Υλοποίηση δρομολογητή με τον αλγόριθμο Ουρές Προτεραιότητας

Εργασία στα πλαίσια του μαθήματος Λειτουργικά Συστήματα

{
Κασσελάκης Γεώργιος
Κρητικός Απόστολος
Σκαλιστής Στέφανος
}

Θεσσαλονίκη 2004

Περιεχόμενα

Η έννοια της δρομολόγησης

Γιατί χρειαζόμαστε δρομολόγηση

Με την πάροδο των χρόνων οι απαιτήσεις των ανθρώπων σε υπολογιστικούς αλλά και γενικότερα σε πόρους γίνονταν όλο και μεγαλύτερες. Αυτή η ανάγκη οδήγησε στη δημιουργία ταχύτερων επεξεργαστών. Ωστόσο η δημιουργία αυτών καθώς και η εισαγωγή της έννοιας του πολυπρογραμματισμού είχε σαν αποτέλεσμα την εμφάνιση προβλημάτων που επιζητούσαν άμεση λύση. Προβλήματα όπως η βελτιστοποίηση της χρήσης της CPU, ο καθορισμός προτεραιοτήτων και η ελαχιστοποίηση του μέσου χρόνου απόκρισης και επιστροφής.

Είναι γεγονός ότι ο επεξεργαστής είναι ένας από τους ακριβότερους πόρους στα υπολογιστικά συστήματα. Έτσι, λοιπόν, με την εισαγωγή του πολυπρογραμματισμού, την εκτέλεση δηλαδή πολλών διεργασιών από τον επεξεργαστή ταυτόχρονα, έγινε δυνατή η βελτιστοποίηση της χρήσης της CPU.

Όταν για παράδειγμα έχουμε μία διεργασία που απαιτεί χρήση Ε/Ε (δηλ. αρκετά αργή) τότε ο επεξεργαστής δεν χρησιμοποιείται από τη διεργασία αυτή και συνεπώς είναι ελεύθερος να εκτελέσει μια άλλη, παράλληλα με την προηγούμενη. Με τον τρόπο αυτό επιτυγχάνουμε τη διεκπεραίωση της "πρώτης" διεργασίας (που δεν χρησιμοποιεί τον επεξεργαστή) καθώς και μιας ακόμη που σε άλλη περίπτωση δεν θα εκτελούνταν.

Ένα ακόμη πρόβλημα που αντιμετωπίζουμε είναι η ιεράρχηση των διεργασιών με βάση κάποια υποκειμενικά κριτήρια. Για παράδειγμα μια διεργασία του λειτουργικού συστήματος μπορεί να είναι πιο σημαντική από το αίτημα ενός χρήστη.

Τέλος η ελαχιστοποίηση του μέσου χρόνου απόκρισης και επιστροφής των διεργασιών είναι ένα ακόμη κρίσιμο θέμα. Είναι φανερό ότι θέλουμε να εκτελούνται οι διεργασίες ταυτόχρονα και παράλληλα να έχουν όσο το δυνατό μικρότερο χρόνο επιστροφής. Επίσης σημαντικό είναι να ελαχιστοποιηθεί ο χρόνος απόκρισης ώστε να αποφευχθεί η παρατεταμένη στέρηση (starvation).

Λύσεις για όλα τα παραπάνω ζητήματα προσφέρει η δρομολόγηση με τις διάφορες πολιτικές της (First Come First Serve, Shortest Job First. Round Robin).

Τι ευθύνες έχει η δρομολόγηση

Δρομολόγηση είναι η διαδικασία κατά την οποία αποφασίζεται ποια διεργασία θα καταλάβει συγκεκριμένο πόρο, για πόσο χρονικό διάστημα και παράλληλα έχει τη δυνατότητα να διακόπτει τις διεργασίες σε περίπτωση που αυτό κριθεί απαραίτητο.

Ο δρομολογητής είναι προφανώς υλοποίηση της έννοιας της δρομολόγησης. Είναι ένα μέρος του λειτουργικού συστήματος που έχει ως σκοπό τη σωστή διαχείριση των διεργασιών ώστε να μεγιστοποιηθεί η χρήση του πόρου που δρομολογεί.

Ποιο είδος δρομολόγησης θα εξεταστεί

Ο δρομολογητής επεξεργαστή που χρησιμοποιούμε υλοποιεί τον αλγόριθμο ουρών προτεραιοτήτων (Priority Queues), με πολιτική δρομολόγησης την Εξυπηρέτηση εκ Περιτροπής (Round Robin). Δηλαδή, δημιουργούνται 7 διαφορετικές ουρές, από τις οποίες η πρώτη περιλαμβάνει διεργασίες με προτεραιότητα 1 (τη μεγαλύτερη), η δεύτερη διεργασίες με προτεραιότητα 2 και ούτω καθεξής. Κατόπιν αρχίζουν να εκτελούνται οι διεργασίες ξεκινώντας από αυτές που βρίσκονται στην πρώτη ουρά. Σε περίπτωση που υπάρχουν περισσότερες από μία διεργασίες στην ουρά αυτή, εξυπηρετούνται με βάση τον αλγόριθμο Εξυπηρέτηση εκ Περιτροπής (Round Robin). Αφότου αδειάσει η ουρά των διεργασιών με προτεραιότητα 1, εκτελούνται αυτές που βρίσκονται στην ουρά 2 και αυτό συνεχίζεται για όλες τις ουρές μέχρις ότου διεκπεραιωθούν όλες οι διεργασίες. Σε περίπτωση που έρθει διεργασία με προτεραιότητα μεγαλύτερη από αυτή της τρέχουσας, τότε στο τέλος του χρόνου θα αρχίσει να εκτελείται η διεργασία με τη μεγαλύτερη προτεραιότητα.

