Μοντέλα επικοινωνίας μεταξύ νημάτων και άγρυπνοι μπαρμπέρηδες: μια προγραμματιστική εξερεύνηση κλασικών προβλημάτων συγχρονισμού διεργασιών

Νεσλεχανίδης Οδυσσέας

ΑΜ: 1407

Επιβλέπων: Καθηγητής Αντωνιάδης Νικόλαος

20 Σεπτεμβρίου 2020

Περίληψη

Περιγράφεται συνοπτικά το θεωρητικό πεδίο της παράλληλης υπολογιστικής όπως ορίζεται στον άξονα της διαδιεργασιακής επικοινωνίας. Παρουσιάζονται κοινές αρχέγονες δομές συγχρονισμού (Semaphore, Spinlock, Barrier), και εφαρμογές τους στη λύση κλασικών προβλημάτων συγχρονισμού κοινόχρηστης μνήμης (Προβλήματα κοιμώμενου κουρέα, παραγωγού - καταναλωτή). Επιχειρείται η ταυτόχρονη διαχείριση των δύο προβλημάτων σε κοινή υλοποίηση σε κώδικα C++ με χρήση της βιβλιοθηκης pthread. Τέλος, ο κώδικας τεκμηριώνεται συνοπτικά, με αναφορά στις παραμέτρους που επηρέασαν τις αποφάσεις υλοποίησης.

Περιεχόμενα

I Θεωρία παράλληλου προγραμματισμού συστημάτων **2**

1 Εισαγωγή **3**

2 Παραλληλία ή ταυτοχρονισμός; **3**

3 Επικοινωνία μέσω κοινόχρηστης μνήμης (shared memory) **4**
3.1 Δομές συγχρονισμού νημάτων/διεργασιών 4
3.2 Συστήματα μη ομοιογενούς πρόσβασης μνήμης (NUMA) . . 5
3.3 Υβριδικά παράλληλα συστήματα 6

4 Επικοινωνία μεσω μηνυμάτων (message passing) **6**
4.1 Το πρότυπο MPI 6
4.2 Υπολογιστική γενικού σκοπού σε GPU (GPGPU) 7

II | sleepingbbq.cpp: Μια "ταυτόχρονη" επίλυση δύο κλασικών προβλημάτων συγχρονισμού διεργασιών **8**

5 Εισαγωγή **8**

6 Σύγκριση και θεωρητική οριοθέτηση **8**

7 Υλοποίηση **9**
7.1 Κουρά με ουρά 9
7.2 Ανταγωνισμός νημάτων παραγωγής πελατών 11
7.3 Το νήμα κουρέας 12
7.4 Τυχαιότητα και η συναρτηση randWait() 13
7.5 Λοιπά ζητήματα 13

Μέρος I

Θεωρία παράλληλου προγραμματισμού συστημάτων

1 Εισαγωγή

Τα προγραμματιστικά εργαλεία που μας επιτρέπουν να εκμεταλλευόμαστε τις δυνατότητες παράλληλης επεξεργασίας των σύγχρονων πολυεπεξεργαστών και συστοιχιών (clusters) επεξεργαστών σχεδιάζονται βάσει μιας ομάδας μοντέλων παράλληλου προγραμματισμού. Ιστορικά, αυτά τα μοντέλα ανταποκρίνονταν πιο άμεσα στη δομή του παράλληλου υλικού. Σήμερα, αποτελούν κυρίως βολικές αφαιρέσεις για την κατηγοριοποίηση των αλγορίθμων που χρησιμοποιούνται στον παράλληλο προγραμματισμό και για τους όρους σύνθεσης και χρήσης τους εντός των προγραμμάτων.[1][2]

Την ίδια στιγμή, η επιλογή του μοντέλου έχει θεμελιώδη ρόλο στις αποφάσεις που λαμβάνονται κατά το σχεδιασμό παράλληλων υπολογιστικών συστημάτων μεγάλης κλίμακας, στον κλάδο της υπολογιστικής υψηλών αποδόσεων (high performance computing).[3]

Τα μοντέλα παράλληλου προγραμματισμού χωρίζονται σε άξονες βάσει του τρόπου (εάν υφίσταται) που επικοινωνούν μεταξύ τους οι παράλληλες διεργασίες, και βάσει της φύσης των προβλημάτων που επιδιώκεται να λυθούν με παράλληλη επεξεργασία.[1]

Η παρούσα εργασία κινείται στον πρώτο άξονα, παρουσιάζοντας τα βασικά μοντέλα επικοινωνίας μεταξύ διεργασιών, και ειδικότερα μελετώντας τα κλασικά προγραμματιστικά εργαλεία για την επίλυση προβλημάτων συγχρονισμού που ενυπάρχουν στον προγραμματισμό κοινόχρηστης μνήμης, τις λεγόμενες αρχέγονες δομές συγχρονισμού (synchronization primitives).

