**Skedulimi i CPU, Sinkronizimi, Deadlocks**

*Burst Time në CPU* paraqet frekuencën maksimale që cpu mud ti dedikoj një procesi.

*CPU Scheduler* e bën përzgjedhien e procesit i cili do ekzekutohen e i cili gjindet në *ready queue*.

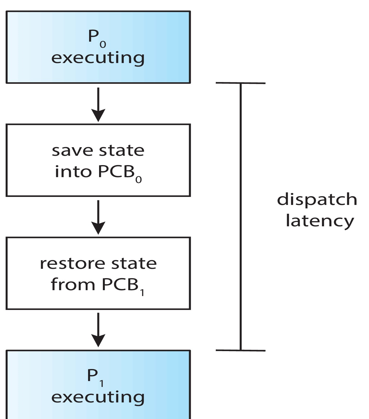
CPU Scheduling merr vendime kur procesi kalon nga : 1.*Running në waiting, 2.Running në ready, 3.Waiting në ready, 4.kur terminohet.* Në rastin 1 dhe 4 nuk ka nevoj të mirret ndonjë vendim, ndërsa në 2 dhe 3 mirren vendime.

Scheduling ndahet në *preemitive dhe non-preemitive*. *Preemitive* – na jep kontroll të plotë mbi procesin, duke e mundësuar edhe context switching. *Non-preemtive* – nuk e lejon context switching deri sa procesi vet të terminohet ose të kaloj në gjendje tjetër përveq running.

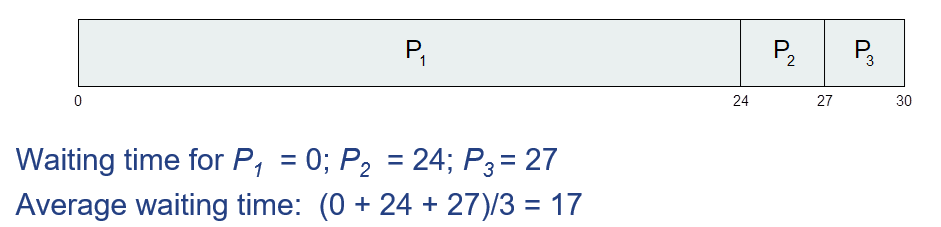
*Preemitive* mund të shkaktoj race-condition.

*Dispatcher* bën alokimin e CPU për processet që zgjedhen nga *scheduling. Dispatcher* përfshihet/përdoret kur ndodhë context switching, switch to user mode.

*Dispatcher latency* është vonesa që shkaktohet gjatë kohës kur ndodhë context switching, kjo përshkak të vonesës në ruajtien dhe restartimin e PCB, gjat ksaj kohe *dispatcher* nuk ka proces tek i cili mund të bëj alokimin.



Për të përzgjedhur algoritmin e scheduling duhet përdorur kriteret e cekura në vijim : *CPU Utalization* – Mbajtia e cpu të zënë gjat gjith kohës. *Throughput* – Numri i proceseve që mund të ekzekutohen për njësi kohe. *Turnaround* – Koha që nevojitet për ekzekutimin e një procesi. *Waiting time* – Koha për të cilën procesi gjindet në *ready queue*. *Response time* – Koha që shpezohet nga request i parë deri në respone të parë.

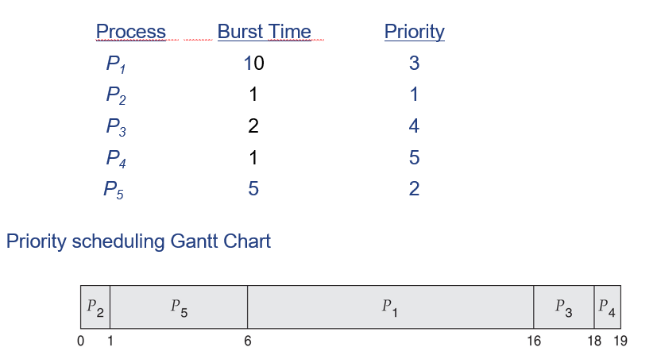
*First Come First Served* : Algoritëm për scheduling, procesi i parë që gjindet më *ready queue* është procesi i parë që do të ekzekutohet, ky algoritëm është non-preemtive. Në rastin më të keq është shum i ngadaltë. 