Υλοποίηση του δαίμονα δρομολόγησης

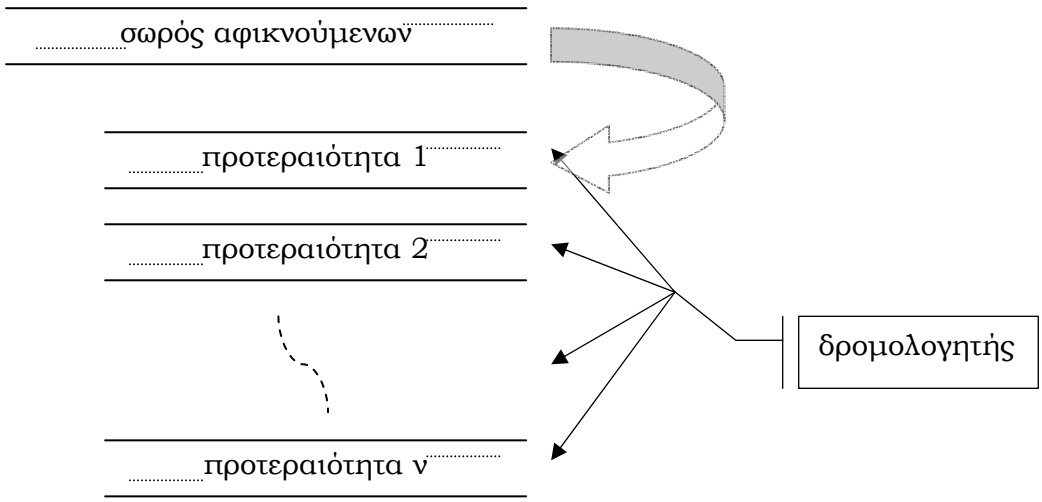
Αρχιτεκτονική του εγχειρήματος

Αρχιτεκτονικές επιλογές και αιτιολόγηση

Για να εκτελεί τις εργασίες του ο δρομολογητής χρησιμοποιεί δύο δομές αποθήκευσης .
Η πρώτη είναι ο σωρός αφικνούμενων διεργασιών . Σε αυτόν αποθηκεύονται οι διεργασίες που θα αφιχθούν στο μέλλον , και στην κορυφή βρίσκεται πάντα η διεργασία με τον μικρότερο χρόνο άφιξης .
Η δεύτερη είναι μια ιδιαίτερα ευέλικτη δομή η Simult όπου αποθηκεύονται οι διεργασίες των οποίων η εκτέλεση έχει ξεκινήσει . Χρησιμοποιείται αυτή η δομή ώστε να βελτιωθούν οι χρόνοι απόκρισης , για λόγους που θα αναλυθούν παρακάτω και θα δειχθούν πειραματικά στα δείγματα εκτέλεσης . Υπάρχουν 7 στο σύνολο τέτοιες δομές αποθήκευσης εργασιών , μία για κάθε επίπεδο προτεραιότητας .
Επελέγη αυτός ο τρόπος οργάνωσης των διεργασιών επειδή :

- είναι πιο πιστός στο πνεύμα της υπάρχουσας βιβλιογραφίας και δη στον τίτλο «ουρές προτεραιότητας» .
- κλιμακώνεται με απλό τρόπο για περισσότερα επίπεδα προτεραιότητας
- επεκτείνεται εύκολα για να υποστηρίξει μοντέλα παρατεταμένης στέρησης – ωρίμανσης [αρκεί να αλλάξει η δομή Simult με τον απόγονό της SHeap]
- αποτελεί λύση με μικρή πολυπλοκότητα [n ή ακόμη και log(n)] έναντι των εναλλακτικών λύσεων με κάποια ταξινόμηση επι της δομής στην οποία αποθηκεύονται [με ελάχιστο κόστος nlog(n)]

Σχηματική απόδοση



Στο σχήμα απεικονίζεται συνοπτικά η λειτουργία του δρομολογητή . Ο σωρός αφικνούμενων διεργασιών τροφοδοτεί τις ουρές προτεραιότητας . Όποτε ο χρόνος άφιξης της κορυφαίας διεργασίας ταυτίζεται με τον τρέχοντα χρόνο , η διεργασία τοποθετείται στην κορυφή της ουράς προτεραιότητας στην οποία ανήκει . Ο δρομολογητής σε κάθε χρονική στιγμή που αυτό απαιτείται , και σύμφωνα με τις διαθέσιμες εκτιμήσεις άφιξης εξετάζει πόσο χρόνο θα εκχωρήσει στην πρώτη διαθέσιμη – δηλαδή σε ανώτερο επίπεδο προτεραιότητας – διεργασία , και έπειτα αν δεν έχει ολοκληρωθεί την τοποθετεί στο τέλος της ουράς .

Οι εργασίες που εκτελεί ο δρομολογητής – Πηγαίος κώδικας

Θα παρουσιαστούν αναλυτικά και με σειρά εφαρμογής οι εργασίες που εκτελεί ο δρομολογητής . Μαζί με την τεκμηρίωση θα παρουσιάζεται και ο αντίστοιχος κώδικας της εργασίας που επιτελεί τον εκάστοτε σκοπό . Από τον κώδικα έχουν αλλαχθεί τα σχόλια, και έχουν αφαιρεθεί οι εκπομπές μηνυμάτων προς τον χρήστη , αφού εδώ στόχος είναι να δειχθεί ο τρόπος λειτουργίας και όχι η έξοδος των αποτελεσμάτων . Σε αρκετά σημεία έχουν γίνει επίσης απλουστεύσεις ώστε να μπορεί ο αναγνώστης να εστιάσει στην αρχιτεκτονική περισσότερο παρά στην υλοποίηση .