2 Παραλληλία ή ταυτοχρονισμός;

Η παραλληλία (parallelism) και ο ταυτοχρονισμός (concurrency) είναι δύο διακριτές έννοιες συναφούς περιεχομένου που συχνά συγχέονται. Παρότι δεν υπάρχει πλήρης συμφωνία στο πώς ακριβώς διακρίνονται μεταξύ τους οι δυο όροι, για πολλούς συγγραφείς οι παραπάνω όροι επιδέχονται τους εξής ορισμούς:

- Στην ταυτόχρονη υπολογιστική, πολλές εργασίες του ίδιου προγράμματος μπορούν να είναι σε εξέλιξη οποιαδήποτε στιγμή.
- Στην παράλληλη υπολογιστική, πολλές εργασίες ενός προγράμματος συνεργάζονται στενά για την επίλυση ενός προβλήματος.[4]

Για μια εξήγηση σε πιο πρακτικούς όρους, κρίνεται σκόπιμο να εξεταστεί το πώς τα παραπάνω είδη υπολογιστικής σχετίζονται. Ο ταυτοχρονισμός εργασιών μπορεί να επιτευχθεί με διάφορους τρόπους. Ο ένας είναι η παράλληλη εκτέλεση των εργασιών με χρήση πολλαπλών κεντρικών μονάδων επεξεργασίας (CPU). Ένας δέυτερος είναι μέσω της εναλλαγής εργασιών (task switching), που στηρίζεται στην περιοδική, ή βάσει κανόνων, παύση εκτέλεσης εργασιών για το διαμοιρασμό του χρόνου (timesharing) της ίδιας επεξεργαστικής μονάδας μεταξύ πολλών σειριακών εργασιών που εκτελούνται σε αυτή. Εάν τα χρονικά μερίδια είναι αρκετά μικρά, η εκτέλεση των εργασιών δύναται για τους σκοπούς του χρήστη να ισοδυναμεί με αυτή που θα επιτυγχανόταν σε μια γνησίως παράλληλή υπολογιστική διάταξη.[5]

Ένας ακόμη όρος που ταιριάζει να αναφερθεί σε αυτό το πλαίσιο είναι αυτός της κατανεμημένης υπολογιστικής. Ο όρος αυτός περιγράφει συστήματα όπου ίσως υπάρχει αναγκαιότητα συνεργασίας μεταξύ διαφορετικών προγραμμάτων για την επίλυση ενός προβλήματος. Στην πηγή αναφέρονται τα εξής:

«Δεν υπάρχει κάποια ξεκάθαρη διάκριση μεταξύ παράλληλων και κατανεμημένων προγραμμάτων, αλλά ένα παράλληλο πρόγραμμα συνήθως εκτελεί πολλές εργασίες ταυτόχρονα σε πυρήνες που βρίσκονται κοντά ο ένας στον άλλο και οι οποίοι είτε μοιράζονται την ίδια μνήμη είτε συνδέονται μεταξύ τους μέσω ενός δικτύου πολύ υψηλής ταχύτητας. Από την άλλη, τα κατανεμημένα προγράμματα συνήθως είναι πιο "χαλαρά συζευγμένα". Οι εργασίες μπορούν να εκτελούνται σε πολλούς υπολογιστές που βρίσκονται σε μεγάλες αποστάσεις ο ένας από τον άλλον, και οι ίδιες οι εργασίες συχνά διεκπεραιώνονται από προγράμματα που αναπτύχθηκαν ανεξάρτητα.»[4]

3 Επικοινωνία μέσω κοινόχρηστης μνήμης (shared memory)

Στην παράλληλη ή ταυτόχρονη υπολογιστική, ένας πολύ φυσικός και προγραμματιστικά βολικός τρόπος για την επίτευξη επικοινωνίας μεταξύ εργασιών, καθώς και για την αποφυγή σπατάλης χώρου από πολλαπλά όμοια αντίγραφα δεδομένων, είναι η χρήση ενός κοινού χώρου μνήμης από πολλές εργασίες. Κατά τον προγραμματισμό βάσει αυτού του μοντέλου, προκύπτει μια σειρά προβλημάτων συγχρονισμού των εργασιών, για τη διαχείριση των οποίων έχουν αναπτυχθεί, και συνεχίζουν να αναπτύσσονται, πολλαπλές παραλλαγές δομών που συνολικά ονομάζονται "αρχέγονες δομές συγχρονισμού" (synchronisation schemes/primitives).

3.1 Δομές συγχρονισμού νημάτων/διεργασιών

Barrier

Είναι μια εύκολα υλοποιήσιμη δομή που αναγκάζει τα διερχόμενα νήματα σε αναμονή, μέχρι τα υπόλοιπα νήματα να φτάσουν στο ίδιο σημείο εκτέλεσης. Αυτό πραγματοποιείται με διάφορους τρόπους, ένας εκ των οποίων είναι η καταμέτρηση

των νημάτων που έχουν φτάσει στη δομή, με κατ'επανάληψιν έλεγχο για το αν έχει συμπληρωθεί ο αριθμός των νημάτων υπό εκτέλεση.

Spinlock

Υλοποιείται επίσης εύκολα, κρατάει τα νήματα σε αναμονή μέχρι να διέλθει το νήμα προτεραιότητας, που έχει την ευθύνη να ξεκλειδώσει τη σημαία (flag) που θα επιτρέψει σε κάποιο επόμενο νήμα να τη διεκδικήσει αποκλειστικά (για το ζήτημα της προτεραιότητας μεταξύ νημάτων σε αναμονή, δείτε το κεφάλαιο 7.1). Η αδυναμία της εν λόγω δομής έγκειται στο ότι για την αναμονή του νήματος δεσμεύεται ένας πυρήνας επεξεργασίας, καθώς αυτή επιτυγχάνεται μέσω επαναλαμβανόμενης εκτέλεσης του ελέγχου κατάστασης της σημαίας.

Σημαφόρος (Semaphore, Mutex)

Ο σημαφόρος, συγκεκριμένα στη δυαδική του παραλλαγή (mutex) ομοιάζει λειτουργικά του spinlock. Η υλοποίηση του είναι πιο σύνθετη, καθώς αξιοποιεί τις δυνατότητες του λειτουργικού συστήματος προκειμένου να αποδεσμεύει προσωρινά τους πυρήνες επεξεργασίας από νήματα που βρίσκονται σε αναμονή.

