Υλοποίηση δρομολογητή με τον αλγόριθμο Ουρές Προτεραιότητας

Εργασία στα πλαίσια του μαθήματος Λειτουργικά Συστήματα
Μέρος Β

{
Κασσελάκης Γεώργιος
Κρητικός Απόστολος
Σκαλιστής Στέφανος
}

Θεσσαλονίκη 2004

Περιεχόμενα

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.

Το δεύτερο μέρος της εργασίας εισάγεται ο έλεγχος μονάδων εισόδου/εξόδου . Για την υλοποίηση επιλέξαμε να μην διαχωρίσουμε την έννοια της επεξεργασίας από τις μονάδες Ε/Ε , αλλα να ομαδοποιήσουμε τα πάντα υπο τον όρο πόροι . Για την συμμόρφωση ωστόσο στους περιορισμούς της εργασίας ορίσαμε και κάποιες σταθερές που αναφέρονται ειδικά σε ΚΜΕ, εκτυπωτή και δίσκο αποθήκευσης . Σε κάθε περίπτωση όποτε μια διεργασία εξυπηρετηθεί πλήρως από ένα πόρο , είτε τερματίζεται [kill] είτε δρομολογείται σε άλλο πόρο .

Υλοποίηση του δαίμονα δρομολόγησης

Αρχιτεκτονική του εγχειρήματος

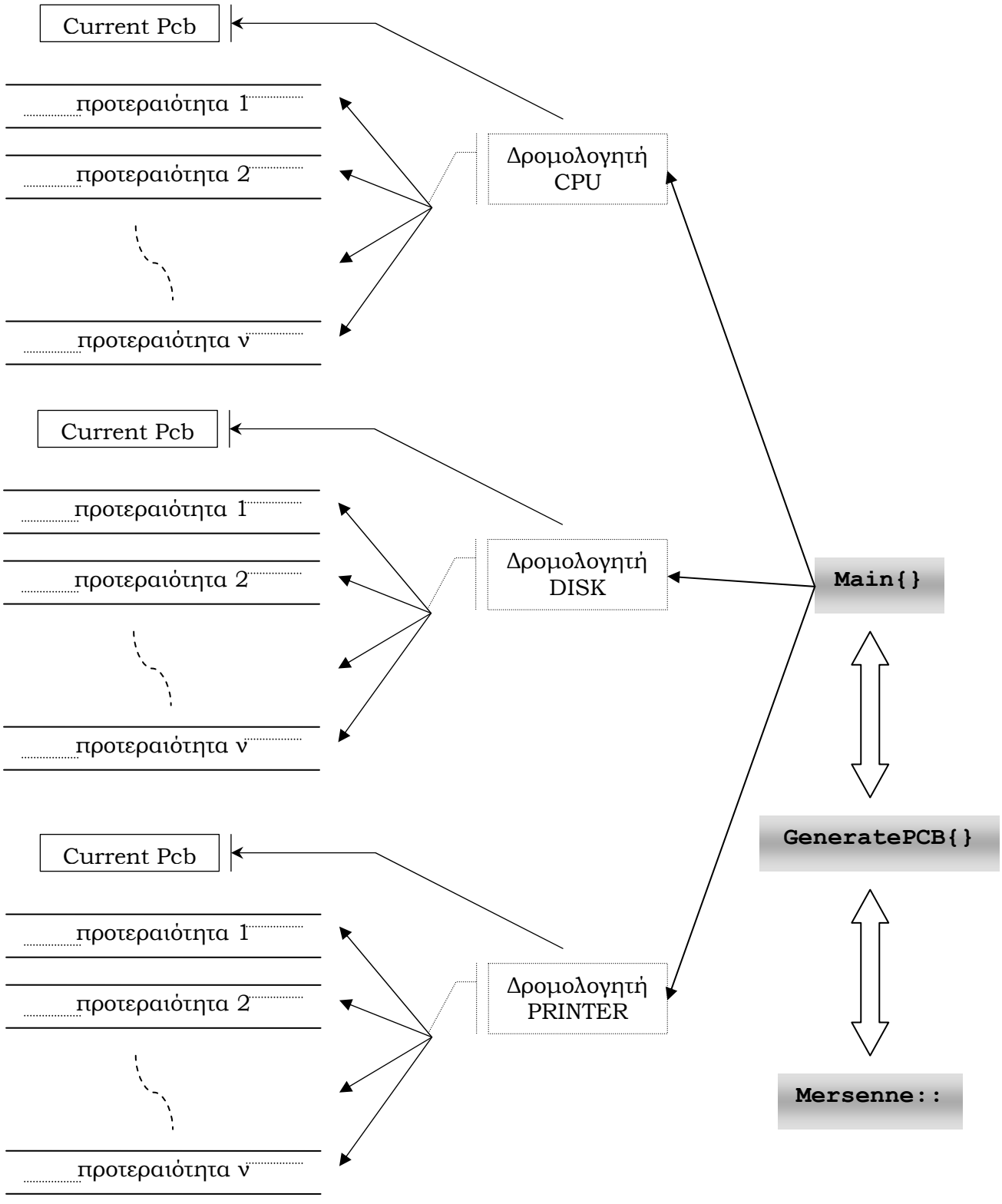
Αρχιτεκτονικές επιλογές και αιτιολόγηση

Για τις ανάγκες αποθήκευσης δεδομένων , σε όλο το εγχείρημα χρησιμοποιείται η 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 , κάτι που δείξαμε πειραματικά στο πρώτο μέρος της εργασίας . 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 συστήματος δρομολογητών . Η Mersenne παράγει αριθμούς για το GeneratePCB που με την σειρά του παράγει pcb για τους δρομολογητές . Όποτε μια διεργασία εκμεταλευτεί πλήρως τον πόρο στον οποίο ενδημούσε , αιτείται μετανάστευσης από την main() η οποία με τις βοηθητικές συναρτήσεις της αποφασίζει αν θα την φονεύσει [kill] ή θα της δώσει κάποιο άλλο πόρο . Ο κάθε 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 .