Βοηθητικές αποθηκευτικές δομές

Η δομή `process` **:**

```
struct process{
    long int arrival_time ;
    long int burst_time ;
    int priority ;
};
```

Χρησιμοποιείται για την προσωρινή αποθήκευση των στοιχείων που αντλούνται από το αρχείο proc.db . Περιέχει πεδία για τον χρόνο άφιξης [arrival_time], χρόνο καταιγισμού [burst_time] και προτεραιότητα [priority] .

Η κλάση pcb :

```
class pcb {
private:
    process ip;
    int remaining_time;
    int pid;
    int started;

public:
    pcb( int PID, process source ) {
        ip = source;
        remaining_time = ip.burst_time;
        pid = PID;
        started = -1;
    }
```

Φυλάσσεται ο χρόνος που απομένει για την ολοκλήρωση της διεργασίας ως remaining_time . Είναι ευθύνη του κάθε process control block να κρατά πληροφορίες σχετικά με τον χρόνο που απομένει μέχρι την ολοκλήρωση .

```
    pcb() {
    }
```

Η συνάρτηση που αναλαμβάνει να εκτελέσει την διεργασία , δέχεται σαν ορίσματα το ποσό χρόνου [quantity] που της εκχωρείται και την ώρα στην οποία βρισκόμαστε [current_time] . Επιστρέφει δε τον χρόνο που πραγματικά κατανάλωσε αν υπήρχε πλεόνασμα.

```
    int Dotime( int quantity, long int current_time ) {
```

Φυλάσσεται ο χρόνος στον οποίο ξεκίνησε η εκτέλεση της διεργασίας . Αυτό χρησιμεύει για την εξαγωγή στατιστικών.

```
        if ( started == -1 ) {
            started = current_time;
        }
```

Εάν ο απομείνας χρόνος είναι μικρότερος από τον χρόνο που απαιτείται για την ολοκλήρωση της διεργασίας τότε αυτή επιστρέφει ανακοινώσεις ολοκλήρωσης , και εκτυπώνει στατιστικά :
Ο χρόνος απόκρισης είναι started - ip.arrival_time
Ο χρόνος επιστροφής είναι current_time - started + quantity

```
        if ( remaining_time <= quantity ) {
            remaining_time = 0;
            return remaining_time;
        }
```

Αλλιώς , απλώς μειώνεται αντίστοιχα ο απομείνας χρόνος και επιστρέφει ο έλεγχος.

```
        remaining_time -= quantity;
```

```
        return remaining_time;
    }
```

Ακολουθούν συναρτήσεις πρόσβασης στα ιδιωτικά μέλη της κλάσης που ελλείψει ενδιαφέροντος δεν θα σχολιαστούν .

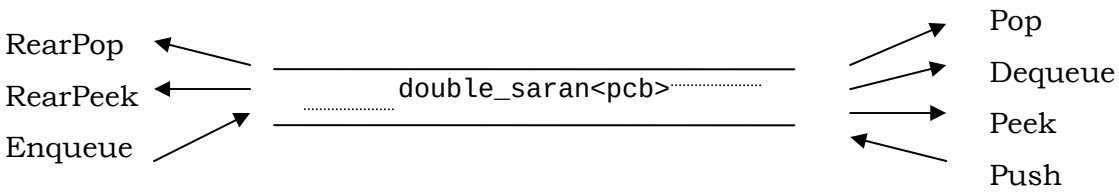
Η κλάση double_saran

Η κλάση double_saran χρησιμοποιείται σαν τυπική κλάση εγκιβωτισμού , που περιέχει το πεδίο Value που φέρει την τιμή του στοιχείου , και τους δείκτες Head , Tail που δείχνουν σε άλλα αντικείμενα της κλάσης . Προορίζεται κυρίως για να χρησιμοποιείται σαν σπόνδυλος σε διπλά συνδεδεμένη λίστα . Επειδή η υλοποίηση είναι τετριμμένη δεν θα σχολιαστεί περισσότερο από ότι είναι ήδη στον πηγαίο κώδικα .

```
template < class data >
class double_saran {
private:
    double_saran * _tail;
    double_saran * _head;
    data _value;

public:
    // δημιουργός της κλάσης
    double_saran( double_saran * Tail, double_saran * Head, data
Value ) {
        _tail = Tail;
        _head = Head;
        _value = Value;
    }

    // αλλαγή κεφαλής
    void setHead( double_saran * Head ) {
        _head = Head;
    }

    // επιστρογή κεφαλής
    double_saran * getHead() {
        return _head;
    }

    // αλλαγή ουράς
    void setTail( double_saran * Tail ) {
        _tail = Tail;
    }

    // επιστροφή ουράς
    double_saran * getTail() {
        return _tail;
    }

    // επιστροφή φιλοξενούμενης τιμής
    data Value() {
        return _value;
    }
};
```

Η δομή δεδομένων Simult

Η Simult λειτουργεί σαν δομή που ενθυλακώνει υπηρεσίες στοίβας, ουράς και λίστας με βάση μια δισυνδετική λίστα . Η λειτουργία και τα μέλη της παρουσιάζονται συνοπτικά στο ακόλουθο σχήμα .



Η κλάση με βάση τον τύπο που περιγράφει το template μπορεί να φιλοξενήσει διάφορους τύπους .

```
template < class data >
class Simult {
protected:
    double_saran < data > * _head;
    double_saran < data > * _tail;
    long int _count;

public:
    // δημιουργός
    Simult() {
        _head = 0;
        _tail = 0;
        _count = 0;
    }

    // επιστρέφει το πλήθος των αποθηκευμένων στοιχείων
    long int Count() {
        return _count;
    }

    // απαντά στο εάν η δομή είναι κενή στοιχείων
    bool const IsEmpty() {
        return _count == 0;
    }
```

Επιστρέφει το κορυφαίο στοιχείο χωρίς να το αφαιρεί [υπηρεσία στοίβας]

```
    data Peek() {
        if ( IsEmpty() ) throw;

        return _head->Value();
    }
```