Η απρόσεκτη χρήση των παραπάνω δομών για την επίτευξη συγχρονισμού μπορεί εκ νέου να δημιουργήσει προβλήματα, η αποφυγή των οποίων επιδεικνύεται στην πράξη στην εφαρμογή της επόμενης ενότητας. Ονομαστικά, τα πιο συνήθη κρίσιμα από αυτά είναι το νεκρό ή ζωντανό κλείδωμα (Deadlock, Livelock) και η λιμοκτονία. Μικρά ή μεγάλα προβλήματα επιδόσεων του κώδικα είναι, ακόμη, πολύ εύκολο να προκύψουν.[6]

3.2 Συστήματα μη ομοιογενούς πρόσβασης μνήμης (NUMA)

Στο παρελθόν, ήταν διαδεδομένα τα υπολογιστικά συστήματα πολυεπεξεργαστών που διέθεταν αληθινά κοινόχρηστη μνήμη, με την οποία επικοινωνούσαν μέσω κοινού διαύλου. Σήμερα, η πραγματικότητα είναι διαφορετική.[2]

Με την αύξηση του πλήθους και της ταχύτητας των κεντρικών μονάδων επεξεργασίας (CPU) των πολυεπεξεργαστών, η ανάγκη διαχείρισης του προβλήματος συμφόρησης (bottleneck) von Neumann επέβαλε αλλαγή στο μοντέλο επικοινωνίας των κεντρικών μονάδων επεξεργασίας με τη μνήμη. Οι σημερινοί πολυεπεξεργαστές παρακάμπτουν σε ένα βαθμό το παραπάνω πρόβλημα έχοντας σχεδιαστεί σε πρότυπο μη-ομοιογενούς πρόσβασης μνήμης (Non-Uniform Memory Access: NUMA).[7]

Στις αρχιτεκτονικές αυτού του προτύπου, όπως μαρτυράει το όνομα του, χαρακτηριστική είναι η ανομοιογένεια στην ταχύτητα πρόσβασης (latency) ενός επεξεργαστή στα διάφορα στοιχεία μνήμης, ανάλογα με την απόσταση του στοιχείου

μνήμης από τον εν λόγω επεξεργαστή.[8] Οι καθιερωμένες διεπαφές προγραμματισμού εφαρμογών (APIs) κοινόχρηστης μνήμης (pthread, OpenMP) είτε δεν παρέχουν εύχρηστα μέσα για τη διαχείριση, μέσω κώδικα, της παραπάνω ιδιότητας των σύγχρονων πολυεπεξεργαστών[9], είτε οι λύσεις που παρέχουν, παρεχόμενες ως επεκτάσεις, καταργούν τη φορητότητα του κώδικα[10]. Η διαχείριση της μνήμης συνήθως αφήνεται, κατά συνέπεια, στο λειτουργικό σύστημα.

3.3 Υβριδικά παράλληλα συστήματα

Τα αναφερθέντα χαρακτηριστικά των εργαλείων προγραμματισμού που στηρίζονται στην αφαίρεση μιας εικονικά ενιαίας κοινόχρηστης μνήμης, συντελούν στο αποτέλεσμα η χρήση τους να περιορίζεται σε υπολογιστικά συστήματα μικρής κλίμακας. Διαδεδομένη, ωστόσο, είναι η χρήση τους σε συνδυασμό με εργαλεία προγραμματισμού κατανεμημένης μνήμης, για την υλοποίηση παράλληλων προγραμμάτων που εκτελούν επί μέρους εργασίες σε μικρά σύνολα επεξεργαστών, εντός υπολογιστικών συστημάτων μεγαλύτερης κλίμακας. Σε αυτού του είδους τις υβριδικές προσεγγίσεις, η επικοινωνία των επι μέρους έργων μεταξύ τους, αλλά και ευρύτερα με το σύστημα, υλοποιείται σε όρους προγραμματισμού κατανεμημένης μνήμης (message passing).[11]

4 Επικοινωνία μεσω μηνυμάτων (message passing)

Ο κλάδος του παραλληλισμού κατανεμημένης μνήμης είναι ευρύτατος και εσχάτως ραγδαία εξελισσόμενος. Καθώς δεν αποτελεί κεντρικό θέμα αυτής της εργασίας, θα καλυφθεί σύντομα και αμιγώς θεωρητικά.

Παραδοσιακά, με τον όρο "παραλληλισμός κατανεμημένης μνήμης" (να μη συγχέεται με τον όρο "κατανεμημένη υπολογιστική" που εισήχθη παραπάνω) περιγράφεται το μοντέλο όπου μια μονάδα επεξεργασίας εκτελεί ένα έργο σε έναν τοπικό χώρο μνήμης, το αποτέλεσμα του οποίου στη συνέχεια επικοινωνείται μέσω μηνυμάτων προς μια απομακρυσμένη μονάδα επεξεργασίας.[2] Αυτό προγραμματιστικά μεταφράζεται σε συναρτήσεις αποστολής (send) και λήψης (receive) μηνυμάτων, που συντονίζονται σε άμεση επικοινωνία βάσει κοινού πρωτοκόλλου (π.χ. master - slave)[12] ή, σε πιο σύνθετα και/ή ασύγχρονα συστήματα, με τη χρήση ενδιάμεσου υλικού ή λογισμικού διαχείρισης μηνυμάτων (message handlers, buffers).[13]

4.1 Το πρότυπο MPI

Στο προγραμματιστικό μοντέλο επικοινωνίας μέσω μηνυμάτων, ένα πρόγραμμα που εκτελείται σε ένα ζεύγος κεντρικής μονάδας επεξεργασίας (CPU) και μνήμης ονομάζεται επεξεργαστής (processor). Στην προγραμματιστική κοινότητα έχει

