**ROMÂNIA**

**MINISTERUL APĂRĂRII NAŢIONALE**

**ACADEMIA TEHNICĂ MILITARĂ**

**„FERDINAND I”**

**FACULTATEA DE SISTEME ELECTRONICE ŞI INFORMATICE MILITARE**

**Specializarea: Calculatoare și sisteme informatice pentru apărare**

**și securitate națională**



**Implicații ale realocării resurselor asupra fezabilității ordonării activităților**

CONDUCATOR STIINFIC:

**CS. III Dana-Mihaela VÎLCU**

ABSOLVENT:

**Std. Sg. Maj. Mihai-Valentin BADEA**

Conține \_\_\_\_\_\_\_\_\_\_\_ file

Inventariat sub nr. \_\_\_\_\_\_

Poziția din indicator: \_\_\_\_

Termen de păstrare: \_\_\_\_\_

**București**

**-2022-**

Cuprins

[Listă de figuri 3](#_Toc101980323)

[Listă de tabele 4](#_Toc101980324)

[1. Introducere 5](#_Toc101980325)

[1.1. Prezentare generala 5](#_Toc101980326)

[1.2. Importanța temei 5](#_Toc101980327)

[1.3. Utilitatea sistemului 6](#_Toc101980328)

[1.4. Aplicații de project management 6](#_Toc101980329)

[2. Stadiul actual 8](#_Toc101980330)

[2.1. Soluții similare 8](#_Toc101980331)

[2.1.0. Trello 8](#_Toc101980332)

[2.1.1. Asana 9](#_Toc101980333)

[2.2. Articole și cărți ce abordează problema propusă 10](#_Toc101980334)

[2.2.1. An updated survey of variants and extensions of the resource-constrained project scheduling problem 10](#_Toc101980335)

[*2.2.2.* Resource-Constrained Project Scheduling: *Models, Algorithms, Extensions and Applications* 11](#_Toc101980336)

[3. Ordonarea task-urilor în cadrul sistemelor de operare 13](#_Toc101980337)

[3.1. Definiții de bază 13](#_Toc101980338)

[3.2. Ordonarea în sisteme uni-procesor și multi-procesor 14](#_Toc101980339)

[3.2.1. Conditii sufieciente pentru seturi de task-uri fara conflicte 19](#_Toc101980340)

# Listă de figuri

Fig. 1.0 x pag

# Listă de tabele

|  |  |  |  |
| --- | --- | --- | --- |
| **Nr. crt.** | **Tabel** | **Descriere** | **Pagina** |
|  |  |  |  |
|  |  |  |  |

# Introducere

Proiectul își propune crearea unei aplicații software desktop de ordonare a activităților destinată organizațiilor. Ordonarea activităților presupune implementarea unor algoritmi astfel încât resursele să fie folosite cât mai eficient.

## Prezentare generala

Proiectul constă într-o aplicație client-server capabilă să realizeze managementul diferitelor organizații. Astfel, in cadrul aplicației, utilizatorii își pot înregistra compania având astfel in mod implicit rolul de administrator al acesteia. Ca și administrator, utilizatorii pot vizualiza proiectele ce se afla in derulare în cadrul organizației, pot crea conturi angajaților si pot defini resursele companiei. Angajații au posibilitatea să vizualizeze activitățile pe care aceștia le au de realizat, să creeze noi proiecte, iar in cazul in care aceștia au dreptul, pot realiza realocări de resurse în proiectele respective.

Rolul principal al aplicației, acela de a stabili dacă există o ordonare fezabilă a activităților, se va concretiza în momentul în care un proiect este creat sau în cazul în care se realizează realocări de resurse.

## Importanța temei

Dezvoltarea oricărei organizații, instituții sau companii stă in modalitatea în care acestea își manageriază activitățile. Fie că vorbim despre proiecte ale multinaționalelor în care sunt angrenați sute sau chiar mii de angajați (de exemplu pentru realizarea jocului Grand Theft Auto V au lucrat peste 1000 de programatori[[1]](#footnote-1)) sau despre startup-uri ce au un număr restrâns de angajați, organizarea activității fiecărui membru în parte este extrem de importantă pentru o productivitate crescută. Într-o economie de piață extrem de competitivă cum este cea din prezent, o aplicație de management al activităților poate face diferența dintre o companie ce se dezvoltă si una ce se duce încet spre faliment.

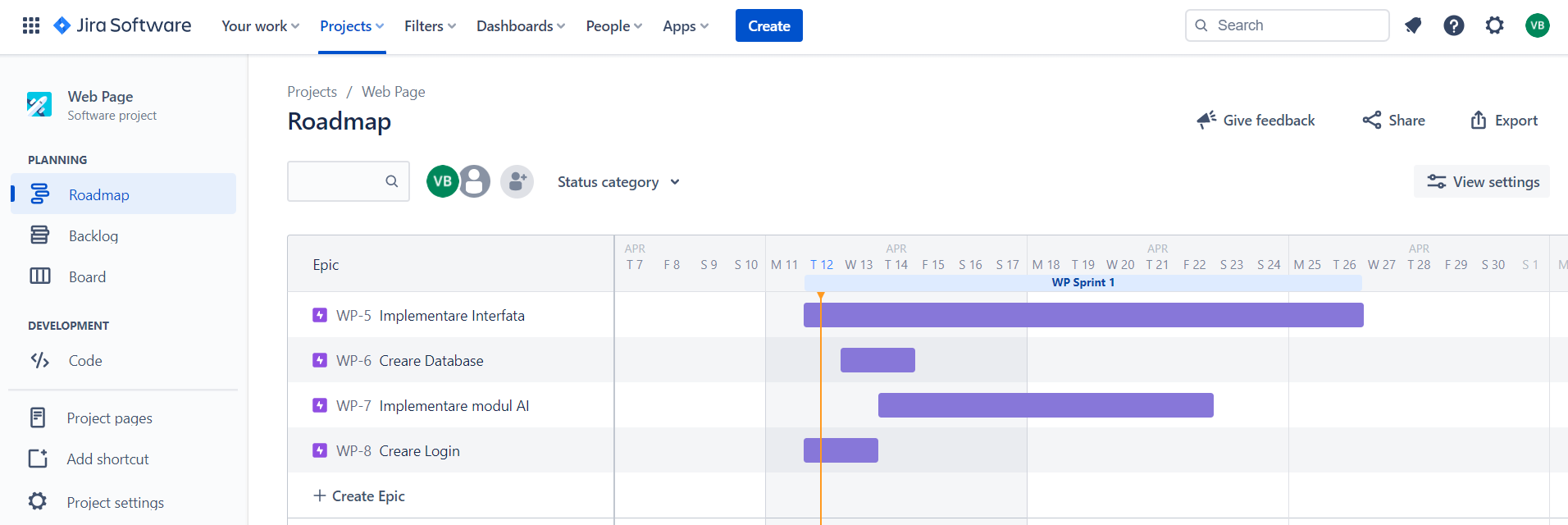
## Utilitatea sistemului

Soluția propusă în această lucrare este utilă pentru organizațiile ce își doresc să li se ofere o ordonare automată a activităților pe care acestea le au de îndeplinit, în funcție de resursele de care dispun. De asemenea, prin intermediul interfeței grafice, se poate monitoriza utilizarea resurselor si modalitatea în care sunt organizate activitățile fiecărui angajat.