Ωθεί ένα στοιχείο στην κορυφή [υπηρεσία στοίβας]

```
    void Push( data newValue ) {
        if ( IsEmpty() ) {
            _head = new double_saran < data > ( _head, 0, newValue );
            _tail = _head;
            ++_count;
```

```
            return;
        }
        //else
        _head->setHead(
new double_saran < data > ( _head, 0, newValue )
           );
        _head = _head->getHead();
        ++_count;
    }
```

Τραβά ένα στοιχείο απο την αρχή [υπηρεσία ουράς]

```
    data Dequeue() {
        return this->Pop();
    }
```

Τραβά το κορυφαίο στοιχείο [υπηρεσία στοίβας]

```
    data Pop() {
        if ( IsEmpty() ) throw;

        data result = _head->Value();
        double_saran < data > * dmw = _head;
        _head = _head->getTail();
        --_count;
        delete dmw;
        return result;
    }
```

Τοποθετεί ένα στοιχείο στο τέλος της δομής [υπηρεσία ουράς]

```
    void Enqueue( pcb newValue ) {
        if ( IsEmpty() ) {
            _head = new double_saran < data > ( _head, 0, newValue );
            _tail = _head;
            ++_count;
            return;
        }
        //else
        _tail->setTail(  new  double_saran  <  data  >  (  0,  _tail,
newValue ) );
        _tail = _tail->getTail();
        ++_count;
    }
```

Τραβά ένα στοιχείο από το τέλος της δομής

```
    data RearPop() {
        if ( IsEmpty() ) throw;

        data result = _tail->Value();
        double_saran < data > * dmw = _tail;
        // dmw για dead man walking
        _tail = _tail->getHead();
        --_count;
        delete dmw;
        return result;
    }
```

Επιστρέφει το ουριαίο στοιχείο χωρίς να το αφαιρεί

```
    pcb RearPeek() {
        if ( IsEmpty() ) throw;

        return _tail->Value();
    }

    // επιστρέφει την κλάση περιτύλιγμα κεφαλής
    double_saran < data > * getHead() {
        return _head;
    }

};
```

Ανάγνωση των διεργασιών από το αρχείο

Την εργασία αυτή ενθυλακώνει η κλάση IOEngine . Με την δημιουργία του αντικειμένου επιχειρείται το άνοιγμα του αρχείου proc.db στο οποίο βρίσκονται αποθηκευμένες οι διεργασίες με την μορφή:
χρόνος_άφιξης|χρόνος_καταιγισμού|προτεραιότητα

```
class IOEngine {
private:
    ifstream reader;
    ofstream writer;
    bool _eof_flag;

public:
```

Αρχικοποίηση των ρευμάτων

```
    IOEngine() {
        reader.open( "proc.db" );
        if ( reader.bad() ) {
            cerr << "Απέτυχε η ανάγνωση του proc.db";
            throw;
        }
        _eof_flag = 0;
        return;
    }
```

Αντλεί από το αρχείο την επόμενη εγγραφή με την μορφή του process

```
    process deSerializeProcess() {
        process result;
        char buffer[20];

        reader.getline( buffer, 21, '|' );
        result.arrival_time = atol( buffer );
        reader.getline( buffer, 21, '|' );
        result.burst_time = atol( buffer );
        reader.getline( buffer, 21, '|' );
        result.priority = atoi( buffer );

        if ( result.arrival_time == 0 &&
             result.burst_time == 0 &&
             result.priority == 0 )
           _eof_flag = 1;

        return result;
```

```
    }
```

Εξωτερικά αναγνώσιμη ερώτηση απαντά στο εάν η τελευταία ανάγνωση απέτυχε

```
    bool eof(){
        return _eof_flag ;
    }

};
```

Οι διεργασίες ανασύρονται με την μορφή `process` και ενκιβωτίζονται σε pcb με όνομα new_process . Κάθε φορά που ολοκληρώνεται επιτυχώς μια ανάγνωση [δηλαδή δεν έχουμε φθάσει στο τέλος του αρχείου] , αυτό τροφοδοτείται στο χρονοπρογραμματιστή που εδώ ονομάζεται harvester .

```
int main() {
    int processcounter = 0;
    IOEngine * lio = new IOEngine();
    Scheduler harvester( quantum );
    pcb * new_process;

    new_process = new pcb( ++processcounter,
                           lio->deSerializeProcess() );
    if ( lio->eof() ) exit( 0 );
    harvester.feedpcb( * new_process );
getnext:
    new_process = new pcb( ++processcounter,
                           lio->deSerializeProcess() );
```

Εάν η τελευταία άντληση ήταν επιτυχής τροφοδοτείται στο harvester

```
if ( !lio->eof() ) {
    harvester.feedpcb( * new_process );
    goto getnext;
}
```

Η διαδικασία του χρονοπρογραμματισμού

Τον χρονοπρογραμματισμό ενθυλακώνει στο σύνολό του η κλάση Scheduler . Κατά την δημιουργία της εκχωρεί τις δομές που περιγράψαμε στην παρουσίαση αρχιτεκτονικής .

```
SHeap < pcb > pending;
```

Οι ουρές προτεραιότητας

```
Simult < pcb > pq[priorities]; //ουρές προτεραιότητας
```

Βοηθητικές μεταβλητές

```
pcb current;
long int time;
int _quantum;
bool do_nop;

Scheduler( int quantum ) {
        _quantum = quantum;
        time = 0;
```

```
        do_nop = 1;
}
```

Ο σωρός ελάχιστων χρόνων άφιξης, σεχα στο εξής, προσφέρει πάντοτε στην κορυφή του τις διεργασίες που πιθανόν να έχουν αφιχθεί . Η παρακάτω συνάρτηση τον εξετάζει και εάν κάποιες διεργασίες έχουν αφιχθεί , τις τοποθετεί στην κατάλληλη ουρά προτεραιότητας .

```
void arrangepending() {
    while ( ( !pending.IsEmpty() )
              && pending.Peek().getArrival_time() =< time ) {
        int slot = pending.Peek().getPriority();
        pcb stub = pending.Pop();
        pq[slot].Push( stub );
    }
}
```

