι.

Inter-Process Communication

Το message passing γίνεται με strings. Τα prime algorithms και τα internals στέλνουν αριθμούς και timestamps στην ρίζα με πολύ συγκεκριμένο τρόπο.

- fomat για πρώτους, χρόνο *: "$prime, timestamp$",
- format για πέρασμα elapsed time, worker numeric id: " : $timestamp$: $worker\_id$: "

Με αυτό τον τρόπο όταν το parent process διαβάζει από τα παιδιά του μπορεί να κάνει σωστό parsing (το οποίο εκτελείται από τις ρουτίνες *use_input* και *get_timestamp*) και ύστερα να κάνει batching output passing στο father process. Το batching γίνεται με την routine *send_batched_messages*, η οποία έχει ένα batch το μέγεθος του οποίου ορίζεται να είναι το $\max\{BATCHSIZE, \max_{prime,time \in input\_pairs}\{len(prime, time)\}\}$, όπου το $BATCHSIZE$, καθορίζεται από τον χρήστη στο αρχείο $Process.h$. Το batching γίνεται με το pop των ζευγαριών {prime, time}, την μετατροπή τους σε string στο προκαθορισμένο format και την εγγραφή τους στο batched buffer, ΑΝ υπάρχει χώρος και τέλος την εγγραφή του batched buffer στο stdout, όταν αυτό γεμίσει. Η διαδικασία συνεχίζεται μέχρι το priority queue να αδειάσει. Τα worker timestamps στέλνοται την στιγμή που διαβάζονται.

Signal Handling

Για την υλοποίηση του signal handling και του SIGUSR1 gathering από την ρίζα έχω ακολουθήσει 2 τρόπους:

- Χρήση μόνο της *sigaction* στην ρίζα και της *sigqueue* στα workers, για το USR1 gathering και sending αντιστοιχα, που δεν προσδίδει σημαντικό overhead στο όλο application,
- χρήση των παραπάνω 2 συναρτήσεων και την περαιτέρω χρήση μίας λούπας στην συναρτηση $wait\_signal\_from$ στα workers, όπου στέλνει αρκετά USR1 στην ρίζα μέχρι να τελειώσουν τα timeouts, ή μέχρι η ρίζα να στείλει πίσω ένα USR1 για να τερματίσει. Ταυτόχρονα η ρίζα έχει εγκαταστήσει ένα handler, οποίος φροντίζει να στέλνει USR1 στα παιδιά από τα οποία δέχεται signal και ταυτόχρονα με ένα array ελέγχει άν το παιδί έχει ξαναστείλει, αν όχι τότε ανεβάζει τον counter of received USR1 και ακολουθεί το πρωτόκολλο παραπάνω. Ο τρόπος αυτός θα πιάσει $70-100\%$ των USR1, ΑΛΛΑ θα προσδώσει ένα αρκετά σημαντικό overhead στο πρόγραμμα σε ορισμένα runs. Για την ενεργοποίηση του ορίστε την σταθερά PROTOCOL όπως φαίνεται στο section *Compilation and Run.*

Note: Η υλοποιησή μου ακολουθεί την λογική: *Μόνο οι workers στέλνουν SIGUSR1 στο root.*

Workers Profiling

Το workers profiling βασίζεται στο γεγονός ότι οι internals θα περάσουν όλους τα timestamps των workers. Βάσει αυτού υπάρχει ένα table στην $internal\_read\_from\_child$, όπου $children\_timestamps[worker\_num\_id] = time\_elapsed$. Με βάση αυτό το board, η ρίζα και μόνο αυτή εκτυπώνει τα κατάλληλα profiling stats, με την αντίστοιχη συνάρτηση. Η ανάθεση στο table γίνεται στην *use_input*.