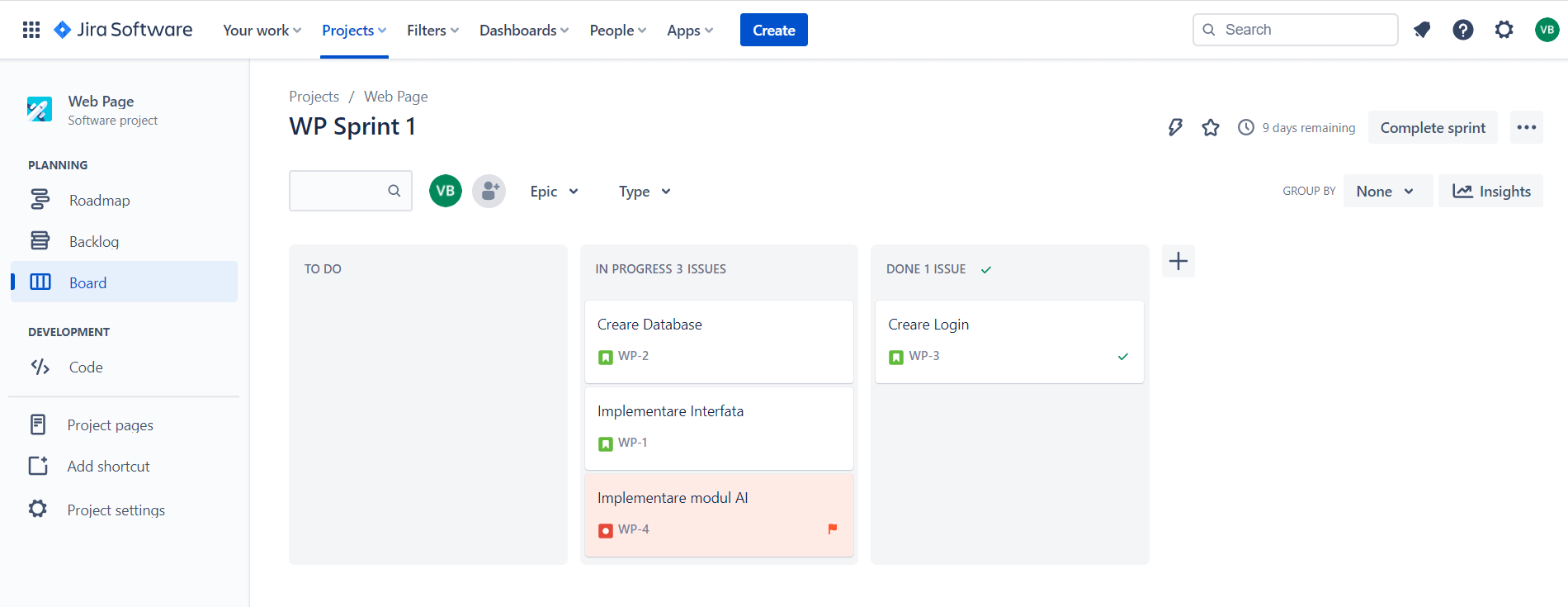
Fiecare angajat ce se autentifica în aplicație este capabil să vizualizeze activitățile pe care acesta le are de realizat in următoarea perioadă. În cazul în care se impune, acesta poate realiza realocări ale resurselor in cadrul proiectelor în care este deja asignat, lucru ce va duce în mod implicit la apelarea algoritmului de ordonare ce va determina dacă există o nouă soluție fezabilă pentru datele actuale.

## Aplicații de project management

În clipa de față există numeroase aplicații de project management ce pot fi folosite pentru gestionarea activităților, însă principala problemă pe care am întâlnit-o in analizarea acestora a fost reprezentata de faptul ca aceste aplicații nu realizează in mod automat o ordonare a activităților pe care utilizatorul le definește. Astfel, rolul acestora este mai degrabă acela de a fi un „to do list”. Desigur, o mare majoritate a acestora oferă alte facilitați precum accesul prin intermediul interfeței la Git-ul proiectului, făcând astfel mult mai ușoară actualizarea acestuia, însă pe partea de ordonare automata a activităților acestea sunt deficitare.



*Fig. 1.0. Jira Roadmap – aplicație de project management*



*Fig. 1.1. Jira Board – aplicație de project management*

Un exemplu de aplicație folosită foarte frecvent in industria IT este Jira. Aplicația are o interfață ce permite o foarte ușoară creare si vizualizare a activităților din cadrul proiectelor.

In cadrul tab-ului „Board” se pot vizualiza activitățile împărțite pe 3 categorii:

* „to do” – activități ce urmează a fi realizate
* „in progress” – activități curente ce se află in desfășurare
* „done” – activități ce au fost finalizate

În cadrul tab-ului „Roadmap” se pot vizualiza activitățile din punct de vedere temporal. Consider ca acest mod de vizualizare este mult mai intuitiv pentru utilizatori, dar în continuare problema ordinii în care se vor executa aceste activități este nedeterminată și rămâne astfel la latitudinea celui ce le execută.

# Stadiul actual

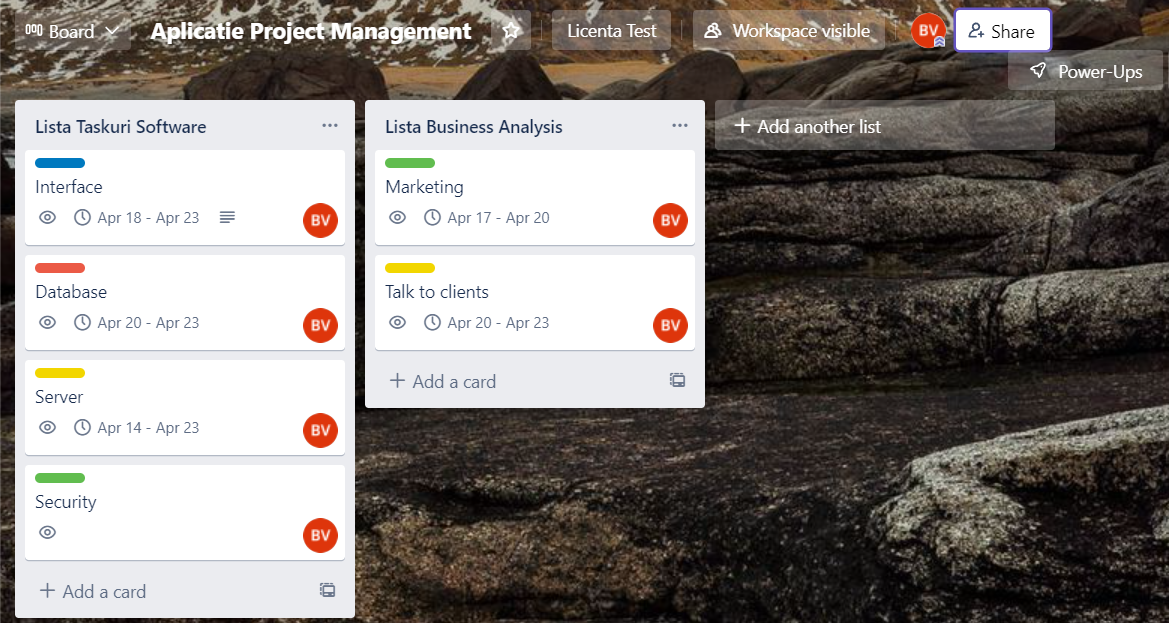
## Soluții similare

În clipa de față există numeroase soluții software ce pot fi folosite pentru managementul proiectelor. Printre acestea amintim: Jira, Trello si Asana.