καθιερωθεί αυτός ο όρος και για τις διεργασίες (processes) που ορίζονται από το πρότυπο της Διεπαφής Μεταβίβασης Μηνυμάτων (Message Passing Interface: MPI).[14]

Ο όρος διεργασία υιοθετήθηκε σε αυτό το πρότυπο για να υποδηλώσει ότι, σε υλοποιήσεις στηριγμένες σε αυτό, η αντιστοίχιση μιας μονάδας επεξεργασίας ανά μονάδα μνήμης παύει να είναι απόλυτη. Αυτό είναι ένα θεμελιώδες χαρακτηριστικό της MPI, που διευκολύνει τη φορητότητα του κώδικα ανάμεσα σε διαφορετικές πλατφόρμες (λειτουργικά συστήματα) και διαφορετικής κλίμακας υπολογιστικά συστήματα.[15, 16, 17] Δημιουργεί, εν τούτοις, μια σύγχυση με τις διεργασίες (processes) των λειτουργικών συστημάτων. Γι'αυτό έχει εμφανιστεί και ο όρος proclets, για πιο ειδική αναφορά στις διεργασίες του προτύπου MPI. [18]

Το πρότυπο MPI υπήρξε για περισσότερο από μια δεκαετία, και παραμένει, το κυρίαρχο πρότυπο μεταβίβασης μηνυμάτων των προγραμμάτων που χρησιμοποιούνται στη βιομηχανία της υπολογιστικής υψηλών αποδόσεων.[17] Σήμερα, ωστόσο, η εμφάνιση νέων εργαλείων κατανεμημένης υπολογιστικής υψηλών αποδόσεων, όπως τα σχετικά εύκολα στην εγκατάσταση (deployment) frameworks Apache Hadoop και Spark, και γλωσσών κατασκευασμένων εξ αρχής για παράλληλη επεξεργασία κατανεμημένης μνήμης, όπως οι Charm++, Chapel, Julia, Erlang / Elixir, έχουν αρχίσει να περιορίζουν το εύρος εφαρμογών όπου το πρότυπο MPI, στις διάφορες υλοποιήσεις του, διατηρεί την πρωτοκαθεδρία.[19, 20, 21]

4.2 Υπολογιστική γενικού σκοπού σε GPU (GPGPU)

Οι μονάδες επεξεργασίας γραφικών (GPU) κατασκευάζονται με γνώμονα την επίλυση προβλημάτων γραφικής απεκόνισης και επεξεργασίας εικόνων. Η φύση αυτών των προβλημάτων έχει προσανατολίσει το σχεδιασμό των GPUs στην επίτευξη υψηλών αποδόσεων συνολικά σε υπολογισμούς που επωφελούνται από έντονο παραλληλισμό δεδομένων.[22]

Την ίδια στιγμή, οι GPUs έχουν πολύ περιορισμένες υπολογιστικές λειτουργίες και δυνατότητες προγραμματισμού, και η χρήση τους ενδείκνυται αποκλειστικά για επίλυση προβλημάτων που επωφελούνται από την επεξεργασία ροών (stream processing).[23]

Η τεχνολογία διαύλων υπολογιστικής γενικού σκοπού σε GPU (GPGPU pipelines) που αναπτύχθηκε ως λογισμικό με σκοπό τη βελτίωση των γραφικών σκίασης (shaders) στις αρχές της προηγούμενης δεκαετίας[24], σύντομα αξιοποιήθηκε και για πολύ διαφορετικούς σκοπούς, σε υπολογισμούς που επωφελούνται από υψηλές επιδόσεις στην παραλληλία δεδομένων (επιστημονική υπολογιστική, βιοπληροφορική, εξόρυξη κρυπτονομισμάτων).[25]
Τα προγραμματιστικά μοντέλα που αναπτύχθηκαν για τον προγραμματισμό συστημάτων GPGPU στηρίχθηκαν στην θεώρηση τους ως συστήματα κατανεμημένης μνήμης. Αυτή η αφαίρεση είχε μεγαλύτερη ανταπόκριση στην πραγματικότητα

του τότε υλικού, όπου η επικοινωνία μεταξύ επεξεργαστή και κάρτας γραφικών γινόταν μέσω διαύλου PCI, που μεταφραζόταν σε μεγάλη καθυστέρηση (latency) στην επικοινωνία μεταξύ των δύο επεξεργαστικών μονάδων (CPU και GPU) με τις επιμέρους μονάδες μνήμης τους. Σήμερα, υπάρχουν προτάσεις για ανάπτυξη προτύπων προγραμματισμού GPGPU που να προσομοιάζουν περισσότερο το μοντέλο παραλληλισμού κοινόχρηστης μνήμης.[2]

Μέρος II

| sleepingbbq.cpp: Μια "ταυτόχρονη" επίλυση δύο κλασικών προβλημάτων συγχρονισμού διεργασιών

5 Εισαγωγή

Σε αυτή την ενότητα παρουσιάζεται η μελέτη μου πάνω στα κλασικά προβλήματα συγχρονισμού Sleeping Barber και Producer-Consumer. Θεώρησα οφέλιμο να δοκιμάσω να γράψω ένα πρόγραμμα σε γλώσσα C++ με pthreads με μια λύση για κάθε πρόβλημα, προκειμένου να θεμελιώσω καλύτερα την κατανόηση μου επί των θεμάτων. Στην πορεία, βρήκα μεγαλύτερο ενδιαφέρον στην προσπάθεια ταυτόχρονης λύσης και των δύο προβλημάτων σε ένα πρόγραμμα.[26]

Στο παρόν, θα προσπαθήσω να κάνω μια κριτική παρουσίαση των προβλημάτων και των λύσεων που επέλεξα να υλοποιήσω γι'αυτά, αποφεύγοντας να επαναλάβω όσα παρουσιάζονται επαρκώς στα σχετικά λήμματα της wikipedia. Θα γίνει, ωστόσο, εκτενής αναφορά σε αυτά, καθώς και σε άλλες ελεύθερες πηγές.