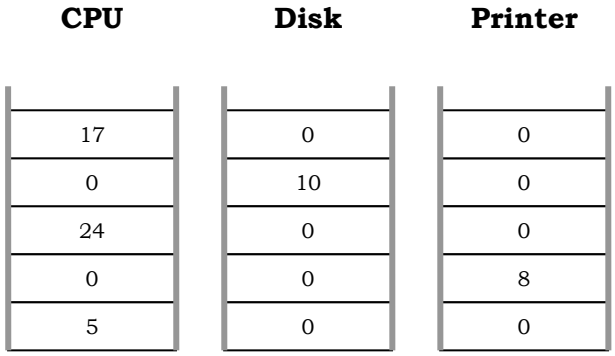
Η κλάση `pcb` :

```
#define N 3

class pcb
{
 private:

       Simult < int > _times[N];

      int time_created;
      int time_started;

      int done_time;
      int pid;
      int mode;
      int priority;
```

Σκοπός της κλάσης αυτής είναι να κρατά τις πληροφορίες process control block για κάθε διεργασία. Μέσα της ενθυλακώνεται ένας πίνακας (μεγέθους ίσου με το πλήθος των διαθέσιμων πόρων) από Simult που κρατούν τους χρόνους που χρειάζεται η διεργασία από τον κάθε πόρο ξεχωριστά με τη σωστή σειρά. Επίσης κρατά τον χρόνο που δημιουργήθηκε και τον χρόνο που άρχισε να εκτελείται [για στατιστικούς λόγους] . Ακόμη αποθηκεύει τον χρόνο που έχει απασχολήσει συγκεκριμένο πόρο, ποιο πόρο απασχολεί τη συγκεκριμένη στιγμή, το pid και την προτεραιότητα της διεργασίας αυτής.

Ένα σχηματικό παράδειγμα από στοίβες με χρόνους :

Είναι φανερό ότι σε κάθε γραμμή υπάρχει μόνο ένα μη μηδενικό στοιχείο. Με αυτό τρόπο δηλώνεται έπ' ακριβώς η σειρά με την οποία θα πρέπει να γίνει η επίσκεψη των πόρων.



Οι δύο βασικοί δημιουργοί της κλάσης:

Ιδιαίτερης σημασίας είναι ο δεύτερος ο οποίος καλείται από τον τυχαιοποιητή και έχει ως ορίσματα τρεις ακέραιους ένα για τον ταυτοποιητή της διεργασίας, ένα για την προτεραιότητα και ένα για το ποιο πόρο θα απασχολήσει μόλις ξεκινήσει. Ακόμη έχει ως όρισμα ένα δείκτη σε λίστα ακεραίων που ως σκοπό έχει να θέσει τις κατάλληλες τιμές στο πίνακα με τους χρόνους του pcb.

```
public:
  // δημιουργός
  pcb( int PID, int Priority, int Mode )
  {
    time_created = Time ;
    priority = Priority;
    mode = Mode;
    time_started = -1;
    pid = PID;
    done_time = 0;
  }

  //δημιουργός απο τυχαιοποιητή
  pcb( int PID, int Priority, int Mode, Simult < int > *Times)
  {
    int all = N - 1;
    while ( all >= 0 )
    {
      this->_times[all] = * ( Times + all );
      --all;
    }

       time_created = Time ;
       time_started = -1;
       pid = PID;
       done_time = 0;
       mode = Mode ;
       priority = Priority ;
```

```
  }
```

Η συνάρτηση που αναλαμβάνει να εκτελέσει τη διεργασία ελέγχει αν η διεργασία έχει ξεκινήσει ή όχι..

```
void Do_Time(){
      if ( time_started == -1 )
```

Αν όντως είναι η πρώτη φορά που εκτελείται θέτει ως χρόνο εκκίνησης τον υπάρχοντα χρόνο και ενημερώνει την υπηρεσία στατιστικών δεδομένων για το χρόνο απόκρισης.

```
    {
      time_started = Time;
      Log.logResponseTime( Time - time_created ) ;
    }
```

Αν δεν υπάρχει εργασία για να γίνει τότε το pcb παύει εαυτόν.

```
     if ( _times[mode].IsEmpty() || _times[mode].Peek() == 0 )
       {
             cout << "το pcb" << pid << " παύει εαυτόν" << endl ;
             Stop();
             return ;
       }
```

Αλλιώς μειώνεται ο χρόνος που απομένει.

```
    _times[mode].Push( _times[mode].Pop() - 1 );
    done_time++;
  }
```

Στη περίπτωση που μια διεργασία έχει τελειώσει με κάποιο πόρο και χρειάζεται κάποιον άλλο πρέπει να μεταναστεύσει.

```
bool Migrate(){
```

Έτσι, λοιπόν, γίνεται απόρριψη του άδειου τμήματος εργασίας.

```
      _times[mode].KillTopUnmanaged();
```