Pe lângă soluțiile gratuite accesibile pe Internet, sau demo-urile celor ce necesită un abonament plătit, soluția prezentată în prezenta lucrare implementează și o sugestie de ordonare a activităților, lucru ce poate fi util organizațiilor în realizarea obiectului lor de activitate. În continuare, voi prezenta câteva soluții software similare ce pot fi găsite pe Internet, prezentând totodată modul acestora de utilizare.

### 2.1.0. Trello

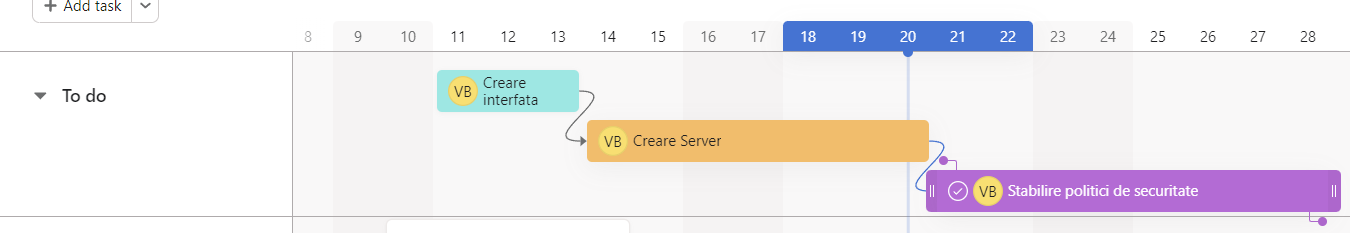
Trello reprezintă o aplicație web în care utilizatorii își pot organiza activitățile pe care le au de realizat în cadrul organizațiilor din care fac parte. Aplicația permite crearea de proiecte în care activitățile sunt împărțite pe liste de lucru. Ca și parametrii, fiecărei activități îi sunt atribuite o descriere, o perioadă în care aceasta trebuie realizată și un membru al echipei ce trebuie să o execute. De asemenea, se mai pot atașa link-uri sau fișiere necesare completării activității. În varianta demo gratuită se pot afișa activitățile doar sub forma unui Board. Pentru varianta plătită a aplicației există beneficii suplimentare precum: conectarea la Google Drive, Dropbox si altele; posibilitatea de vizualizare a activităților sub forma unui calendar, lucru ce face mult mai ușor de urmărit evoluția acestora.



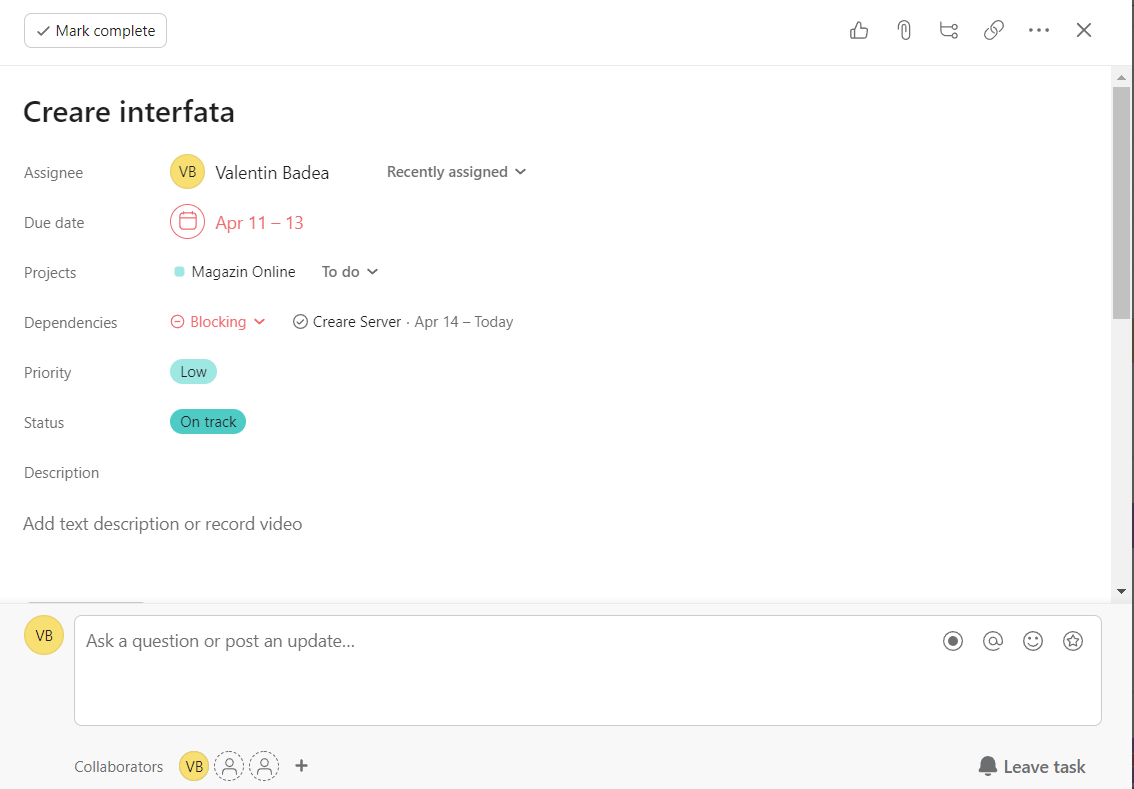
*Fig. 2.0. Trello – interfața grafică*

### Asana

O aplicație ce oferă mai multe module pentru utilizatori este Asana. Aceasta oferă în mod implicit 3 moduri de vizualizare a activităților: Calendar, Board si List. În ceea ce privește parametrii pe care o activitate îi are, aceștia sunt : utilizatorul ce trebuie sa îndeplinească task-ul, perioada de desfășurare a activității si prioritatea. În cadrul aplicației sunt definite 3 niveluri de prioritate: Low, Medium si High. Pe lângă acestea, o caracteristică foarte interesantă este reprezentată de posibilitatea de a defini relații de precedență, astfel putând arăta dacă execuția unei activități blochează sau este blocată de o alta. Săgețile ce se observă în imaginea următoare reprezintă relațiile de precedență definite de utilizator. În cazul în care se încearcă schimbarea perioadei de execuție a unei activități care nu ar respecta relațiile definite, se va realiza in mod automat o schimbare în perioada de execuție și pentru activitatea ce nu ar mai îndeplini condițiile impuse.



*Fig. 2.1. Asana – afișarea timeline-ului*



*Fig. 2.2. Asana – Caracteristicile unei activități*

## Articole și cărți ce abordează problema propusă

În secțiunea următoare voi prezenta baza științifică ce motivează alegerea temei curente și importanța pe care o are in cadrul desfășurării activităților de project management.

### An updated survey of variants and extensions of the resource-constrained project scheduling problem[[2]](#footnote-2)

Articolul precizat definește problema ordonării activităților având constrângeri de resurse, având scopul de a găsi o ordonare a activităților fără să fie încălcate constrângerile date de relațiile de precedență și de numărul de resurse disponibil, astfel încât timpul de îndeplinire al tuturor activităților să fie minim. Se specifică faptul că această problemă a devenit una standard in domeniul organizării proiectelor, atrăgând un număr mare de cercetători ce au venit cu diverse proceduri de rezolvare.