6 Σύγκριση και θεωρητική οριοθέτηση

Το πρόβλημα Παραγωγού-Καταναλωτή (Producer-Consumer)[27], ή αλλιώς Bounded Buffer problem, αφορά τη διαφορά στο ρυθμό εισαγωγής και αφαίρεσης δεδομένων (throughput) σε έναν περιορισμένο χώρο μνήμης, από τον παραγωγό και τον καταναλωτή αντίστοιχα. Είναι λοιπόν, στη βάση του, ένα πρόβλημα συντονισμού του έργου των δυο νημάτων, ούτως ώστε να μην υπάρχουν λάθος αιτήματα προς τη μνήμη.

Το πρόβλημα του Κοιμώμενου Κουρέα (Sleeping Barber)[28] αφορά την αποφυγή συνθηκών ανταγωνισμού (race conditions) μεταξύ του κουρέα, που αφαιρεί πελάτες από μια περιορισμένου μεγέθους αίθουσα αναμονής, και των πελατών που προσθέτουν, ας πούμε, τους εαυτούς τους, σε αυτήν. Στη βάση του αφορά, λοιπόν, την εξασφάλιση ασφάλειας νημάτωσης (thread safety)[29] σε εργασίες που απαιτούν

πρόσβαση στο χώρο μνήμης που αντιπροσωπεύει την αίθουσα αναμονής. Έτσι, αποφεύγεται ο κίνδυνος υλοποιούμενες λύσεις να οδηγούν σε νεκρό κλείδωμα (deadlock).

Από τις παραπάνω παραγράφους γίνεται ξεκάθαρο ότι τα δύο προβλήματα, παρότι φαινομενικά όμοια, αφορούν διαφορετικά ζητήματα συγχρονισμού νημάτων/διεργασιών. Εν τούτοις, στις περισσότερες περιπτώσεις, τα δύο σύνολα ζητημάτων τείνουν να χρήζουν ταυτόχρονης διαχείρισης. Γι'αυτό το λόγο, επέλεξα να μελετήσω τα προβλήματα γράφοντας ένα πρόγραμμα και για τα δύο.

7 Υλοποίηση

7.1 Κουρά με ουρά

Η απλούστερη λύση για το πρόβλημα του κοιμώμενου κουρέα, όπως παρουσιάζεται στη wikipedia, μπορεί να οδηγήσει σε λιμοκτονία (starvation) ενός νήματος πελάτη. Για αυτό ευθύνεται το γεγονός ότι, ανάλογα με τις προδιαγραφές των αρχέγονων δομών συγχρονισμού (synchronization primitives) που παρέχονται με την εκάστοτε βιβλιοθήκη, μπορεί η προτεραιότητα των νημάτων να εξασφαλίζεται με διαφορετικούς τρόπους ή να αφήνεται απροσδιόριστη [30].

Αυτό με οδήγησε στη χρήση C++ προκειμένου να εκμεταλλευτώ τη δομή STL Queue. Πρόκειται για μια templated δομή, που ήλπιζα να χρησιμοποιήσω εισάγοντας σε αυτήν κλειδωμένους σημαφόρους για κάθε νεοεισερχόμενο νήμα-πελάτη. Καθώς, όμως, ο τύπος δεδομένων sem_t, που αντιπροσωπεύει το σημαφόρο της βιβλιοθήκης <semaphore.h> , δεν επιδέχεται συμβατικό χειρισμό (στα πρότυπα π.χ. του τύπου int), δεν βρήκα κάποιον τρόπο να το πετύχω. Αναγκάστηκα, τελικά, να στηριχτώ σε μια, φαινομενικά, μη αποδοτική, ad hoc υλοποίηση ουράς σημαφόρων. Η έμπνευση γι'αυτήν προήλθε από το πολύτιμο The Little Book of Semaphores [30].

```
int main(int argc, char* argv[]){
  ...
  int numChairs = atoi(argv[1]);
  waitingRoomQueue = new sem_t[numChairs+1];
  sem_init(&seatsAvailable, 0, numChairs);
  ...
  for(int i = 0; i < numChairs + 1; i++){
    sem_init(&waitingRoomQueue[i], 0, 1);
  }

}
void *customer(void *numChairs){
  int* number_of_chairs = (int*)numChairs;
  int positionInQueue{*number_of_chairs - 1};
```

```
  while(1){
    customerMessage(id, arrive_at_barbershop);
    if(sem_trywait(&seatsAvailable)==-1){
      customerMessage(id, waiting_room_full);
      }

    else{
      customerMessage(id, enter_waiting_room);
      while(positionInQueue > 0){
      sem_wait(&waitingRoomQueue[positionInQueue]);
      sem_post(&waitingRoomQueue[positionInQueue+1]);
      positionInQueue--;
      }
    }
    sem_wait(&barberChairEmpty);
    ...
    sem_post(&waitingRoomQueue[1]);
    ...
  }
}
```

Παραπάνω, η sem_trywait δοκιμάζει να μειώσει το σημαφόρο seatsAvailable. Εάν επιτύχει, που σημαίνει ότι ο σημαφόρος ήταν αρχικά μεγαλύτερος του μηδέν, επιστρέφει 0 (επιτυχία) και η εκτέλεση του προγράμματος συνεχίζει απορρίπτοντας τη συνθήκη. Σημειώστε ότι, σε αυτήν την περίπτωση, ο σημαφόρος μεταβλήθηκε κατά τον έλεγχο, παρότι ο έλεγχος απέτυχε. Αυτό είναι σκόπιμο.