Έπειτα αναζητάτε η επόμενη εργασία αφαιρώντας τις τελειωμένες εργασίες μέχρις ότου βρεθεί η επόμενη διεργασία ή διαπιστωθεί ότι η διεργασία έχει ολοκληρωθεί.

```
  int cursor = mode + 1;
  while ( !this->Dead() )
      {
            if ( this->Finished( cursor % N ) )
                              _times[mode].KillTopUnmanaged();
            else {
                mode = cursor % N;
                return true;
            }
            ++cursor;
        }
```

```
        return false;
    }
```

Η συνάρτηση Finished έχει ως σκοπό τον έλεγχο αν μια διεργασία έχει τελειώσει με το πόρο που απασχολεί ή όχι.

```
    bool Finished( int Mode ) {
```

Αν είναι άδεια η στοίβα

```
        if ( _times[Mode].IsEmpty() ) return true;
```

Ή η κορυφαία τιμή είναι 0 επιστρέφει true (1) αλλιώς επιστρέφει false (0).

```
        return _times[Mode].Peek() == 0;
    }
```

Υπεύθυνη συνάρτηση για το αν μια διεργασία έχει ολοκληρωθεί είναι η Dead.

```
  bool Dead() {
```

Αν άδειασαν όλες οι στοίβες ενημερώνει την υπηρεσία στατιστικών δεδομένων για το χρόνο επιστροφής και επιστρέφει true (1).

```
        for ( int i = 0; i < N; i++ ) {
            if ( !_times[i].IsEmpty() ) return 0;
        }
        Log.logReturnTime( Time - time_started );
        return 1;
    }
```

Η συνάρτηση Start θέτει τη διεργασία στο πόρο που δέχεται ως όρισμα αφού πρώτα ελέγξει αν η διεργασία χρειάζεται καθόλου αυτό το πόρο.

```
  void Start( int type ) {
        if ( _times[type].IsEmpty() ) cout << endl << "Επιχειρείται
                          εκτέλεση pcb που έχει τερματίσει" << endl ;
        mode = type;
    }
```

Η συνάρτηση που είναι υπεύθυνη για τη διακοπή της εκτέλεσης μια διεργασίας είναι η Stop όπου απλώς τυπώνεται ο συνολικός χρόνος εκτέλεσης στον συγκεκριμένο πόρο.

```
void Stop() {
      done_time = 0;
      if ( _times[mode].Peek() == 0 )
    }
```

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

```
  int getPID(){
    return pid;
  }

  int getRemaining_time( int type ){
    return _times[type].Peek();
  }

  int getDone_time(){
    return done_time;
  }

  int getMode(){
    return mode;
  }

  int getPriority(){
    return priority;
  }
 void setMode( int newmode ) {
         mode = newmode;
     }

};
```

Η κλάση double_saran

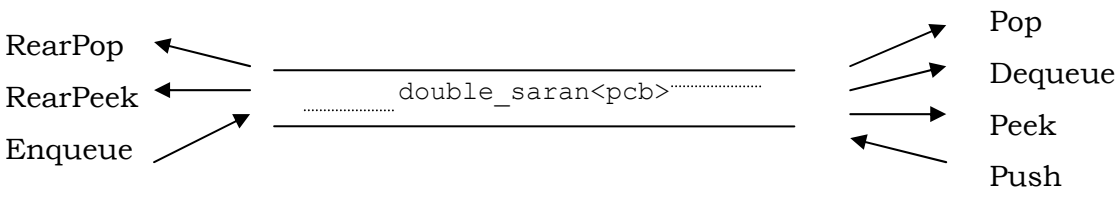
Η κλάση double_saran χρησιμοποιείται σαν τυπική κλάση εγκιβωτισμού , που περιέχει το πεδίο Value που φέρει την τιμή του στοιχείου , και τους δείκτες Head , Tail που δείχνουν σε άλλα αντικείμενα της κλάσης . Προορίζεται κυρίως για να χρησιμοποιείται σαν σπόνδυλος σε διπλά συνδεδεμένη λίστα . Επειδή η 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;
    }
```

```
    // επιστροφή 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;
    }

};
```

Γένεση τυχαίων process control blocks

Για την παραγωγή των εργασιών χρησιμοποιούνται δύο αρθρώματα : Η Κλάση Mersenne και η συνάρτηση GeneratePCB .
Η Κλάση Mersenne παρέχει υπηρεσίες δημιουργίας τυχαίων ακέραιων αριθμών, ομοιόμορφα κατανεμημένων σε ένα σαφώς καθορισμένο πεδίο . Η Κλάση χρησιμοποιεί τον αλγόριθμο : Στρόβιλος Mersenne . Ο αλγόριθμος διακρίνεται για την πολύ καλή ομοιόμορφη κατανομή και την εξαιρετική ταχύτητά του . Η μέθοδος έχει εισηγηθεί από τους &&&&&&&& και δημοσιεύτηκε αρχικά σε περιοδική έκδοση του ACM . Έχει εκδοθεί επίσημα υπό την άδεια General Public License . Παρατίθεται ο πηγαίος κώδικας . Ο λεπτομερής σχολιασμός του ξεφεύγει από τους σκοπούς της εργασίας .