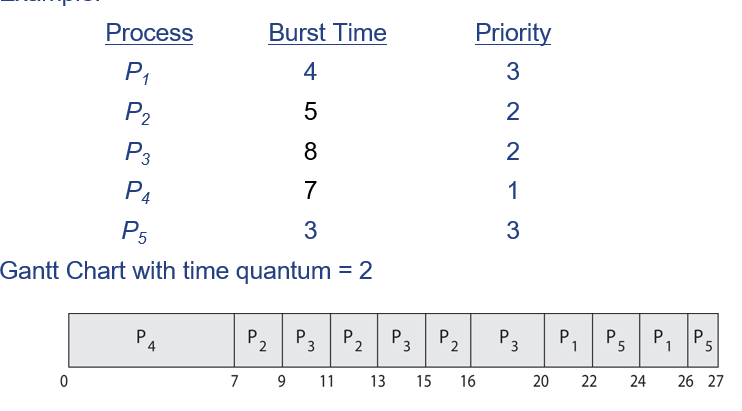
*Convoy Effect* : Do të thotë se processet e shkurta gjenden pas procesve të gjata.

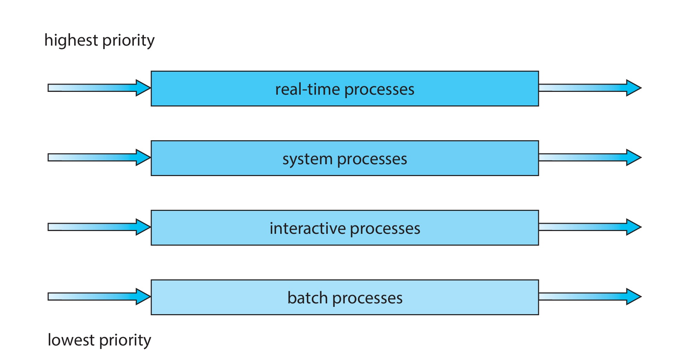
*Shortest Job First :* Nga ready queue zgjedhet procesi i cili kërkon më së pakti koh për ekzekutim.

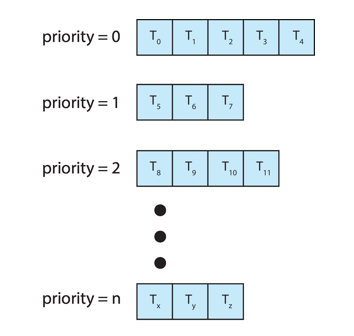
*Shortest Remain Time First* : Është si SJF veqëse është preemtive, le të themi se algoritmi ka zgjedhur processin më të shkurtë nga ready queue dhe ka filluar ta ekekutoj mirpo në ato qaste vjen një process që kërkon më pak kohë, kjo do bëj që algortimi të kërkoj context switching.

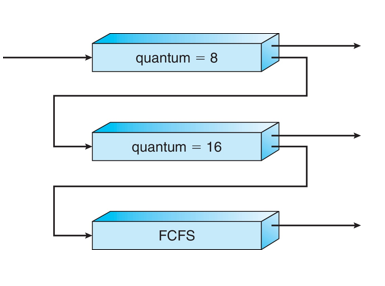
*Round Rabin* – Secilit proces i caktohet nga një njësi kohore për ta përdorur CPU e cila është e njëtjtë për të gjitha processet kjo njesi kohe quhet *quantum time (q)*. Pasi të përfundoj *q* procesi vendoset në ready queue dhe vazhdon ekzukutimi i procesit tjetër për *q* kohë, nëse kemi n procese në ready queue dhe *q-quantum time*, secili proces do përdor 1/n të kohës së cpu dhe secili proces do pret në ready queue asnjëher më shumë se (n-1)\*q. *RR* në krahasim me *SJF* ka response më të shpejt mirpo *turnaround* më të madh.

*Priority scheduling* : Secilit proces i asocohet nga një integer i cili e reprezenton prioritetin e ati procesi, numri më i vogël simbolizon prioritet më të lartë, numri më i madh simbolizon priotet më të ulët. Nga queue lista përzgjedhen processet me prioritetin më të lartë. Problem që mund të shfaqet me këtë algoritëm është *stravation* – ku proceset me prioritet më të ultë mund të mos ekzekutohen kurrë. Zgjedhia e kti problemi është *agging*- Ku processeve i u rritet prioriteti mbas kohës në kohë.