Εάν η sem_trywait συναντήσει το σημαφόρο seatsAvailable σε κατάσταση που δε δύναται να μειωθεί (0), η sem_trywait επιστρέφει -1 και τερματίζει, σε αντίθεση με την sem_wait, που στην ίδια περίπτωση θα έθετε το νήμα σε αναμονή.[31] Ακολούθως, εισέρχεται στη συνθήκη, τυπώνει μήνυμα αποχώρησης του πελάτη λόγω συμφόρησης της αίθουσας αναμονής, και φτάνει στο τέλος της while για να τρέξει από την αρχή, με νέο id πελάτη (δείτε το επόμενο block κώδικα).

Το waitingRoomQueue είναι ένας πίνακας σημαφόρων μεγέθους numChairs +1 που αρχικοποιούνται με 1. Ο σημαφόρος waitingRoomQueue[numChairs] αυξάνεται σε κάθε εκτέλεση του σεναρίου else κατά 1, χωρίς ποτέ να μειωθεί. Αυτή είναι μια αδυναμία που δεν έχω σκεφτεί πώς θα επιλύσω, δε φαίνεται να δημιουργεί ωστόσο πρόβλημα βραχυπρόθεσμα.

Η τιμή του σημαφόρου waitingRoomQueue[0] δεν μεταβάλλεται σε κανένα σημείο του προγράμματος. Αφέθηκε για λόγους διαισθητικής ομαλότητας. Η αναμονή του κορυφαίου πελάτη της ουράς, στην πραγματικότητα εξαρτάται από το σημαφόρο barberChairEmpty.

7.2 Ανταγωνισμός νημάτων παραγωγής πελατών

Στη wikipedia, η λύση του χαρακτηριστικού προβλήματος του κοιμώμενου κουρέα, παρουσιάζεται σε συνθήκη ανταγωνισμού μεταξύ κουρέα και πελάτη για την πρόσβαση στο χώρο μνήμης που αντιπροσωπεύει την αίθουσα αναμονής. Στη δική μου υλοποίηση ανέβαλα αυτό το ζήτημα, αναθέτοντας στον πελάτη τα καθήκοντα προσθαφαίρεσης του εαυτού του στην ουρά αναμονής. Προχώρησα σε αυτήν τη διασκευή με την προοπτική να κάνω τη ζωή μου ευκολότερη σε μελλοντική υλοποίηση όπου θα προσθέσω πολλαπλά νήματα κουρέων.

Το ίδιο ζήτημα ανταγωνισμού για την πρόσβαση εμφανίστηκε ξανά όταν προσέθεσα πολλά νήματα-πελάτες. Η υλοποίηση της wikipedia προσομοιώνει πολλαπλούς πελάτες στηριζόμενη σε έναν ατέρμονο βρόχο του ίδιου νήματος. Θεώρησα αυτού του είδους την προσομοίωση ελάχιστα ρεαλιστική και τεχνικά απλουστευτική, και γι'αυτό επέλεξα να πολλαπλασιάσω τα νήματα-πελάτες, που ωστόσο άφησα να λειτουργούν σε ατέρμονα βρόχο. Έτσι, η λύση της wikipedia για το συγχρονισμό κουρέα - πελατών, εμφανίζεται στη δική μου υλοποίηση μεταξύ κουρέα και "νημάτων παραγωγής πελατών". Η επιλογή αυτού του νέου ονόματος ίσως γίνει πιο κατανοητή εαν ληφθεί υπόψιν ο τρόπος που επέλεξα να χειριστώ την ονοματοδοσία κάθε πελάτη.

```
//guards critical section where customer() pulls
//a unique id from global counter new_id.
pthread_mutex_t id_lock;
int new_id{1};
...
void *customer(...){
  ...
  while(1){
    pthread_mutex_lock(&id_lock); //fetch unique id for
    id = new_id;                  //customer
    new_id++;
    pthread_mutex_unlock(&id_lock);
    ...
  }
  ...
}
```

Αξίζει να σημειωθεί ότι, στη δική μου προσέγγιση, προκύπτει ένα πρόβλημα από την ελάττωση του αριθμού ενεργών νημάτων παραγωγής πελατών, όποτε κάποιο από αυτά εισέρχεται στην αίθουσα αναμονής. Αυτή η αδυναμία γίνεται ολότελα φανερή εφόσον οριστεί μεγάλος αριθμός θέσεων στην αίθουσα αναμονής από το χρήστη (απαιτείται ως όρισμα κατά την εκτέλεση). Καθώς ο αριθμός νημάτων παραγωγής πελατών περιορίζεται στα 5, υπάρχει το ενδεχόμενο όλα τα νήματα να τεθούν σε αναμονή μέσα στην αίθουσα αναμονής. Σε μια τέτοια περίπτωση, δε θα υπάρχουν αναχωρήσεις νέων πελατών μέχρι να ελευθερωθεί το κορυφαίο

νήμα από τον κουρέα, για να ξαναμπεί στο βρόχο. Δεν μπόρεσα να σκεφτώ έναν ικανοποιητικό τρόπο να αντιμετωπίσω το πρόβλημα, προς επίτευξη μεγαλύτερου ρεαλισμού, χωρίς να ξεφύγω πολύ από τις τεχνικές προδιαγραφές του προβλήματος. Το άφησα, λοιπόν, ως έχει.

```
void *customer(void *numChairs){
  ...
  while(1){
    ...
    else{
      ...
      sem_wait(&barberChairEmpty);
      pthread_mutex_lock(&room_state_change);
        customerMessage(id, check_on_barber);
        sem_post(&waitingRoomQueue[1]);
        sem_post(&seatsAvailable);
        sem_post(&barberPillow);
      pthread_mutex_unlock(&room_state_change);

      sem_wait(&seatBelt);
      customerMessage(id, leave_successfully);
      sem_post(&barberChairEmpty);
    }
  }
}
```