```
typedef signed long int32;
typedef unsigned long uint32;

#define MERS_N   351
#define MERS_M   175
#define MERS_R   19
#define MERS_U   11
```

```
#define MERS_S   7
#define MERS_T   15
#define MERS_L   17
#define MERS_A   0xE4BD75F5
#define MERS_B   0x655E5280
#define MERS_C   0xFFD58000

class Mersenne {
private:
    uint32 mt[MERS_N];
```

Το διάνυσμα κατάστασης είναι το παράθυρο στην ακολουθία τυχαίων bits που παράγει ο αλγόριθμος

```
    int mti; // δείκτης στο mt

public:
    Mersenne( uint32 seed ) { // δημιουργός
        RandomInit( seed );
    }
```

Αρχικοποίηση με ένα σπόρο

```
    void RandomInit( uint32 seed ) {
        // νέος σπόρος
        mt[0] = seed;
```

Τροφοδότηση του αριθμού στο διάνυσμα κατάστασης

```
        for ( mti = 1; mti < MERS_N; mti++ ) {
            mt[mti]=(1812433253UL*(mt[mti-1]^(mt[mti-1]>>30))+mti);
        }
    }
```

Παραγωγή αριθμού περιορισμένου στο πεδίο [low,high]

```
    int IntegerRandom( int low, int high ) {
        // πειστροφή αριθμού στο πεδίο low <= x <= high

        union {
            double f; uint32 i[2];
        }
        convert;

        uint32 y; // Γένεση 32 τυχαίων bits
```

Αν δεν έχουμε ξεπεράσει το όριο επαναληπτικότητας

```
        if ( mti >= MERS_N ) {
```

Γένεση `MERS_N` λέξεων σε μια κίνηση

```
            const uint32 LOWER_MASK = ( 1LU << MERS_R ) - 1;
```

Ορισμός των κάτω `MERS_R bits`

```
            const uint32 UPPER_MASK = -1L << MERS_R; //
```

Ορισμός των άνω `(32 - MERS_R)` bits

```
            static const uint32 mag01[2] = {
                0, MERS_A
            };
```

Επανάληψη στην οποία

```
        int kk;
```

Ανάγνωση και κύλιση του παραθύρου δεξια στην ακολουθία [ή αριστερά ως προς τον καταχωρητή] . Εύρεση του `MERS_N - MERS_M` τμηματος .

```
        for ( kk = 0; kk < MERS_N - MERS_M; kk++ ) {
          y = ( mt[kk] & UPPER_MASK ) | ( mt[kk + 1] & LOWER_MASK );
               mt[kk] = mt[kk + MERS_M] ^ ( y >> 1 ) ^ mag01[y & 1];
           }
```

Εύρεση του 32 - `MERS_N + MERS_M` τμηματος .

```
            for ( ; kk < MERS_N - 1; kk++ ) {
           y = ( mt[kk] & UPPER_MASK ) | ( mt[kk + 1] & LOWER_MASK );
mt[kk] = mt[kk + ( MERS_M - MERS_N )] ^ ( y >> 1 ) ^ mag01[y & 1];
            }
```

Εφαρμογή των μασκών στον καταχωρητή

```
y = ( mt[MERS_N - 1] & UPPER_MASK ) | ( mt[0] & LOWER_MASK );
mt[MERS_N - 1] = mt[MERS_M - 1] ^ ( y >> 1 ) ^ mag01[y & 1];
          mti = 0;
      }

      y = mt[mti++];
```

Προαιρετική ανάδευση [δεν βελτιώνει εγγυημένα την τυχαιότητα]

```
      y ^= y >> MERS_U;
      y ^= ( y << MERS_S ) & MERS_B;
      y ^= ( y << MERS_T ) & MERS_C;
      y ^= y >> MERS_L;

      convert.i[0] = y << 20;
      convert.i[1] = ( y >> 12 ) | 0x3FF00000;
```

Πολλαπλασιασμός τυχαίου με το διάκενο και αποκοπή

```
      int r=int( ( high - low + 1 ) * ( convert.f - 1.0 ) ) + low;
      if ( r > high ) r = high;
      if ( high < low ) return 0x80000000;
      return r;
   }
};
```

Ακολουθεί η συνάρτηση GeneratePCB . Η συνάρτηση παράγει ένα νέο pcb σύμφωνα με τους περιορισμούς που θέτουμε . Είναι σημαντικό να παρατηρήσουμε στην δημιουργία των pcb είναι μόνο το seed του Mersenne και η πιθανότητα δημιουργίας . Η πιθανότητα εξασφαλίζει ότι δεν θα δημιουργείται σε κάθε κύκλο νέα διεργαία . Παρατίθεται ο κώδικας .

```
pcb * Generatepcb( int PID, float Chance, int Low_Count, int
High_Count, int Low_Duration, int High_Duration )
     {
         int Degree = RESOURCES ;
```

Δημιουργείται πίνακας από στοίβες [ο λόγος της ύπαρξής τους αναλύεται στην τεκμηρίωση του pcb .

```
       Simult < int > * Result = new Simult < int > [RESOURCES];
       int Mode = m.IntegerRandom( 0, RESOURCES - 1 ) ;
       if ( Chance * 100 > m.IntegerRandom( 0, 98 ) )
       {
         while ( Degree > 0 )
         {
```

Πλήρωση του πίνακα με τεμάχια εργασιών

```
           int elements = m.IntegerRandom( Low_Count, High_Count );
           while ( elements > 0 )
           {
             ( Result + Degree - 1 )->Push(
                  m.IntegerRandom( Low_Duration, High_Duration ) );
             --elements;
           }
           --Degree;
         }
```

Θεωρούμε ότι οι διεργασίες πρώτα περνούν από ένα πόρο , το ποιος είναι αυτός είναι τετριμμένο πρόβλημα

```
         return new pcb( PID, 1, Mode, Result );
       }
       else
         return NULL;
}
```

Η διαδικασία του χρονοπρογραμματισμού

Η κλάση scheduler δημιουργήθηκε για να υλοποιεί όλα τα διαφορετικά είδη δρομολογητών που μπορεί να χρειαστεί να προσομοιώσει το λειτουργικό μας σύστημα. Επιλέξαμε σε αυτή τη φάση να χρησιμοποιούμε 3 διαφορετικά είδη δρομολογητή (CPU, DISK, PRN), τα οποία προσομοιώνουν την *κεντρική μονάδα επεξεργασίας,* το *δίσκο* (διεργασίες Ε/Ε) και τον *εκτυπωτή* αντίστοιχα.