În cadrul articolului sunt prezentate pe scurt diferite variații ale unor soluții implementate de către cercetători in domeniu.

În Capitolul 2 se specifică faptul că problema de bază a ordonării activităților cu constrângeri pe resurse necesită procesarea task-urilor in mod non-preemptiv, ceea ce înseamnă ca odată începută o activitate, aceasta își continua execuția până în punctul în care este finalizată. Articolul precizează faptul că în lucrarea *„Handbook on Project Management and Scheduling Vol.1 ”*[[3]](#footnote-3) autorii oferă o soluție în care activitățile pot fi întrerupte fără nicio restricție. Tot in cadrul aceluiași capitol, este abordată problema numărului de resurse solicitat de către fiecare activitate în parte. Se specifica faptul că în cadrul problemei standard, orice activitate *Ai* care necesită o cantitate de *rik* din resursa *k*, nu își va schimba aceasta valoare pe parcursul procesării acesteia. Se oferă o referință către lucrarea redactata de *Hartmann(2013)*[[4]](#footnote-4), în care autorul abordează și situația în care activitatea *Ai* necesită o cantitate de *rikt* unități din resursa *k* la perioada *t* din timpul său total de procesare.

### Resource-Constrained Project Scheduling: *Models, Algorithms, Extensions and Applications[[5]](#footnote-5)*

În capitolul 1 al cărții este definită problema ordonării activităților cu constrângeri pe resurse ca fiind o problemă de optimizare combinatorială. O problemă de optimizare combinatorială este definită de o soluție discretă în spațiul X sau care poate fi redusa la un set discret, si printr-un subset de soluții fezabile asociat unei functii obiectiv . Astfel, o problemă de optimizare combinatorială presupune găsirea unei soluții fezabile astfel încât este minimizată sau maximizată.

Având la bază informația teoretică despre probleme de optimizare combinatorială, autorul definește problema curenta de ordonare a activităților cu constrângeri asupra resurselor ca fiind dată de următorul tuplu :

Un set este folosit pentru a identifica activitățile ce constituie baza proiectului. Prin convenție, activitatea *A0* indică începerea procesului de ordonare, iar activitatea *An+1* indică, în mod simetric, finalizarea acestuia. Mulțimea rămasă prin eliminarea celor doua activități, , reprezintă activitățile propriu-zise ce trebuie sa fie executate.

Duratele de execuție a activităților sunt reprezentate printr-un vector *p* in , unde *pi* reprezintă durata activității *Ai*. O caracteristică ce trebuie menționată este legată de valorile speciale folosite pentru activitățile *A0* si *An+1* , acestea având duratele asociate *p0* = *pn+1* = 0.

Relațiile de precedență sunt definite de o colecție E având perechi de forma (*Ai* , *Aj*) care denotă faptul că activitatea *Ai* va preceda activitatea *Aj* . Prin definirea acestor relații de precedență se poate crea un graf G(V,E) în care fiecărui nod îi este asociat o activitate. Pentru ca relațiile de precedență să aibă sens, în cadrul grafului nu vor exista cicluri. Deoarece precedența reprezintă o relație binară tranzitivă, prezența unei rute în cadrul grafului G de la nodul *Ai* către nodul *Aj* denotă faptul că *Ai* va începe prima. Având in vedere cele doua activități ce reprezintă începutul, respectiv finalizarea ordonării, rezultă faptul că activitatea *A0* va preceda toate celelalte activități, iar activitatea *An+1* va fi succesorul acestora.

Resursele sunt organizate sub forma unei mulțimi .

Disponibilitatea resurselor este reprezentată de un vector *B* in , unde *Bk* reprezintă numărul disponibil din resursa *Rk* .

Cererile de resurse sunt reținute într-o matrice de dimensiuni notată cu *b*. Astfel, *bik* reprezintă cantitatea din resursa *Rk* folosită în fiecare perioadă de timp pentru executarea activității *Ai* .

O ordonare reprezintă un punct *S* in unde *Si* reprezintă punctul de început al activității *Ai* . *Ci* reprezintă timpul de completare al activității *Ci* , unde . *S0* reprezintă un punct de referință pentru începutul proiectului, astfel, având in vedere mențiunile anterioare, *S0* = 0. Pentru ca o ordonare să fie considerată fezabilă, aceasta trebuie să respecte simultan atât relațiile de precedență (1.0), cât și constrângerile rezultate din numărul limitat de resurse disponibile (1.1). Mulțimea *Åt* reprezintă totalitatea activităților ce se află în execuție la momentul de timp *t* : .

(1.0)

(1.1)

Timpul de finalizare al ordonării este egal cu timpul de început al activității *Sn+1*. Autorul folosește termenul de *„makespan”* pentru a reprezenta timpul necesar finalizării tuturor activităților. Astfel, având in vedere toate cele prezentate mai sus, autorul oferă următoarea definiție:

DEFINITIA 1.0. - RCPSP[[6]](#footnote-6) reprezintă problema găsirii unei ordonări non-preemptive S având makespan-ul minim Sn+1 care respectă constrângerile date de relațiile de precedenta (1.0) si de resurse (1.1).

# Ordonarea task-urilor în cadrul sistemelor de operare

În capitolul acesta vor fi prezentate noțiuni teoretice ce stau la baza ordonării task-urilor în cadrul sistemelor de operare, prezentându-se astfel principiile și algoritmii folosiți. Următoarele informații teoretice au fost selectate din cartea *„Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks”[[7]](#footnote-7)*.

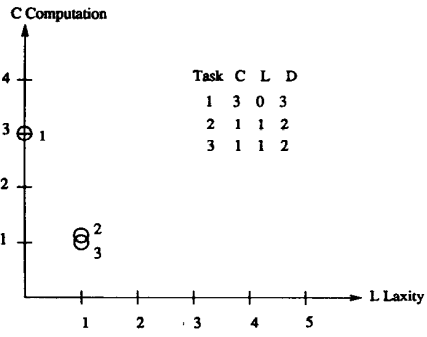
## Definiții de bază

În general, un task poate fi caracterizat prin următorii parametri întregi: S (momentul la care task-ul își începe execuția), C (numărul de unități CPU) și D (deadline-ul). De asemenea, mai pot fi adăugate și constrângeri adiționale precum relații de precedență sau periodicitate. Starea fiecărui task care și-a început execuția poate fi caracterizat prin doi parametrii: timpul de procesare rămas *C(i)* la momentul *t* = i și deadline-ul *D(i)* când acesta își termină execuția.

Laxitatea unui task la momentul *t* = i este definită de relația:

(2.0)

Această caracteristică a taskului definește prioritatea cu care trebuie executat task-ul respectiv . În mod evident, un task cu laxitate zero trebuie executat imediat și fără întreruperi. Laxitatea negativă a unui task arată faptul că deadline-ul a fost depășit. Problema de organizare a taskurilor poate fi modelată prin configurarea de *„token-uri”* în primul cadran al unui plan cartezian ce are pe axa verticală *C*, iar pe cea orizontală *L*. Astfel, un task *j* având parametrii *Cj(i)* și *Lj(i)* la momentul *t* = i, va fi poziționat la *L = Lj(i)* și *C = Cj(i)* în planul *L-C*. Un astfel de exemplu de reprezentare a unui scenariu în care avem un număr de 3 task-uri este prezentat in *Fig. 3.0*. De reținut faptul că o poziție din acest plan poate fi ocupată de mai mult de 1 task în același moment de timp, în cazul de față observându-se faptul că task-urile 2 si 3 ocupă aceeași poziție. De asemenea, în reprezentarea planului nu este trecut numărul de unități CPU ce pot fi folosite, dar acesta va fi folosit în momentul în care se realizează execuția unui task.