Η επιλογή να γίνει Push παρά Enqueue γίνεται ώστε να βελτιωθεί ο χρόνος απόκρισης , αφού οι νέες διεργασίες δεν θα χρειαστεί να περιμένουν τις παλαιότερες . Αυτό είναι ένα αντάλλαγμα που ευνοεί τον χρόνο απόκρισης σε βάρος του χρόνου επιστροφής . Η σύγκριση =< έχει επικουρικό περισσότερο σκοπό παρά λειτουργικό . Η παρακάτω συνάρτηση σαρώνει για τις διεργασίες που θα αφιχθούν ώστε να προτείνει ένα χρόνο εκτέλεσης που θα εξασφαλίσει ότι δεν πρόκειται να περιμένει μια διεργασία προτεραιότητας υψηλότερης ή ίσης να τελειώσει η εξεταζόμενη .

```
int lookforinterruptors() {
    int result = time + _quantum;
    double_saran < pcb > * cursor = pending.getHead();
    pcb cursorcontent = cursor->Value();
    int i = 0;
    while ( i < pending.Count() - 1 ) {
        if ( cursorcontent.getArrival_time() < result
          && cursorcontent.getPriority() <= current.getPriority() )
            result = cursorcontent.getArrival_time();
        cursor = cursor->getTail();
        cursorcontent = cursor->Value();
        ++i;
    }
    return result - time;
}
```

Αφού γίνει η διευθέτηση των διεργασιών , τρέχει η ακόλουθη συνάρτηση που αναζητά κατά προτεραιότητα για να βρει την πρώτη διαθέσιμη διεργασία . Όταν βρεθεί αυτή τοποθετείται στην μεταβλητή current .

```
void putnextpcb() {
    int i = 0;
    while ( i < priorities ) {
        if ( !pq[i].IsEmpty() ) {
            current = pq[i].Pop();
            do_nop = 0;
            return;
        }
        ++i;
    }
    //άν δεν υπάρχει κανένα τοτε εκπληρώνει ημίσια τερματική συνθήκη
    do_nop = 1;
```

```
}
```

Ακολουθεί η πραγματική συνάρτηση δρομολόγησης, που ρυθμίζει ουσιαστικά τον χρονοπρογραμματισμό των διεργασιών.

```
void execute() {
        goto skippushback;
again:
```

Αν δεν έχει ολοκληρωθεί η προηγούμενη τρέχουσα εργασία τότε τοποθετείται στο τέλος της αντίστοιχης ουράς προτεραιότητας.

```
        if ( !current.getRemaining_time() <= 0 )
             pq[current.getPriority()].Enqueue( current );
skippushback:
```

Αν ο σωρός αφικνούμενων περιέχει κάτι που μας ενδιαφέρει το τοποθετούμε στην αντίστοιχη ουρά προτεραιότητας .

```
        arrangepending();
```

Τοποθέτηση στο current της διεργασίας που θα εκτελεστεί

```
        putnextpcb();

  int allowed = 0;
  if ( pending.IsEmpty() ) allowed = 1000;
```

Αποφασίζεται ποιο είναι το ποσό χρόνου που επιτρέπεται να εκτελεστεί η διεργασία, που είναι min(χρονοτεμαχίου, lookforinterruptors)

```
  else
      allowed = lookforinterruptors();
  if ( !do_nop ) {
      if ( allowed < current.getRemaining_time() )
            dotime( allowed );
        else
             dotime( current.getRemaining_time() );
  }
  else {
```

Αν δεν υπάρχουν εργασίες να γίνουν τότε απλώς προχωρά ο χρόνος

```
  time += 1;
  }
  if ( pending.IsEmpty() && do_nop ) {
      return;
  }
```

Επανάληψη μέχρι να μην αναμένουν εργασίες πια , και επιπλέον να μην υπάρχει εργασία στις ουρές προτεραιότητας

```
  goto again;
  }
```

Οδηγίες χρήσης

Τρόπος χρήσης

Ο δρομολογητής επενεργεί σε ένα αρχείο εισόδου και εξάγει τα αποτελέσματα της εργασίας του στο ρεύμα κονσόλας standard error .
Οι δοκιμαστικές διεργασίες πρέπει να βρίσκονται στο αρχείο proc.db , το οποίο οφείλει να βρίσκεται στον ίδιο κατάλογο συστήματος με το εκτελέσιμο . Το αρχείο πρέπει να έχει την μορφή:
`χρόνος_άφιξης|χρόνος_καταιγισμού|προτεραιότητα|`
Το εκτελέσιμο OSMachine.exe καλείται και παράγει τα αποτελέσματά του , στην έξοδο σφάλματος . Εάν δεν έχει οριστεί ανακατεύθυνση τότε αυτά τυπώνονται πάλι στην κονσόλα . Η εργασία περιέχει και το εκτελέσιμο αρχείο δέσμης ενεργειών execute.bat . Το execute περιέχει την εντολή ανακατεύθυνσης του τυπικού σφάλματος στο αρχείο results.txt .

Εφικτές τροποποιήσεις κώδικα

Η αρθρωτή σχεδίαση της εργασίας επιτρέπει μερικές ενδιαφέρουσες τροποποιήσεις που το φέρνουν πιο κοντά στην πραγματικότητα .

Τροποποίηση του quantum

Αρκεί να αλλάξει η εντολή προεπεξεργαστή #define quantum 1 στο αρχείο OSProject.cpp σε κάποια άλλη θετική ακέραια τιμή . Οποιαδήποτε άλλη τιμή θα οδηγήσει σε ατέρμονα βρόχο που δεν θα εντοπιστεί στην μεταγλώττιση . Ένα τέτοιο δείγμα είναι και η Τρίτη δοκιμαστική εκτέλεση .