```
#define CPU 0
#define DISK 1
#define PRN 2
```

Οι παραπάνω δηλώσεις αντιστοιχίζουν τους αριθμούς 0,1,2 στα είδη δρομολογητών που αναφέρθηκαν ώστε να μπορεί το σύστημα τόσο να δημιουργεί τους δρομολογητές όσο και να μοιράζει τις διεργασίες σωστά σε αυτούς.

```
#define QUANTUM 4
#define NULL 0
```

Το quantum εδώ ορίζεται 4. Αυτό βέβαια μπορεί να αλλάζει σύμφωνα με τις απαιτήσεις του εκάστοτε χρήστη.

Κατά τη δημιουργία της η κλάση Scheduler εκχωρεί τη δομή

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

Βοηθητικές μεταβλητές

```
pcb *current;
bool currentisNULL;
pcb *immigrant;
int _type; // καθορίζει τον τύπο του δρομολογητή
bool do_nop;

// ο δημιουργός
Scheduler( int type ) {
      do_nop = 1;
      _type = type;
      currentisNULL = true;
      immigrant = 0;
}
```

Η συνάρτηση Load αναλαμβάνει να φορτώσει από τις ουρές την επόμενη διεργασία στο current, διασφαλίζοντας ότι η διεργασία αυτή δεν έχει τελειώσει (αν δηλ. ο εναπομείνας χρόνος της δεν είναι μηδέν).

```
void Load( ) {
    int i = 0;
    while ( i < 6 )
    {
      if ( !pq[i].IsEmpty() )
            {
                  current = ( pq[i].Pop() );
                  currentisNULL = false;
                  // ελέγχει αν η διεργασία current έχει τελειώσει
                  if ( current.Finished( _type ) )
                  {
                       // αν ναι πετάει σφάλμα
                       throw;
                  }
                  // αλλιώς την εκκινεί
                  current.Start( _type );
                  do_nop = 0;
                  return;
            }
            ++i;
    }
    do_nop = 1;
}
```

Αντίθετα η Unload σταματά την εκτέλεση του current εξετάζοντας αν ο Scheduler τελείωσε με τον συγκεκριμένο πόρο ή αν θα συνεχίσει να εργάζεται με αυτόν.

```
void UnLoad() {
    current.Stop();
    // τελείωσε από αυτό τον πόρο;
    if ( current.getRemaining_time( _type ) == 0 )
```

```
    {
      // αν ναι δημιουργεί μετανάστη
      immigrant = & current;
      currentisNULL = true;
    }
    // αν όχι ήρθε διεργασία μεγαλύτερης προτεραιότητας
    // ή απλά τελείωσε το quantum οπότε κάνει enqueue το current
    else
      pq[current.getPriority()].Enqueue( current );
}


// ελέγχει αν υπάρχουν διεργασίες (έστω και μία) μικρότερης
// προτεραιότητας από την τρέχουσα που περιμένουν.
bool AtLeastOneIsWaiting( int Starting_Priority )
  {
    //throw; (αμφισημία με nop)
    if( Starting_Priority < 0 || Starting_Priority > 6 ) throw;
    while ( Starting_Priority >= 0 )
    {
      if ( !pq[ Starting_Priority ].IsEmpty() ) return true;
      -- Starting_Priority;
    }
    return false;
  }
```

Η feedpcb διοχετεύει τα νέα pcb που προκύπτουν στις ουρές προτεραιοτήτων. Όταν έρχεται ένα νέο pcb στον δρομολογητή, η προτεραιότητα αυτού συγκρίνεται με αυτή του ήδη εκτελούμενου (αν υπάρχει κάποιο). Κατόπιν αν είναι ανάγκη σταματά η εκτέλεση του τρέχοντος pcb, δρομολογείται το εισερχόμενο και μετά καλείται η Load για να συνεχιστεί η επεξεργασία των διεργασιών που εκκρεμούν.

```
void feedpcb( pcb * newpcb ) {
    // άν υπάρχει ήδη κάποιο pcb που εκτελείται τότε εξετάζει
    // αν το νέο pcb έχει μεγαλύτερη προτεραιότητα
    if ( !currentisNULL )
    {
      // αν ναι τότε αποκαθηλώνει το current
      if ( newpcb->getPriority() > current.getPriority() ) UnLoad();
    }

    // τοποθετεί το νέο pcb στην κατάλληλη ουρά
    pq[newpcb->getPriority()].Push( * newpcb );
    do_nop = 0;
    Load();
}
```

Με τη συνάρτηση execute παρακολουθούμε την εκτέλεση των διεργασιών, ρυθμίζοντας την μετανάστευση μεταξύ των διαφόρων δρομολογητών. Ο σχολιασμός πάνω στον ίδιο των κώδικα μας δίνει καλύτερη εικόνα του έργου που επιτελεί η συγκεκριμένη συνάρτηση. Οι έξοδοι προς τον χρήστη έχουν αφαιρεθεί για να αποφευχθούν παρανοήσεις.