*Fig. 3.0. – Reprezentarea a 3 task-uri in planul cartezian L-C*

În cazul în care avem la dispoziție un număr de n procesoare și m task-uri (*n* < *m*) ce se pot executa la momentul *t* = i, atunci se vor putea executa un număr de *n* din totalul de *m* task-uri posibile. Reprezentând acest lucru tot în planul *L-C* menționat anterior, execuția task-urilor se traduce prin deplasarea în jos cu o unitate pe axa paralelă cu *C* și, simultan, spre stânga tot cu o unitate pe axa paralelă cu *L*. Astfel, ecuațiile ce prezintă execuția unui task sunt următoarele:

(3.1)

(3.2)

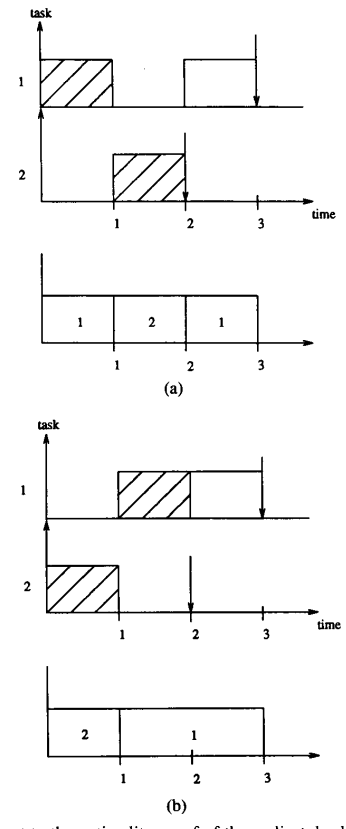
Pentru un task ce nu este executat, ecuațiile sunt următoarele : si . Algoritmul de ordonare este cel ce impune ce set de task-uri se va executa la un moment de timp. În cazul în care un task ajunge în al doilea cadran al planului, atunci acesta și-a ratat deadline-ul și, drept urmare, ordonarea nu este una fezabilă. Contrar acestui caz, dacă un task ajunge la valoarea 0 pe axa C, iar valoarea laxității este pozitivă, atunci task-ul a fost executat cu succes înainte de deadline.

## Ordonarea în sisteme uni-procesor și multi-procesor

În cazul sistemelor cu un singur procesor există algoritmi de ordonare ce pot fi considerați optimi, contrar sistemelor multi-procesor unde studiile au arătat că problema găsirii unui algoritm optim nu este rezolvabilă.

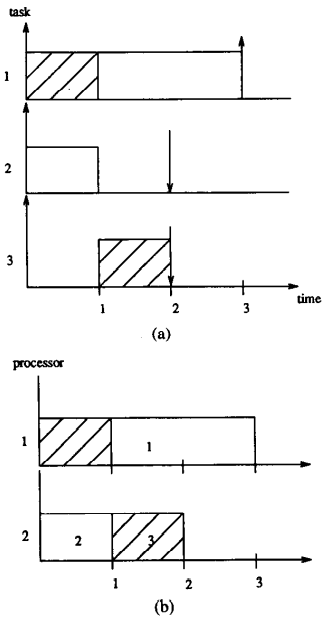
În continuare, voi prezenta 2 algoritmi ce sunt folosiți pentru ordonarea task-urilor: *Earliest Deadline* si *Least Laxity.* **Earliest Deadline** presupune execuția task-ului cu cel mai apropiat deadline, iar **Least Laxity** presupune execuția task-ului ce are cea mai mică valoarea a laxității. În ambele cazuri, dacă există task-uri situate pe aceeași poziție, se va alege unul in mod arbitrar.

Dertouzos[[8]](#footnote-8) a demonstrat caracterul optim algoritmului Earliest Deadline prin faptul că întotdeauna este posibil ca o ordonare fezabilă să fie transformată într-o ordonare ce implementează algoritmul Earliest Deadline. O astfel de situație este prezentată in *Fig. 3.1.* unde avem 2 task-uri: primul are deadline-ul la momentul *t*=3 și are nevoie de 2 unități de procesare, iar al doilea are deadline-ul la momentul *t*=2 si are nevoie de o singură unitate de procesare. În cazul *a)* se prezintă o ordonare fezabilă în care prima dată este executat task-ul cu deadline-ul mai mare, în locul task-ului cu deadline-ul mai mic. Având in vedere cele menționate mai sus, dacă înlocuim varianta de la punctul *a)* cu implementarea algoritmului Earliest Deadline, vom obține de asemenea o soluție fezabilă, așa cum se poate observa în figură la subpunctul *b)*.



*Fig. 3.1. – Demonstrarea caracterului optim al algoritmului Earliest Deadline*

Din nefericire, caracterul optim al algoritmului nu se menține și în cazul sistemelor multi-procesor. Pentru a demonstra acest fapt, vom considera problema de ordonare reprezentată în *Fig. 3.2.a)* în care avem 3 task-uri ce trebuie executate si un număr n=2 procesoare. Task-urile 2 si 3 au ambele deadline-ul peste 2 unități de timp din momentul în care se începe ordonarea, iar task-ul numărul 1 are deadline-ul peste 3 unități de timp. Pentru a își finaliza execuția cu succes, task-ul 1 are nevoie de 3 unități de procesare, iar celelalte 2 au nevoie de 1 unitate de procesare. Algoritmul Earliest Deadline ar presupune prima dată executarea task-urilor 2 si 3, lucru ce ar rezulta în mod imediat trecerea task-ului 1 în al doilea cadran al planului *L-C* prezentat la începutul capitolului. Astfel, nu există o ordonare fezabilă ce se poate realiza pentru scenariul dat în cazul în care se dorește folosirea algoritmului Earliest Deadline.



*Fig. 3.2. – Exemplificarea nemenținerii caracterului optim al algoritmului Earliest Deadline in cadrul sistemelor multi-procesor*

Totuși, scenariul prezentat în figura de mai sus poate avea o ordonare fezabilă în cazul în care este folosit algoritmul Least Laxity. Implementarea acestuia este reprezentată prin diagrama din *Fig. 3.2.b).*

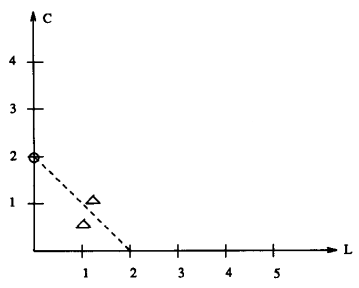
Am demonstrat mai sus caracterul optim al algoritmului Earliest Deadline în cazul sistemelor uni-procesor și modul în care acesta realizează ordonarea task-urilor. O caracteristică ce face ca prezentul algoritm să fie considerat „best effort” este reprezentată de atributele task-urilor pe care acesta le necesită: algoritmul va realiza ordonarea acestora doar în funcție de deadline-ul precizat, necontând timpul de procesare și nici măcar starttime-ul. Astfel, ar fi foarte util daca am putea găsi un algoritm ce are această proprietate pentru sistemele multi-procesor. Din păcate, un astfel de algoritm **nu exista**.