Kombinimi i *Priority scheduling* me *RR* është logjika e njëjtë me atë të Priority scheduling, mirpo në rast se proceset kan prioritet të njëjtë në to aplikohet *RR*.

*Multilevel queue :* krijohen disa queue lista ku kategorizohen proceset sipas pririteteve.



Kur përdoret Multilevel Queue duhet pasur parasysh këto parametra : *Numri i ready queue, Algoritmi për secilin queue, metoda që përdoret për ta rritur dhe ulur prioritetin e procesit*.

*P-Processi, Gjendet në Q1-ready queue, Q1 e përdor alogoritmin RR, nëse për 8ms nuk terminohet atëher, P kalon në Q2 e cili po ashtu përdor RR aty ekzekutohen për 16ms, nësë përsëri nuk terminohet kalon në Q3 ku përdoret FCFS aty nuk del deri sa të terminohet.*

Proceset mund të ekzekutohen në mënyr konkurrente gjithashtu mund të ndërpriten nga qasti në qast. Ekzekutimi dhe qasja konkurrente në burimet e përbashkëta mund të shkakoj mospërputhje të të dhënave. Për ta menjanuar kët problem duhet ofruar mekanizëm i cili e rendit ekzekutimin e cooperating proceseve.

*Sinkronizimi* bëhet kur duam të koordinojmë proceset dhe të evitojmë konfliktet, për të bër *sinkronizim* duhet të ekzistojn komunikim në mes procesve.

*Sinkronizimi* përdoret për të kontrolluar qasjen në : *sisteme të vogla multi-pocsuese*, *sisteme kompjuterike multi-procesuese ose multi-threde*, *sisteme distibuive* si *databaza, sisteme bankare, web servera*.

Secili proces në kodin e vet e ka një pjes të kodit e cila quhet *critical section* në kët seksion procesi mund të jet duke ndryshuar ndonjë variabël, rishkruar ndonjë tabel, shkruar ndonjë fajll...

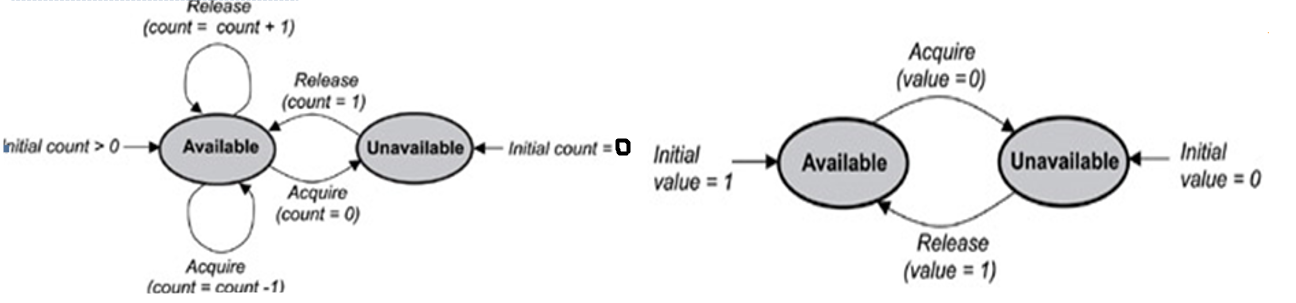
Për të zgjedhur problemin e *critical section* duhet përmbushur tri kërkesa : *1.Përjashtimin reciprok* : Kur një proces është duke e ekzekutuat *cs* asnjë proces tjetër nuk mund ta ekzekutoj *cs*. *2.Progresin :* Nëse asnjë proces nuk është duke e ekzekutuar *cs* menjëher duhet futur një process tjetër për ta ekzekutuar *cs* përshkak se është i rëndësishëm, pra nuk mundet dy procese ta ekzekutojn përnjëher *cs* por nuk bën që asnjë proces mos të jet duke ekzekutuar *cs*. *3.Pritja e kufizuar* : Limiti i numrit që një proces mund të futet në seksionin kritik, pasi që procesi ka bër kërkesën dhe ka pranuar aprovimin llogaritet një futje në seksionin kritik.