```
void execute() {
    // αν δεν υπάρχει τρέχον pcb τότε φόρτωση ενός
    if ( currentisNULL )
    {
      Load();
```

```
      // αν δεν φορτώθηκε κάτι τότε περιμένουμε
      if ( do_nop )
      {
        return;
      }
    }

    // αν φορτώθηκε κάτι ή ηταν φορτωμένο από πρίν τότε
    if ( !currentisNULL )
    {
      // αν δεν έχει τελειώσει με αυτό τον πόρο
      if ( !current.Finished( _type ) )
      {
        // αν δεν έχει ξεπεράσει το όριο εκτέλεσης
        if ( current.getDone_time() <= QUANTUM )
          current.Do_Time();
        else
        {
          // και υπάρχει ανταγωνιστής
          // επιστρέφει στις ουρές
          if ( AtLeastOneIsWaiting( current.getPriority() ) )
          {
            UnLoad();
            Load();
          }
          // είναι ασφαλές να γίνει εκτέλεση αφου στην χειρότερη
          // περίπτωση θα επιστρέψει η ίδια διεργασία
          current.Do_Time();
        }
      }
      // άν έχει τελειώσει με τον πόρο αυτό αιτείται μετανάστευσης
      else
      {
        immigrant = & current;
        currentisNULL = true;
        // μετά την μετανάστευση επιχειρούμε φόρτωση
        Load();
        // αν απέτυχε τότε περιμένουμε
        if ( do_nop )
        {
          return;
        }
        // αλλιώς το εκτελούμε
        else
          current.Do_Time();
      }
    }
}

//  επιστρέφει ένα δείκτη p που δείχνει στον immigrand
pcb * getImmigrant() {
    // Μόλις πάρει ο αναγνώστης το pcb ο μετανάστης γίνεται NULL
    pcb * p = immigrant;
    immigrant = NULL;
    return p;
}

// επιστρέφει τον τύπου του δρομολογητή
int getType()
  {
    return _type;
```

```
  }

// ενημερώνει τον δρομολογητή αν πρέπει να εργαστεί ή όχι
bool Nop() {
    return do_nop;
}
```

Ο Κεντρικός έλεγχος

Τις ανωτέρω κλάσεις και δη τους δρομολογητές πόρων ρυθμίζουν μια σειρά από συναρτήσεις χαλαρά συσχετισμένες , που χειρίζεται η main() .
Παρατίθεται σχολιασμένος ο κώδικας .

Είναι το σύστημα των δρομολογητών . Το όρισμα που λαμβάνουν κατά την δημιουργία καθορίζει μοναδικά τον πόρο .

```
Scheduler S[] = { Scheduler(CPU), Scheduler(DISK), Scheduler(PRN) };
```

Η παρακάτω συνάρτηση ελέγχει εάν όλοι οι δρομολογητές αδρανούν. Αποτελεί τερματική συνθήκη .

```
bool AllSchedulersAreEmpty() {
    int cursor = 0;
    while ( cursor < 3 ) {
        if ( !S[cursor].Nop() ) return false;
        ++cursor;
    }
    return true;
}
```

```
Η συνάρτηση προκαλεί την επεξεργασία για όλους τους δρομολογητές .

void AllSchedulersExecute() {
    int cursor = 0;
    while ( cursor < 3 ) {
        S[cursor].execute();
        ++cursor;
    }
}
```

```
Η συνάρτηση πραγματοποιεί την επεξεργασία των pcb ώστε να εξακριβωθεί
αν θα τροφοδοτηθούν σε κάποιο δρομολόγητή , ή θα φονευθεί [ κάτι που
γίνεται με το delete εδώ ] .