În mod particular, în cazul în care nu cunoaștem *a priori* niciunul din următorii parametri: 1) deadline-uri, 2) timpii de procesare sau 3) starttime-uri, atunci pentru orice algoritm propus există un scenariu de task-uri pentru care nu s-ar putea găsi o soluție fezabilă.

Consideram următoarea lema:

Lema 1: *„*Nu poate exista un algoritm de ordonare optim dacă timpii de procesare ai task-urilor nu sunt cunoscuți *a priori”.*

Pentru a realiza demonstrația lemei, vom considera un scenariu în care avem 3 task-uri si 2 procesoare. Informațiile ce se cunosc despre task-uri sunt următoarele: toate au deadline-ul peste 2 unități de timp, 2 dintre ele necesită o singură unitate de timp de procesare, iar ultimul va necesita 2 unități de timp de procesare. În *Fig. 3.3.* sunt reprezentate prin triunghiuri task-urile cele 2 task-uri cu același timp de procesare necesar, iar ultimul task este reprezentat printr-un cerc. În mod evident, pentru a obține o soluție fezabilă de ordonare, task-ul ce are valoarea laxității egală cu 0 trebuie să fie executat imediat. În cazul în care algoritmul de ordonare nu cunoaște nicio informație legată de timpul de procesare al task-urilor, acestea vor părea identice. Acestea fiind spuse, algoritmul de ordonare ar putea executa task-ul reprezentat printr-un cerc, lucru ce ar presupune eșuarea găsirii unei ordonări fezabile.



*Fig. 3.3. – Reprezentarea unui scenariu cu 3 task-uri în care timpul de procesare nu este cunoscut*

Lema 2: *„*Nu poate exista un algoritm de ordonare optimdacă deadline-urile task-urilor nu sunt cunoscute *a priori”.*

Demonstrația acestei leme se realizează în mod identic cu cea prezentată mai sus.

Lema 3: *„*Nu poate exista un algoritm de ordonare optim dacă distribuția cererilor în timp (start time) nu este cunoscută *a priori”.*

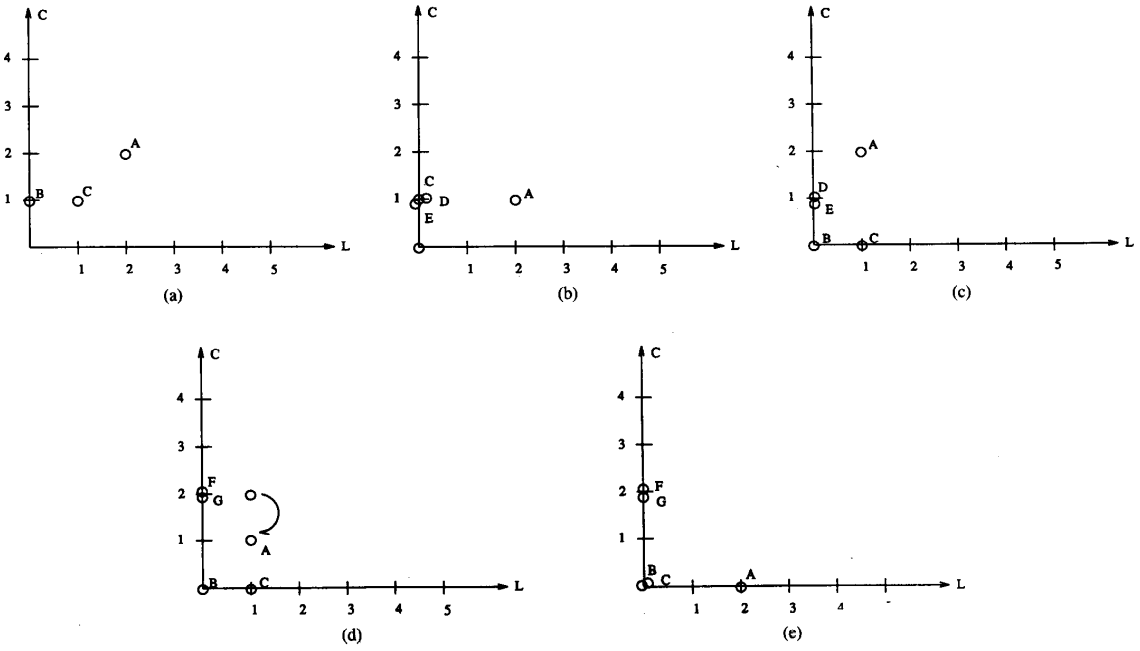
Pentru a realiza demonstrația acestei leme, vom considera un scenariu în care avem inițial 3 task-uri, reprezentate în *Fig. 3.4.a)*, și un număr de 2 procesoare disponibile. Deoarece se observă faptul că task-ul B are laxitatea 0, acesta va trebui executat imediat. În funcție de decizia pe care o ia algoritmul de ordonare, vor exista 3 cazuri posibile.

*Cazul 1:* La momentul *t* = 0 sunt mutate în jos pe axa verticală cu o unitate task-urile A si B. Considerăm că la momentul *t* = 1 mai apar 2 task-uri: C si D, ce au laxitatea egală cu 0, după cum este ilustrat în *Fig. 3.4.b)*. Acest lucru este permis deoarece, după cum precizează lema, nu cunoaștem momentele de timp la care vor fi începute task-urile. În mod evident, având 3 task-uri ce trebuie executate imediat și doar 2 procesoare disponibile, nu există o ordonare fezabilă pentru scenariul propus. Totuși, dacă algoritmul de ordonare ar fi executat prima dată task-urile B si C, scenariu ilustrat în *Fig. 3.4.c)*, atunci am fi putut avea o ordonare fezabilă.

*Cazul 2:* La momentul *t* = 0 sunt mutate în jos pe axa verticală cu o unitate task-urile B si C, astfel task-ul A se va deplasa cu o unitate spre stânga pe axa orizontală. La momentul *t* = 2, sunt introduse 2 noi task-uri: F si G, scenariul ilustrat în *Fig. 3.4.d)*. Având în vedere faptul că cele doua noi task-uri apărute vor avea o prioritate mai ridicată (având laxitatea 0), acestea ar trebui executate în următoarele 2 unități de timp. Problema ce apare este similara cu cea de la primul caz în care avem 3 task-uri ce trebuie executate imediat, dar doar 2 unități de procesare disponibile. Astfel, pentru acest caz nu există o ordonare fezabilă. Totuși, dacă la momentul de timp *t* = 0, ar fi fost executate task-uri A si B, scenariul propus ar fi avut o soluție fezabilă deoarece task-uri C si A și-ar fi încheiat execuția la momentul *t* =1. Acest caz este ilustrat în *Fig.3.4.e)*.

*Cazul 3:* Task-ul B este singurul ce va fi executat la momentul *t* = 0. Astfel, pentru oricare set de task-uri din cele de mai sus ce ar fi introduse ulterior, nu ar exista o ordonare fezabilă.

Prin urmare, pentru orice decizie pe care un algoritm de ordonare ar face-o la momentul inițial, ar exista un set de task-uri (D & E sau F & G) ce ar duce la un blocaj.