*Sinkronizimi mund të jet harduerik ose softuerik.*

*Sinkronizimi harduerik* : Është i bazuar në *locks*. Në sistemet me një cpu përdoret metoda me *interrupts*, e cila ka kët rrjedh hapash *procesi futet në cs dhe i ndalon interrupts, asnjë process nuk mund ta ndërprej deri sa del nga cs kur edhe i lejon interrupts*. Në sistemet me shumë cpu përdoren *instruksionet harduerike atomike* non-interrupted, këtu hyn 4 funksione *test&set, compare&swap* – Janë funksione të pa zbërthyeshme, të cilat mundëson sinkronizimin dhe përjashtimin reciprok. Mbahet një list e kërkesave të procesve për operacione, kto kërkesa krahasohen me resurset në dispozicion nëse ka përputhje lejohet qasja në të kundërtën nuk lejohet.

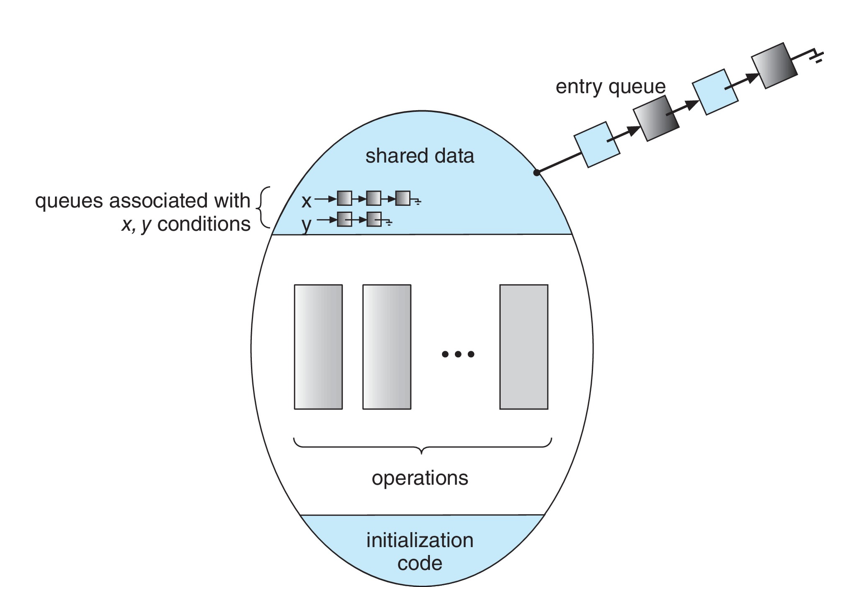
*Sinkronizimi Softuerik* : *Mutex* : Është objekt i cili krijohet me fillimin e ekzekutimit të procesit, ky objekt lejon që thredat të ken qasje në të dhënat e përbashkëta, mirpo jo në të njëjtën kohë. Është i bazuar në *locks* kur thredi dëshiron ti qaset resurseve e bën *lock mutex-in* duke i bër të pa mundur thredave tjer ta marin atë dhe të ken qasje në të dhëna, pasi ta përfundoj ekzekutimin e liron *mutex-in*. Sistemi operativ ofron dy metoda për mutex *aquire() & relase()*. *Semafori* : Përdoret për të sinjalizuar thredat të cilët mund të përdorin resurset. Përdor *tokena* me anë të të cilve e kufizon numrin e thredave që mund të ken qasje në resurse, kur thredit i ipet qasje aj e zvoglon numrin e *token-ave* kur e kryen ekzekutimin e bën relase e rrit numrin e *toke-nave*. Ekzistojn dy lloje të *semaforave* semafori binar dhe semafori numrues. Mutex janë shumë të ngjajshëm me *semaforin binar*. Semafori ka dy funksione kryesore *wait(s)* bën që numri i tokenave *s* të zvoglohet për një, duke ja dhën tokenin thredit që e ka thirrur. Nëse *s* ëhtë 0 procesi pret deri sa të jet thirrur *signal* nga një proces i cili ka token në posedim. *Signal(s)* thirrja e kti funksioni e rrit numrin e tokenave *s* për një gjithashtu larjmron prceset që janë duke fjetur.