Τροποποίηση αριθμού προτεραιοτήτων

Για να υποστηρίξει περισσότερες ή λιγότερες ουρές προτεραιότητας , αλλάζουμε την εντολή προεπεξεργαστή #define priorities 7 στο αρχείο Scheduler.cpp .

Τροποποίηση της ενέργειας υποδοχής νέων διεργασιών

Όταν μια διεργασία αφικνείται πρέπει να τοποθετηθεί στην αντίστοιχη ουρά προτεραιότητας στην οποία ανήκει . Ο τρόπος όμως ίσως έχει σημαντικό αντίκτυπο στο χρόνο απόκρισης . Στο δείγμα εκτέλεσης #4 θα δούμε πως με αποδίδοντας ένα μικρό προνόμιο στην νέες διεργασίες , πετυχαίνουμε σημαντικές βελτιώσεις ταχύτητας . Για να γίνει αυτό αρκεί να αλλάξουμε την εντολή `#define arrival_action` από `Enqueue` σε `Push` .

Παράδειγμα εκτέλεσης

Δείγμα #1 μια δρομολόγηση χωρίς συγκρούσεις

Ακολουθεί το πηγαίο αρχείο για μια εργασία δρομολόγησης στην οποία δεν θα υπάρχουν συγκρούσεις προτεραιότητας [δηλαδή δεν γίνεται κατάμοιρασμός χρόνου σε διεργασίες ισοδύναμης προτεραιότητας] .

Οι διεργασίες στο εξής θα σημειώνονται με την μορφή
δ_x = [χρόνος άφιξης, χρόνος καταιγισμού, προτεραιότητα]
εδώ έχουμε τις διεργασίες
δ_1=[0,2,3], δ_2=[3,4,3], δ_3=[5,4,1]
και το αρχείο proc.db είναι :
0|2|3|3|4|3|5|4|1|

Η έξοδος του προγράμματος είναι :

```
1: Έκινά για πρώτη φορά η εκτέλεσή της διεργασίας
1: Εκτέλεση. Εναπομείνας χρόνος:2
1: Εκτελέστηκε για 1 απομένει 1
χρόνος : 1
1: Εκτέλεση. Εναπομείνας χρόνος:1
1: Εκτελέστηκε για 1 απομένει 0
1: Η διεργασία ολοκληρώθηκε
1: Χρόνος απόκρισης : 0
1: Χρόνος επιστροφής : 2
χρόνος : 2
2 αναμονή
2: Έκινά για πρώτη φορά η εκτέλεσή της διεργασίας
2: Εκτέλεση. Εναπομείνας χρόνος:4
2: Εκτελέστηκε για 1 απομένει 3
χρόνος : 4
2: Εκτέλεση. Εναπομείνας χρόνος:3
2: Εκτελέστηκε για 1 απομένει 2
χρόνος : 5
3: Έκινά για πρώτη φορά η εκτέλεσή της διεργασίας
3: Εκτέλεση. Εναπομείνας χρόνος:4
3: Εκτελέστηκε για 1 απομένει 3
χρόνος : 6
3: Εκτέλεση. Εναπομείνας χρόνος:3
3: Εκτελέστηκε για 1 απομένει 2
χρόνος : 7
3: Εκτέλεση. Εναπομείνας χρόνος:2
3: Εκτελέστηκε για 1 απομένει 1
χρόνος : 8
3: Εκτέλεση. Εναπομείνας χρόνος:1
3: Εκτελέστηκε για 1 απομένει 0
3: Η διεργασία ολοκληρώθηκε
3: Χρόνος απόκρισης : 0
3: Χρόνος επιστροφής : 4
χρόνος : 9
2: Εκτέλεση. Εναπομείνας χρόνος:2
2: Εκτελέστηκε για 1 απομένει 1
χρόνος : 10
2: Εκτέλεση. Εναπομείνας χρόνος:1
2: Εκτελέστηκε για 1 απομένει 0
2: Η διεργασία ολοκληρώθηκε
2: Χρόνος απόκρισης : 0
2: Χρόνος επιστροφής : 8
χρόνος : 11
11 αναμονή

Ο μέσος χρόνος απόκρισης σε δείγμα 3 ήταν 0
```

```
Ο μέσος χρόνος επιστροφής σε δείγμα 3 ήταν 4.66667

Οι διεργασίες καταναλώθηκαν . επιστροφή
```

Βλέπουμε ότι ορθώς :

- το τέλος της στιγμής 2 ακολούθησε μία στιγμή αναμονής
- στη αρχή της στιγμής 5 ξεκίνησε η εκτέλεση της διεργασίας δ_3 η οποία ανέστειλε την εκτέλεση της δ_2

Δείγμα #2 μια δρομολόγηση με συγκρούσεις

εδώ έχουμε τις διεργασίες
δ_1=[0,2,3], δ_2=[3,4,3], δ_3=[5,4,1], δ_4=[6,3,1]
και το αρχείο proc.db είναι :
0|2|3|3|4|3|5|4|1|6|3|1|