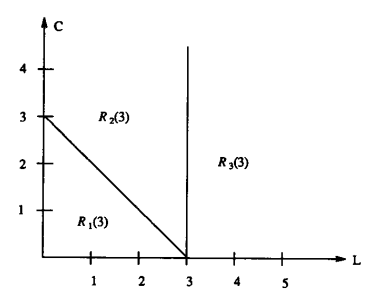


*Fig. 3.4. – Scenarii de task-uri în care nu sunt cunoscute valorile de start-time.*

Pentru a face anumite observații utile asupra ultimei leme menționate, ne vom reîntoarce la scenariul prezentat în *Fig. 3.4.a)*. Putem afirma că există doua alegeri simple pe care algoritmul de ordonare ar putea sa le facă: fie execută task-uri ce necesită un timp mai mic de procesare (B si C), asumându-și astfel riscul ca la momentul *t* = 1 un procesor să nu execute nimic (asemănător cu cazul prezentat în *Fig. 3.4.d)*), fie executa task-ul ce are un timp de procesare mai lung (A) și riscă apariția unor task-uri urgente ce pot apărea, cum se poate observa si în *Fig. 3.4.b)*. Primul caz corespunde unei strategii în care se dorește minimizarea numărului de task-uri ce nu pot fi finalizate, iar al doilea caz corespunde unei strategii în care se dorește minimizarea timpului de procesare per total.

### Condiții suficiente pentru seturi de task-uri fără conflicte

În acest subcapitol, vom continua să păstram notațiile folosite până în acest moment. Astfel, al *j*-lea task va fi notat cu *Jj*, iar parametrii acestuia vor fi, de asemenea, notați cu: *Cj*, *Dj* si *Lj*. Pentru o mai ușoară urmărire a task-urilor, planul *L-C* va fi împărțit în 3 diviziuni după cum se poate observa in *Fig. 3.5.*. Pentru orice valoare întreagă *k*, toate task-urile pot fi împărțite în trei seturi distincte, după cum urmează:



*Fig. 3.5. – Împărțirea cadranului L-C in 3 subdiviziuni. Ilustrație ce ajută la demonstrarea funcției F(k)*

În continuare, vom defini o funcție ce reprezintă surplusul puterii de calcul a sistemului pentru următoarele *k* unități de timp:

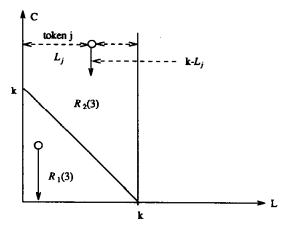
(3.3)

O mențiune foarte importantă este reprezentată de faptul că această ecuație definește o funcție de timp, astfel, de fiecare dată când va fi menționată, se va nota cu *F(k,i)* ceea ce va reprezenta valoarea funcției *F(k)* la momentul de timp *t=i.*

Ne propunem ca prima dată să demonstrăm o condiție necesară pentru a putea ordona un set de task-uri ce au același start-time. Dând o astfel de condiție necesară, va trebui mai apoi să demonstrăm existența unei ordonări fezabile în care valorile de deadline nu sunt depășite. Vor fi alese pentru execuție doar task-urile care și-au depășit start-time-ul, fără a se ține cont de start-time-urile task-urilor viitoare. Cu alte cuvinte, un set de task-uri nu va avea conflicte dacă deadline-urile acestora pot fi atinse atunci când valorile de start-time sunt cunoscute.

Lema 4: *„*O condiție necesară pentru ordonarea unui set de task-uri ale căror valori de start-time sunt egale (la momentul *t=0*) este ca pentru toate valorile *k* > 0, *F(k, 0) ≥ 0” .*

Vom demonstra lema de mai sus luând ca exemplu *Fig. 3.6.* Se observă faptul că orice task ce se afla in *R1* trebuie să fie executat în *k* pași, având în vedere faptul ca deadline-urile acestora nu vor fi mai mari de *k*. Acest lucru implică folosirea a unități de procesare pentru a putea executa toate task-urile din *R1*. Pentru un task ce se află în a doua subdiviziune *R2*, se va permite mutarea acestuia paralel cu axa *L* pentru cel mult *Lj* diviziuni în următorii *k* pași astfel încât acesta să rămână în primul cadran al planului. Pentru restul timpului, aceste task-uri vor fi mutate paralel cu axa *C* până în momentul în care vor atinge axa *L,* terminându-și astfel execuția. Pentru aceste task-uri, timpul de procesare rămas *Cj* poate fi scris sub următoarea formă:



*Fig. 3.6. – Condiții necesare pentru ordonare*

Prin urmare, în următorii *k* pași, fiecare task trebuie mutat cel puțin cu diviziuni pe axa paralelă cu *C.* Timpul minim de procesare necesar executării tuturor task-urilor din *R2* va fi egal cu: . Timpul maxim de procesare ce ar fi necesar unui sistem cu *n* procesoare va fi . Concluzionând, pentru a realiza execuția tuturor task-urilor din primele 2 subdiviziuni *R1* si *R2*, timpul minim de procesare necesar va fi dat de formula:

Întrucât această relație este adevărată pentru , lema este demonstrată.

Teoremă: *„*Dacă există o ordonare fezabilă a unui set de task-uri ce au același start-time, atunci același set de task-uri poate fi ordonat chiar dacă start-time-urile sunt diferite si nu sunt cunoscute *a priori”.*

Cunoașterea dinainte a deadline-urilor și a timpului de procesare este suficientă pentru a se putea realiza ordonarea. Un algoritm „run-time” de ordonare ce are succes este *Least Laxity*. Explicarea algoritmului s-a făcut la începutul acestui capitol.

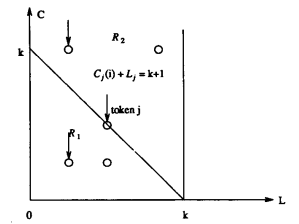
Demonstrația acestei teoreme se va realiza folosindu-se lema menționată anterior. Astfel, cunoaștem faptul că dacă există o ordonare care să îndeplinească deadline-urile task-urilor în situația în care start-time-urile acestora sunt identice. Având această premisă, trebuie să arătăm faptul că niciun deadline nu va fi omis și că după prima mișcare, această ipoteză de inducție se va menține. Astfel, în primul rând, implică ca numărul de task-uri ce se află pe axa *C* să nu depășească numărul de procesoare. În al doilea rând, pentru implică ca pentru dacă pentru ordonarea task-urilor este utilizat algoritmul Least Laxity.

Pentru a demonstra prima propoziție, vom lua ca exemplu . Acest lucru implică, conform ecuației (3.3), . Partea dreaptă a acestei inecuații reprezintă exact numărul de task-uri ce se afla pe axa *C*. Având în vedere faptul că aceste task-uri au laxitatea egală cu 0 și numărul lor este mai mic sau egal cu numărul de procesoare disponibile, acestea vor fi mutate cu o unitate în jos pe axa *C* potrivit algoritmului Least Laxity. Prin urmare, toate task-urile ce se află pe axa *C* vor fi fie mutate, fie vor staționa. Prin faptul că nu se poate ajunge la un eșec imediat al algoritmului, propoziția este demonstrară.

Pentru a demonstra a doua propoziție, va trebui să arătăm că pentru , există un *k’* astfel încât Întrucât din ipoteză, pentru , se poate concluziona că . Utilizând scenariile ilustrate in *Fig. 3.7.* si *Fig. 3.8.,* există 2 cazuri de analizat. De menționat faptul că săgețile din figură reprezintă mutarea task-urilor folosind algoritmul Least Laxity, iar cazurile în care acele săgeți lipsesc reprezintă staționarea task-urilor.