void HandleApplicant( pcb * applicant ) {

    if ( !applicant->Migrate() ) return delete applicant ;
    //αλλιώς
    S[applicant->getMode()].feedpcb( applicant );
}
```

Οδηγίες χρήσης

Τρόπος χρήσης

Ο δρομολογητής εξάγει τα αποτελέσματα της εργασίας του στο ρεύμα κονσόλας standard out .
Το εκτελέσιμο OSProject2.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 της τυπικής εξόδου στο αρχείο results.txt .

Εφικτές τροποποιήσεις κώδικα

Η αρθρωτή σχεδίαση της εργασίας επιτρέπει μερικές ενδιαφέρουσες τροποποιήσεις που το φέρνουν πιο κοντά στην πραγματικότητα . Δεδομένης ωστόσο της φύσης της εργασίας κάθε αλλαγή απαιτεί αναμεταγλώττιση . Σε αντίθεση ωστόσο με την πρώτη εργασία επειδή υπεισέρχεται ο παράγοντας της τυχαιότητας δεν έχουμε δημιουργήσει παραδείγματα εκτέλεσης για κάθε τροποποίηση . Αφού για να τεκμηριώσουμε την αποτελεσματικότητα των τροποποιήσεων θα πρέπει να εκτελέσουμε στατιστική μελέτη .

Τροποποίηση του quantum

Αρκεί να αλλάξει η εντολή προεπεξεργαστή #define QUANTUM 4 σε κάποια άλλη θετική ακέραια τιμή . Οποιαδήποτε άλλη τιμή θα οδηγήσει σε ατέρμονα βρόγχο που δεν θα εντοπιστεί στην μεταγλώττιση . Ένα δείγμα με διαφορετικό χρονοτεμάχιο προεκχώρησης είναι και η Τρίτη δοκιμαστική εκτέλεση .

Τροποποίηση αριθμού προτεραιοτήτων

Για να υποστηρίξει περισσότερες ή λιγότερες ουρές προτεραιότητας , αλλάζουμε την εντολή προεπεξεργαστή #define PRIORITIES 7 .

Τροποποίηση του αριθμού πόρων

Για να υποστηρίζει περισσότερους ή λιγότερους πόρους το σύστημα δρομολογητών αρκεί να αλλάξουμε την δήλωση #define RESOURCES 3 στο αρχείο pcb.h

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 πρώτο μέρος της εργασίας είδαμε το όφελος με το δείγμα εκτέλεσης #4 και πετύχαμε σημαντικές βελτιώσεις ταχύτητας . Για να γίνει αυτό αρκεί να αλλάξουμε την εντολή `#define arrival_action` από `Enqueue` σε `Push` .

Τροποποίηση του πλήθους εργασιών στη στοίβα και της διάρκειάς τους

Κατά την δημιουργία των διεργασιών το πλήθος εργασιών που πρέπει να εκτελέσουν σε κάθε πόρο κυμαίνεται, όπως και το πλήθος αυτών . Η συμπεριφορά της δημιουργίας μπορεί να αλλάξει με την τροποποίηση της δημιουργίας τους στην main() . Η υπογραφή της δημιουργίας είναι Generatepcb(int Μοναδικός Αριθμός, float Πιθανότητα Δημιουργίας, int Κάτω Όριο Πλήθους, int Άνω Όριο Πλήθους, int Κάτω Όριο Διάρκειας, int Άνω Όριο Διάρκειας) .

Παράδειγμα εκτέλεσης

Στην εκτέλεση των δειγμάτων / στιγμιοτύπων χρησιμοποιούμε πάγια το διάνυσμα < σπόρος , quantum > .

Δείγμα #1 μια δρομολόγηση με διάνυσμα < 60 , 4 >

Η έξοδος του προγράμματος είναι :

```
Χρόνος : 0
Ο πόρος : 0 αναμένει
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
το pcb 0 μεταπηδά απο τον πόρο  στο 0
εισήχθη νέο pcb 0
Χρόνος : 1
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 7
Εκτελείται το pcb 0 και απομένει 6
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
Χρόνος : 2
Εκτελειται απο τον πόρο:0 το pcb 0.Εναπομένας χρόνος για ολοκλήρωση:6
Εκτελείται το pcb 0 και απομένει 5
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
το pcb 1 μεταπηδά απο τον πόρο  στο 2
εισήχθη νέο pcb 1
Χρόνος : 3
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 5
Εκτελείται το pcb 0 και απομένει 4
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 1 . Εναπομένας χρόνος για ολοκλήρωση : 6
Εκτελείται το pcb 2 και απομένει 5
το pcb 2 μεταπηδά απο τον πόρο  στο 2
εισήχθη νέο pcb 2
Χρόνος : 4
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 0 και απομένει 3
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 9
Εκτελείται το pcb 2 και απομένει 8
Χρόνος : 5
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 0 και απομένει 2
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 8
Εκτελείται το pcb 2 και απομένει 7
Χρόνος : 6
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 0 και απομένει 1
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 7
Εκτελείται το pcb 2 και απομένει 6
Χρόνος : 7
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 1
```

```
Εκτελείται το pcb 0 και απομένει 0
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 6
Εκτελείται το pcb 2 και απομένει 5
Χρόνος : 8
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 0 αναμένει
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 5
Εκτελείται το pcb 2 και απομένει 4
Χρόνος : 9
το pcb 0 μεταπηδά απο τον πόρο  στο 1
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 1 και απομένει 3
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 2 και απομένει 3
Χρόνος : 10
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 1 και απομένει 2
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 2 και απομένει 2
Χρόνος : 11
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 1 και απομένει 1
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 2 και απομένει 1
Χρόνος : 12
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 1
Εκτελείται το pcb 1 και απομένει 0
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 1
Εκτελείται το pcb 2 και απομένει 0
Χρόνος : 13
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 2 αναμένει

Ο μέσος χρόνος απόκρισης σε δείγμα 3 ήταν 2.66667

Φυσιολογικός τερματισμός στο χρόνο : 14
```

Παρατηρούμε πως δεν έγινε άφιξη διεργασίας την στιγμή 1

Δείγμα #2 μια δρομολόγηση με διάνυσμα < 60 , 1 >

τα αποτελέσματα είναι :

```
Χρόνος : 0
Ο πόρος : 0 αναμένει
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
το pcb 0 μεταπηδά απο τον πόρο  στο 0
εισήχθη νέο pcb 0
Χρόνος : 1
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 7
Εκτελείται το pcb 0 και απομένει 6
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
```