και τα αποτελέσματα:

```
1: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
1: Εκτέλεση. Εναπομείνας χρόνος:2
1: Εκτελέστηκε για 1 απομένει 1
χρόνος : 1
1: Εκτέλεση. Εναπομείνας χρόνος:1
1: Εκτελέστηκε για 1 απομένει 0
1: Η διεργασία ολοκληρώθηκε
1: Χρόνος απόκρισης : 0
1: Χρόνος επιστροφής : 2
χρόνος : 2
2 αναμονή
2: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
2: Εκτέλεση. Εναπομείνας χρόνος:4
2: Εκτελέστηκε για 1 απομένει 3
χρόνος : 4
2: Εκτέλεση. Εναπομείνας χρόνος:3
2: Εκτελέστηκε για 1 απομένει 2
χρόνος : 5
3: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
3: Εκτέλεση. Εναπομείνας χρόνος:4
3: Εκτελέστηκε για 1 απομένει 3
χρόνος : 6
4: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
4: Εκτέλεση. Εναπομείνας χρόνος:3
4: Εκτελέστηκε για 1 απομένει 2
χρόνος : 7
3: Εκτέλεση. Εναπομείνας χρόνος:3
3: Εκτελέστηκε για 1 απομένει 2
χρόνος : 8
4: Εκτέλεση. Εναπομείνας χρόνος:2
4: Εκτελέστηκε για 1 απομένει 1
χρόνος : 9
3: Εκτέλεση. Εναπομείνας χρόνος:2
3: Εκτελέστηκε για 1 απομένει 1
χρόνος : 10
4: Εκτέλεση. Εναπομείνας χρόνος:1
4: Εκτελέστηκε για 1 απομένει 0
```

```
4: Η διεργασία ολοκληρώθηκε
4: Χρόνος απόκρισης : 0
4: Χρόνος επιστροφής : 5
χρόνος : 11
3: Εκτέλεση. Εναπομείνας χρόνος:1
3: Εκτελέστηκε για 1 απομένει 0
3: Η διεργασία ολοκληρώθηκε
3: Χρόνος απόκρισης : 0
3: Χρόνος επιστροφής : 7
χρόνος : 12
2: Εκτέλεση. Εναπομείνας χρόνος:2
2: Εκτελέστηκε για 1 απομένει 1
χρόνος : 13
2: Εκτέλεση. Εναπομείνας χρόνος:1
2: Εκτελέστηκε για 1 απομένει 0
2: Η διεργασία ολοκληρώθηκε
2: Χρόνος απόκρισης : 0
2: Χρόνος επιστροφής : 11
χρόνος : 14
14 αναμονή

Ο μέσος χρόνος απόκρισης σε δείγμα 4 ήταν 0

Ο μέσος χρόνος επιστροφής σε δείγμα 4 ήταν 6.25

Οι διεργασίες καταναλώθηκαν . επιστροφή
```

Την στιγμή 6 ξεκινά η εναλλαγή εκτέλεσης μεταξύ των διεργασιών και η κάθε διεργασία παίρνει 1 χρόνο .

Δείγμα #3 μια δρομολόγηση με διαφορετικό quantum

Θα επαναλάβουμε τώρα την δρομολόγηση με διαφορετικό quantum ίσο με 4 .

Το αρχείο εισόδου παραμένει το ίδιο με αυτό του δείγματος 2 .

Το αποτέλεσμα είναι το εξής .

```
1: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
1: Εκτέλεση. Εναπομείνας χρόνος:2
1: Εκτελέστηκε για 2 απομένει 0
1: Η διεργασία ολοκληρώθηκε
1: Χρόνος απόκρισης : 0
1: Χρόνος επιστροφής : 2
χρόνος : 2
2 αναμονή
2: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
2: Εκτέλεση. Εναπομείνας χρόνος:4
2: Εκτελέστηκε για 2 απομένει 2
χρόνος : 5
3: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
3: Εκτέλεση. Εναπομείνας χρόνος:4
3: Εκτελέστηκε για 4 απομένει 0
3: Η διεργασία ολοκληρώθηκε
3: Χρόνος απόκρισης : 0
3: Χρόνος επιστροφής : 4
χρόνος : 9
4: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
```

```
4: Εκτέλεση. Εναπομείνας χρόνος:3
4: Εκτελέστηκε για 3 απομένει 0
4: Η διεργασία ολοκληρώθηκε
4: Χρόνος απόκρισης : 3
4: Χρόνος επιστροφής : 3
χρόνος : 12
2: Εκτέλεση. Εναπομείνας χρόνος:2
2: Εκτελέστηκε για 2 απομένει 0
2: Η διεργασία ολοκληρώθηκε
2: Χρόνος απόκρισης : 0
2: Χρόνος επιστροφής : 11
χρόνος : 14
14 αναμονή

Ο μέσος χρόνος απόκρισης σε δείγμα 4 ήταν 0.75

Ο μέσος χρόνος επιστροφής σε δείγμα 4 ήταν 5

0ι διεργασίες καταναλώθηκαν . επιστροφή
```

Ο χρόνος ολοκλήρωσης παραμένει ο ίδιος και έχουμε λιγότερες εναλλαγές περιεχομένου . Ο χρόνος απόκρισης αυξήθηκε γιατί η διεργασία δ_4 χρειάστηκε να περιμένει αφού δεν είχε μεγαλύτερη προτεραιότητα από την δ_3 .

Δείγμα #4 μια δρομολόγηση με αποκλειστική χρήση ουρών και quantum = 2

Στην δρομολόγηση #3 παρατηρήσαμε ότι με χρήση ενός μεγαλύτερου quantum αυξήθηκε ο χρόνος απόκρισης . Αυτή όμως η εκλογή είναι απαραίτητη όταν το κόστος εναλλαγής περιεχομένου ανταγωνίζεται τον χρονοτεμάχιο εκτέλεσης . Όταν αναλύαμε τον κώδικα στην συνάρτηση arrangepending() επισημάναμε την επιλογή της ώθησης των νέων διεργασιών παρά της τοποθέτησης στο τέλος της δομής .