Παραπάνω, βλέπουμε πώς κινείται το κορυφαίο στην ουρά νήμα, μόλις ελευθερωθεί από το σημαφόρο barberChairEmpty, που θα αυξήσει ο φρεσκοκουρεμένος πελάτης πριν φτάσει στο τέλος της while και επιστρέψει για επανάληψη με νέο id (δείτε παραπάνω). Η μόνη παρέμβαση του νήματος - κουρέα είναι για την απελευθέρωση του νήματος παραγωγής πελατών μόλις αυτό φτάσει στο sem_wait(&seatBelt).

7.3 Το νήμα κουρέας

Παραπάνω, συναντάμε για πρώτη φορά το σημαφόρο barberPillow. Σε αυτό το σημαφόρο στηρίζεται ο συντονισμός του νήματος - κουρέα. Η συνάρτηση barber είναι αρκετά απλή:

```
void *barber(void *unused){
  int isAsleep{-1};
  while(1){
    sem_getvalue(&barberPillow, &isAsleep);
    if(isAsleep == 0){
```

```
        std::cout<<"The barber is sleeping.\n";
      }
    sem_wait(&barberPillow);
    std::cout <<"The barber is cutting hair.\n";
    randWait(5);
    std::cout <<"The barber has finished cutting hair.\n";
    sem_post(&seatBelt);
    randWait(1);
    }
}
```

Πριν τερματίσει, ο κουρέας απελευθερώνει το νήμα παραγωγής πελατών που έχει περάσει και κλειδώσει την κορυφή της ουράς (σημαφόρος barberChairEmpty). Το νήμα αυτό ξεκλειδώνει με τη σειρά του το σημαφόρο barberChairEmpty, πριν επιστρέψει στην αρχή του βρόχου.

Προτού ο κουρέας προλάβει να επιστρέψει στο sem_wait(&barberPillow) για να κοιμηθεί, μπορεί ένα νήμα παραγωγής πελατών που βρισκόταν σε αναμονή ξεκλειδώματος του barberChairEmpty, να προλάβει να αυξήσει το σημαφόρο barberPillow. Σε αυτή την περίπτωση το νήμα κουρέας θα μειώσει απευθείας το σημαφόρο barberPillow, χωρίς να χρειαστεί να περιμένει (δηλαδή, δε θα κοιμηθεί). Η συνθήκη που βλέπουμε παραπάνω καθιστά πιθανότερο το μήνυμα ύπνου να μην εμφανίζεται όταν ένα νήμα παραγωγής πελατών έχει τρέξει το

customerMessage(id, check_on_barber);

Δεν εξασφαλίζει ωστόσο κάτι τέτοιο. Κρίνω ότι η χρήση σημαφόρου θα ήταν υπερβολή.

7.4 Τυχαιότητα και η συναρτηση randWait()

Για να πετύχω τυχαίους χρόνους κοντά σε μια ζητούμενη διάρκεια, για κάθε εργασία, (κούρεμα, προσέλευση στο κουρείο, ταχύτητα νυστάγματος), έγραψα τη συνάρτηση randWait():

```
void randWait(int approxSeconds){
std::this_thread::sleep_for(static_cast<std::chrono::milliseconds>
(approxSeconds *  (rand() %  1000) ) );
}
```

Ο σπόρος τυχαιότητας (seed) για την rand() απαιτείται ως όρισμα κατά την εκτέλεση του προγράμματος. Εν τούτοις, φαντάζει άχρηστος, δεδομένου ότι ο μη ντετερμινισμός στους χρονισμούς των παράλληλων νημάτων παραγωγής πελατών είναι αρκετός, ώστε κάθε διαδοχική εκτέλεση, ακόμα και με τον ίδιο σπόρο τυχαιότητας, να διαφέρει απο την προηγούμενη.

7.5 Λοιπά ζητήματα

Για τα μηνύματα των νημάτων παραγωγής πελατών, χρειάστηκε να δημιουργήσω μια νέα συναρτηση customerMessage(int id, messageType message) (το mes-

sageType είναι enum). Αυτό επειδή κάθε χρήση του operator« συνιστά ξεχωριστή κλήση της cout, γι'αυτό και αρκετά μηνύματα κόβονταν στην μέση λόγω cout που εκτελούνταν παράλληλα σε διαφορετικά νήματα. Η λύση[32] ήταν απλή και ανευρέθηκε με αναζήτηση στο διαδίκτυο.

Γενικά, τα απαιτούμενα ορίσματα, αυτά του αριθμού των θέσεων στην αίθουσα αναμονής και του σπόρου τυχαιότητας, δεν επαρκούν για να εκθέσουν εύκολα όλες τις διαφορετικές συνθήκες που μπορεί να επικρατούν στο μαγαζί.

Σε έναν υπολογιστή με τις προδιαγραφές του δικού μου, για παράδειγμα, δεν υπάρχει τρόπος, μέσω ορισμάτων, να αυξομειωθεί αισθητά η συχνότητα που ο κουρέας βρίσκει χρόνο να πάρει έναν υπνάκο. Κάτι τέτοιο μπορεί, ωστόσο, να επιτευχθεί εύκολα, αυξομειώνοντας το όρισμα της συνάρτησης randWait(), όταν καλείται μεταξύ της αναχώρησης ενός νέου πελάτη, και της άφιξης του στο κουρείο. Μικρομετατροπές για διευκόλυνση αφήνονται για αργότερα.