```
Χρόνος : 2
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 6
Εκτελείται το pcb 0 και απομένει 5
Ο πόρος : 1 αναμένει
Ο πόρος : 2 αναμένει
το pcb 1 μεταπηδά απο τον πόρο  στο 2
εισήχθη νέο pcb 1
Χρόνος : 3
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 5
Εκτελείται το pcb 0 και απομένει 4
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 1 . Εναπομένας χρόνος για ολοκλήρωση : 6
Εκτελείται το pcb 2 και απομένει 5
το pcb 2 μεταπηδά απο τον πόρο  στο 2
εισήχθη νέο pcb 2
Χρόνος : 4
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 0 και απομένει 3
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 9
Εκτελείται το pcb 2 και απομένει 8
Χρόνος : 5
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 0 και απομένει 2
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 8
Εκτελείται το pcb 2 και απομένει 7
Χρόνος : 6
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 0 και απομένει 1
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 7
Εκτελείται το pcb 2 και απομένει 6
Χρόνος : 7
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 1
Εκτελείται το pcb 0 και απομένει 0
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 6
Εκτελείται το pcb 2 και απομένει 5
Χρόνος : 8
Εκτελειται απο τον πόρο : 0 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 0 αναμένει
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 5
Εκτελείται το pcb 2 και απομένει 4
Χρόνος : 9
το pcb 0 μεταπηδά απο τον πόρο  στο 1
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 1 και απομένει 3
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 4
Εκτελείται το pcb 2 και απομένει 3
Χρόνος : 10
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 1 και απομένει 2
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 3
Εκτελείται το pcb 2 και απομένει 2
Χρόνος : 11
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 1 και απομένει 1
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 2
Εκτελείται το pcb 2 και απομένει 1
Χρόνος : 12
Ο πόρος : 0 αναμένει
Εκτελείται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 1
Εκτελείται το pcb 1 και απομένει 0
```

```
Εκτελείται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 1
Εκτελείται το pcb 2 και απομένει 0
Χρόνος : 13
Ο πόρος : 0 αναμένει
Εκτελειται απο τον πόρο : 1 το pcb 0 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 1 αναμένει
Εκτελειται απο τον πόρο : 2 το pcb 2 . Εναπομένας χρόνος για ολοκλήρωση : 0
Ο πόρος : 2 αναμένει

Ο μέσος χρόνος απόκρισης σε δείγμα 3 ήταν 2.66667

Φυσιολογικός τερματισμός στο χρόνο : 14
```

Παρατηρούμε πως η έξοδος ταυτίζεται πλήρως με αυτήν του δείγματος #1 . Στην μελέτη μας δεν καταφέραμε να βρούμε ένα διάνυσμα εκτέλεσης ώστε : OSProject2(<x,y>)={ … Y_i, Y_{i+1} … } , Y1≠Y2 .
Αυτό μάλιστα επεκτείνεται και στους άλλους παράγοντες με τους οποίους πειραματιστήκαμε . Αποτέλεσμα που υπονοεί ότι ασυμπτωτικά μόνο η αρχιτεκτονική του συστήματος επηρεάζει την απόδοση .

ΘΕΜΑΤΑ 1ης και 2ης ΣΕΙΡΑΣ ΕΡΓΑΣΙΩΝ

Ο δρομολογητής είναι το κομμάτι του λειτουργικού που ελέγχει την σειρά με την οποία οι διεργασίες εκτελούνται από τον επεξεργαστή.

Το αντικείμενο της εργασίας σας αφορά την προσομοίωση λειτουργίας ενός δρομολογητή στη γλώσσα προγραμματισμού C++.

Η εφαρμογή σας θα προσομοιώνει τον αλγόριθμο Priority Queues και θα εκτυπώνει συγκριτικούς πίνακες με μέσο όρο απόκρισης και μέσο όρο επιστροφής για κάθε διεργασία.

1η ΕΡΓΑΣΙΑ:

Προδιαγραφές προσομοίωσης:

Η εφαρμογή θα ζητά από τον χρήστη τον αριθμό των διεργασιών προς διεκπεραίωση από το λειτουργικό σύστημα, τον χρόνο άφιξης, τον χρόνο καταιγισμού τους όπως και την προτεραιότητα κάθε διεργασίας. Ο χρόνος άφιξης και καταιγισμού αντιστοιχεί σε μονάδες χρόνου του ρολογιού (ticks) και η ενότητα χρόνου για προεκχώρηση είναι 1 tick. Οι προτεραιότητες είναι εκφρασμένες στην κλίμακα 1 έως 7, όπου το 1 αντιστοιχεί στην υψηλότερη προτεραιότητα και το 7 στην χαμηλότερη προτεραιότητα.

2η Εργασία :

Η παραπάνω εφαρμογή να προσαρμοστεί έτσι ώστε να δημιουργούνται αυτόματα οι διεργασίες, ο χρόνος άφιξης τους, ο χρόνος καταιγισμού τους και οι προτεραιότητες τους. Η αυτόματη δημιουργία γίνεται τυχαία (μέσω ενός random generator) και η διαδικασία αυτή συνεχίζεται μέχρι να την σταματήσει ο χρήστης.

Επίσης να προσαρμόσετε τον αλγόριθμο σας, έτσι ώστε υπάρχουν και I/O προοριζόμενες διεργασίες. Σε αυτή την περίπτωση, η εφαρμογή σας πρέπει κάνει κατάλληλη διαχείριση για τις συγκεκριμένες διεργασίες.

Τα θέματα των εργασιών θα πραγματοποιηθούν από **ομάδες** φοιτητών/τριών
Κάθε **ομάδα** θα αποτελείται από **3** άτομα

Κάθε υποβολή εργασίας πρέπει να περιλαμβάνει:
Το αρχείο με τις διεργασίες που δημιουργήθηκαν, τον κώδικα (*πηγαίο* και *εκτελέσιμο*), αρχείο README.txt (με περιγραφή των βασικών δομών του κώδικα) και αναλυτική τεκμηρίωση (*documentation*).

ΠΡΟΘΕΣΜΙΕΣ ΥΠΟΒΟΛΗΣ ΕΡΓΑΣΙΩΝ:

1ης Εργασίας *Τρίτη 23 Νοεμβρίου 2004*

2ης Εργασίας *Πέμπτη 9 Δεκεμβρίου 2004*