Οι διεργασίες είναι οι
δ_1=[0,2,3], δ_2=[3,9,3], δ_3=[5,10,1], δ_4=[6,3,1]

Θα χρησιμοποιήσουμε για την επίδειξη το αρχείο
```
0|2|3|3|9|3|5|10|1|6|3|1|
```

Στην πρώτη εκτέλεση θα χρησιμοποιήσουμε το
```
#define arrival_action Enqueue
```

Τα αποτελέσματα είναι

```
1: Έεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
1: Εκτέλεση. Εναπομείνας χρόνος:2
1: Εκτελέστηκε για 2 απομένει 0
1: Η διεργασία ολοκληρώθηκε
1: Χρόνος απόκρισης : 0
1: Χρόνος επιστροφής : 2
χρόνος : 2
2 αναμονή
2: Έεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
2: Εκτέλεση. Εναπομείνας χρόνος:9
2: Εκτελέστηκε για 2 απομένει 7
χρόνος : 5
3: Έεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
```

```
3: Εκτέλεση. Εναπομείνας χρόνος:10
3: Εκτελέστηκε για 2 απομένει 8
χρόνος : 7
3: Εκτέλεση. Εναπομείνας χρόνος:8
3: Εκτελέστηκε για 2 απομένει 6
χρόνος : 9
4: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
4: Εκτέλεση. Εναπομείνας χρόνος:3
4: Εκτελέστηκε για 2 απομένει 1
χρόνος : 11
3: Εκτέλεση. Εναπομείνας χρόνος:6
3: Εκτελέστηκε για 2 απομένει 4
χρόνος : 13
4: Εκτέλεση. Εναπομείνας χρόνος:1
4: Εκτελέστηκε για 1 απομένει 0
4: Η διεργασία ολοκληρώθηκε
4: Χρόνος απόκρισης : 3
4: Χρόνος επιστροφής : 5
χρόνος : 14
3: Εκτέλεση. Εναπομείνας χρόνος:4
3: Εκτελέστηκε για 2 απομένει 2
χρόνος : 16
3: Εκτέλεση. Εναπομείνας χρόνος:2
3: Εκτελέστηκε για 2 απομένει 0
3: Η διεργασία ολοκληρώθηκε
3: Χρόνος απόκρισης : 0
3: Χρόνος επιστροφής : 13
χρόνος : 18
2: Εκτέλεση. Εναπομείνας χρόνος:7
2: Εκτελέστηκε για 2 απομένει 5
χρόνος : 20
2: Εκτέλεση. Εναπομείνας χρόνος:5
2: Εκτελέστηκε για 2 απομένει 3
χρόνος : 22
2: Εκτέλεση. Εναπομείνας χρόνος:3
2: Εκτελέστηκε για 2 απομένει 1
χρόνος : 24
2: Εκτέλεση. Εναπομείνας χρόνος:1
2: Εκτελέστηκε για 1 απομένει 0
2: Η διεργασία ολοκληρώθηκε
2: Χρόνος απόκρισης : 0
2: Χρόνος επιστροφής : 22
χρόνος : 25
25 αναμονή

Ο μέσος χρόνος απόκρισης σε δείγμα 4 ήταν 0.75

Ο μέσος χρόνος επιστροφής σε δείγμα 4 ήταν 10.5

Οι διεργασίες καταναλώθηκαν . επιστροφή
```

Εάν όμως επιλέξουμε το
#define arrival_action Push

τότε έχουμε αποτελέσματα

```
1: Η διεργασία ολοκληρώθηκε
1: Χρόνος απόκρισης : 0
1: Χρόνος επιστροφής : 2
χρόνος : 2
```

```
2 αναμονή
2: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
2: Εκτέλεση. Εναπομείνας χρόνος:9
2: Εκτελέστηκε για 2 απομένει 7
χρόνος : 5
3: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
3: Εκτέλεση. Εναπομείνας χρόνος:10
3: Εκτελέστηκε για 2 απομένει 8
χρόνος : 7
4: Ξεκινά για πρώτη φορά η εκτέλεσή της διεργασίας
4: Εκτέλεση. Εναπομείνας χρόνος:3
4: Εκτελέστηκε για 2 απομένει 1
χρόνος : 9
3: Εκτέλεση. Εναπομείνας χρόνος:8
3: Εκτελέστηκε για 2 απομένει 6
χρόνος : 11
4: Εκτέλεση. Εναπομείνας χρόνος:1
4: Εκτελέστηκε για 1 απομένει 0
4: Η διεργασία ολοκληρώθηκε
4: Χρόνος απόκρισης : 1
4: Χρόνος επιστροφής : 5
χρόνος : 12
3: Εκτέλεση. Εναπομείνας χρόνος:6
3: Εκτελέστηκε για 2 απομένει 4
χρόνος : 14
3: Εκτέλεση. Εναπομείνας χρόνος:4
3: Εκτελέστηκε για 2 απομένει 2
χρόνος : 16
3: Εκτέλεση. Εναπομείνας χρόνος:2
3: Εκτελέστηκε για 2 απομένει 0
3: Η διεργασία ολοκληρώθηκε
3: Χρόνος απόκρισης : 0
3: Χρόνος επιστροφής : 13
χρόνος : 18
2: Εκτέλεση. Εναπομείνας χρόνος:7
2: Εκτελέστηκε για 2 απομένει 5
χρόνος : 20
2: Εκτέλεση. Εναπομείνας χρόνος:5
2: Εκτελέστηκε για 2 απομένει 3
χρόνος : 22
2: Εκτέλεση. Εναπομείνας χρόνος:3
2: Εκτελέστηκε για 2 απομένει 1
χρόνος : 24
2: Εκτέλεση. Εναπομείνας χρόνος:1
2: Εκτελέστηκε για 1 απομένει 0
2: Η διεργασία ολοκληρώθηκε
2: Χρόνος απόκρισης : 0
2: Χρόνος επιστροφής : 22
χρόνος : 25
25 αναμονή

Ο μέσος χρόνος απόκρισης σε δείγμα 4 ήταν 0.25

Ο μέσος χρόνος επιστροφής σε δείγμα 4 ήταν 10.5

Οι διεργασίες καταναλώθηκαν . επιστροφή
```

Παρατηρούμε δηλαδή ένα υποτριπλασιασμό του χρόνου απόκρισης .