Στην παρούσα κατάσταση του κώδικά, αναμενόμενα πρέπει να είναι μικρά κυρίως στυλιστικά παραπτώματα όπως η έλλειψη συνοχής στην ονοματοδοσία των μεταβλητών (camelCase, under_score) και στην επιλογή μεταξύ mutexes και δυαδικών σημαφόρων.

Ολόκληρο το αρχείο cpp υπάρχει στη διεύθυνση:

https://github.com/odnes/pardist/blob/master/sleepingbbq.cpp

Αναφορές

[1] Wikipedia (en), Parallel programming model, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Parallel_programming_model

[2] Gabriel Southern (April 16 2016) Stack Overflow, Main difference between Shared memory and Distributed memory, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://stackoverflow.com/questions/36642382/main-difference-between-shared-memory-and-distributed-memory#answer-36659895

[3] Rubin L, A Beginner's Guide to High–Performance Computing, Oregon State University, Ανακτήθηκε στις 21 Μαϊου 2020 από:
www.shodor.org/media/content/petascale/materials/UPModules/beginnersGuideHPC/moduleDocument_pdf.pdf

[4] Pacheco P. (2011), Εισαγωγή στον Παράλληλο Προγραμματισμό, Εκδόσεις Κλειδάριθμος, 34

[5] Tanenbaum A. (2015), Modern Operating Systems, 4th Edition, 85, 86

[6] Wikipedia (en), Synchronization (computer science), Implementation of Synchronization, Ανακτήθηκε στις 28 Μαϊου 2020 από
https://en.wikipedia.org/wiki/Synchronization_(computer_science)#Implementation_of_Synchronizati

[7] Wikipedia (en), Von Neumann architecture, Mitigations, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Von_Neumann_architecture#Mitigations

[8] Wikipedia (en), Non-uniform memory access, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Non-uniform_memory_access

[9] Hristo Iliev (August 15 2012), Stack Overflow, OpenMP and NUMA relation?, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://stackoverflow.com/questions/11959906/openmp-and-numa-relation#answer-11975593

[10] man7.org, Linux Programmer's Manual, PTHREAD_SETAFFINITY_NP(3), Ανακτήθηκε στις 21 Μαϊου 2020 από:
http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html

[11] Mackay D (2016), Hybrid Parallelism: Parallel Distributed Memory and Shared Memory Computing, Intel, Development Topics & Technologies, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://software.intel.com/en-us/articles/hybrid-parallelism-parallel-distributed-memory-and-shared-memory-computing

[12] Wikipedia (en), Message passing interface, Point-to-point basics, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Message_Passing_Interface#Point-to-point_basics

[13] Wikipedia (en), Message passing, Synchronous versus asynchronous message passing, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Message_passing#Synchronous_versus_asynchronous_message_passing

[14] Wikipedia (en), Message Passing Interface, Functionality, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Message_Passing_Interface#Functionality

[15] netlib.org, The Goals of MPI , Ανακτήθηκε στις 21 Μαϊου 2020 από:
http://www.netlib.org/utk/papers/mpi-book/node3.html

[16] Bosilca G., Herault T., Rezmerita A., Dongarra J, On Scalability for MPI Runtime Systems , Ανακτήθηκε στις 21 Μαϊου 2020 από:
www.netlib.org/utk/people/JackDongarra/PAPERS/cluster.pdf

[17] Wikipedia (en), Message Passing interface, Overview, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Message_Passing_Interface#Overview

[18] Kamal H., Mirtaheri S., Wagner A. (2010), Scalability of communicators and groups in MPI, ACM Digital Library Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://dl.acm.org/citation.cfm?id=1851507

[19] Dursi J., HPC is dying and MPI is killing it, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it.html

[20] Wikipedia (en), Apache Hadoop, Prominent use cases, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Apache_Hadoop#Prominent_use_cases

[21] Wikipedia (en), Julia (programming language), Notable uses, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Julia_(programming_language)#Notable_uses

[22] Wikipedia (en), Graphics processing unit, Stream processing and general purpose GPUs (GPGPU), Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/Graphics_processing_unit#Stream_processing_and_general_purpose_GPUs_(GPGPU)

[23] Wikipedia (en), General-purpose computing on graphics processing units, Stream processing, Ανακτήθηκε στις 21 Μαϊου 2020 από:
https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units#Stream_processing

[24] Wikipedia (en), General-purpose computing on graphics processing units, History, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units#History

[25] Wikipedia (en), General-purpose computing on graphics processing units, Applications, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units#Applications

[26] Νεσλεχανίδης Ο. (odnes), Github, sleepingbbq.cpp, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://github.com/odnes/pardist/blob/master/sleepingbbq.cpp

[27] Wikipedia (en), Producer-consumer problem, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://en.wikipedia.org/wiki/Producer-consumer_problem

[28] Wikipedia (en), Sleeping barber problem, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://en.wikipedia.org/wiki/Sleeping_barber_problem

[29] Wikipedia (en), Thread safety, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://en.wikipedia.org/wiki/Thread_safety

[30] Downey A. (2016), The Little Book of Semaphores, Green Tea Press, 47-52

[31] die.net, sem_trywait(3) - Linux man page, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://linux.die.net/man/3/sem_trywait

[32] Stack Overflow, multiple threads writing to std::cout or std::cerr, Ανακτήθηκε στις 21 Μαΐου 2020 από:
https://stackoverflow.com/questions/15033827/multiple-threads-writing-to-stdcout-or-stdcerr