Cazul 1): În acest caz (*Fig. 3.7.*), fiecare task din stânga dreptei , fie a stationat, fie a ajuns la poziția curentă la momentul de timp *i+1*. Alegem , pentru a demonstra că

Vom avea în vedere posibila mutare a unor task-uri din *R3* . Pentru aceste task-uri, astfel acestea nu vor influența Un alt număr de task-uri poate migra din *R2* spre *R1*. Acestea vor contribui în mod egal la si deoarece . Restul de task-uri ce rămân in *R2* își vor păstra aceeași valoare a laxității si nu vor afecta Toate celelalte task-uri ce se afla in *R1* fie vor contribui la scăzând valoarea din *Cj* cu o unitate, fie vor contribui în mod egal la si în cazul în care rămân staționare. Așadar, pentru acest caz, va reprezenta numărul de task-uri din *R1* ce s-au executat în ultima unitate de timp, echivalent cu

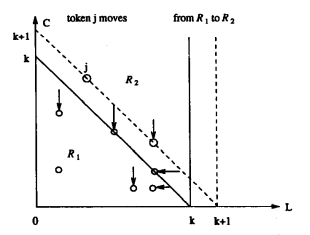


*Fig. 3.7. – Cazul F(k, i+1) ≥ F(k, i)*

Cazul 2) : În acest caz (*Fig. 3.8.*), fiecare task din stânga dreptei , a ajuns pe poziția curenta mișcându-se paralel cu axa *L*. Alegem , pentru a demonstra că

Considerăm task-urile ce vor fi mutate în jos paralel cu axa *C*. Dacă acestea se află în regiunea *R2*, atunci acestea contribuie cu la deoarece . Dacă un task se afla pe dreapta , atunci acesta în mod sigur a fost la momentul anterior de timp pe dreapta , astfel acesta s-a situat in *R1* în ambele momente de timp. Deoarece timpul acestuia de procesare a scăzut, acesta contribuie în mod pozitiv la . În cazul în care task-ul se află deja in *R1*, atunci, în mod evident, contribuția acestuia va fi 1. Concluzionând, în toate cazurile, un task ce a fost mutat vertical în jos va contribui în mod pozitiv la valoarea lui .

În continuare, vom considera task-urile ce vor fi mutate pe axa orizontală. Dacă un asemenea task se află în *R2*, atunci acesta va contribui cu la . Dacă un task se află pe dreapta , atunci acesta a fost mutat la momentul anterior de pe dreapta . Contribuția acestuia la este zero deoarece acesta se află în *R1* în ambele momente de timp și timpul său de procesare nu a scăzut. Așadar, task-urile ce sunt mutate vertical nu vor afecta valoarea . În mod similar, task-urile staționare vor avea o contribuție egală cu 0 sau 1 asupra dacă . Un lucru interesant de notat este reprezentat de faptul că task-uri staționare ce se află pe dreapta se vor muta din la momentul *i* în la momentul *i+1.* În orice caz, acestea vor contribui în mod pozitiv la din moment ce . Astfel, în final, valoarea lui va fi dată de . Această inegalitate este întotdeauna adevărată dacă toate task-urile staționare sunt poziționate în stânga dreptei . Pentru , unul din cere două cazuri prezentate va fi adevărat. Astfel, teorema este demonstrată deoarece în toate cazurile .



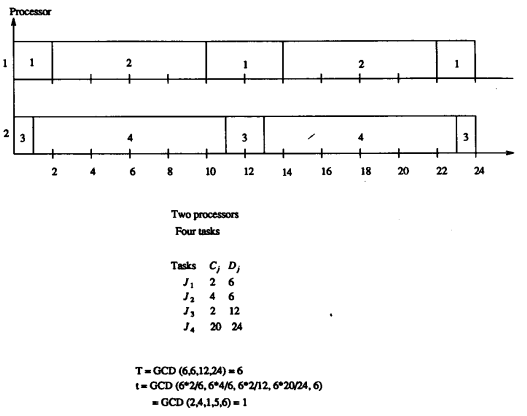
*Fig. 3.7. – Cazul F(k, j+1) ≥ F(k+1, i)*

### Task-uri periodice

Pentru a vorbi despre task-urile periodice, va trebui să introducem o nouă noțiune: factorul de utilizare (*U*). Factorul de utilizare al unui set de task-uri este egal cu . Un task poate solicita un nou timp de procesare de îndată ce deadline-ul acestuia expiră. Un aspect evident este reprezentat de faptul că reprezintă o condiție necesară pentru ordonarea unui set de task-uri, având la dispoziție *n* procesoare. Posibilitatea ca ar reprezenta, de asemenea, și o condiție suficientă pentru a se putea realiza o ordonare fezabilă reprezintă o problemă discutabilă. Vom încerca să dăm un răspuns parțial prin următoarea teoremă:

*Teoremă: „*Presupunem un set de *m* task-uri periodice având factorul de utilizare . Notăm cu si . Atunci, o condiție suficientă pentru a putea ordona având la dispoziție *n* procesoare este ca *t* să fie număr întreg.*”*

Pentru a demonstra această teoremă vom presupune că *t* este un număr întreg. Acest lucru ar rezulta în faptul că vor fi toate numere întregi. Strategia noastră este să executam taskul 1 pentru unități de timp, taskul 2 pentru unități de timp și așa mai departe. Pentru un task *j*, fiecare cerere va solicita un număr de *Cj* unități de timp și care va trebui să se execute in *Dj* unități de timp. Putem diviza *Dj* în părți, având fiecare dimensiunea T. Pentru fiecare parte vom aloca un număr de unități de procesare pentru taskul *j*. Așadar, pentru fiecare task în parte, timpul total de procesare va fi dat de urmarea formulă:. De aici putem deduce faptul că timpul total de procesare al întregului set va fi . Un astfel de exemplu este ilustrat în următoarea figura:



*Fig. 3.8. – Scenariu cu 4 taskuri periodice*

1. <https://www.mcvuk.com/business-news/over-1000-people-made-gta-v/> [↑](#footnote-ref-1)
2. Sonke Hartmann, Dirk Briskorn, “An updated survey of variants and extensions of the resource-constrained project scheduling problem”, European Journal of Operational Research, 2021 [↑](#footnote-ref-2)
3. Carlos Montoya, Odile Bellenguez-Morineau, Eric Pinson & David Rivreau, “Handbook on Project Management and Scheduling Vol. 1”, Springer, 2014 [↑](#footnote-ref-3)
4. Sonke Hartmann, “Project scheduling with resource capacities and requests varying with time: a case study”, Springer, 2013 [↑](#footnote-ref-4)
5. Christian Artigues, Sophie Demassey & Emmanuel Neron, “Resource-Constrained Project Scheduling: *Models, Algorithms, Extension and Applications*”, ISTE, 2008 [↑](#footnote-ref-5)
6. Resource-Constrained Project Scheduling Problem [↑](#footnote-ref-6)
7. M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," in IEEE Transactions on Software Engineering, vol. 15, no. 12, Dec. 1989 [↑](#footnote-ref-7)
8. M. Dertouzos, “Control robotics: The procedural control of physical processes”, Proc. IFIP Cong., 1974, pag. 807-813 [↑](#footnote-ref-8)