Dallimi në mes *mutex* dhe *semaforit* : qëndron në atë se *mutex* mundëson qasjen në resurset e përbashkëta vetëm një thredi i cili përdor metodën *acquire()*, semaforat në tjetrën anë *semaforat* mundësojnë që shumë threda me anë të tokenave të ju qasjen resurseve të përbashkëta.



*Monitoret* janë koncept që mundësojnë thredave qasje në resurset e përbashkët përmes përjashtimeve reciproke dhe ofron mundësi bllokimi deri sa një kusht plotësohet. *Monitori* është klas që e mbështjell seksionin kritik.

*Monitori* me variabla të kushtëzuara *x,y* ofron dy funksione, *x.wait()* – Pret derisa një process ta ket thirrur *x.signal()* i cili kthen ndonjë proces i cili gjendet në *wait()* nëse nuk ka të till nuk bën asgjë.

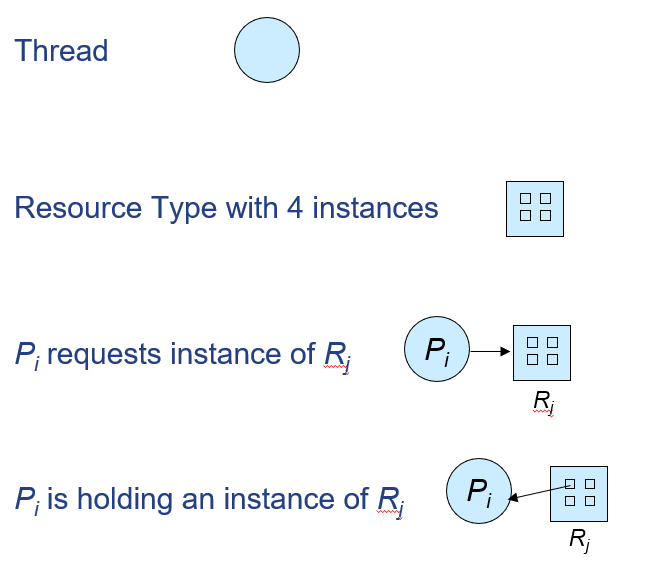


Dallimi në mes *monitorit* dhe *semaforit* : *Semafori* i lejon më shumë procese të futen në ekzekutim nëse numri i *token-ave* është më i madh se 0. Ndërsa me *monitor* vetëm një procesi iu ipet qasje në resurse. Në *semafora* kur thirret funksioni *signal* rritet numri i tokenave, ndërsa në *monitor* nuk ka efekt në variablat kushtëzuese.

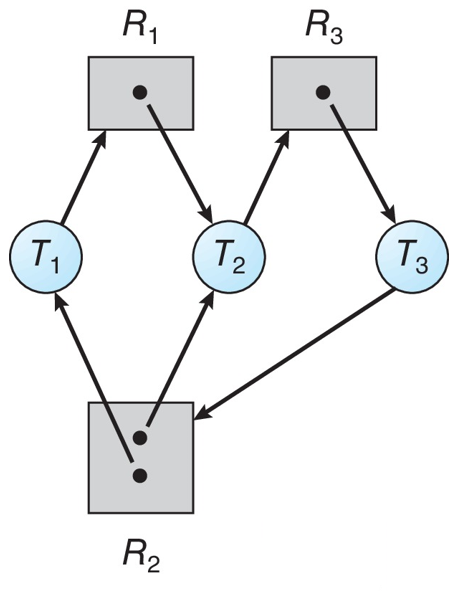
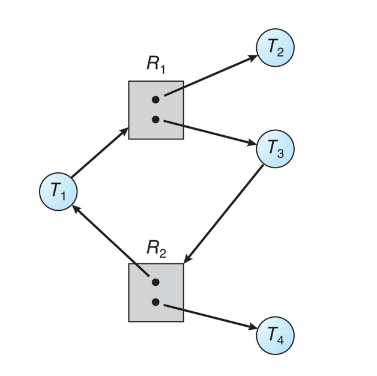
Sistemi ka resurse secili resurs ka instanca të cilat përdoren nga proceset me logjikën *request, use, relase*.

*Deadlock* është pritje e proceseve për njëri tjetrin në pa fundësi. Për të ndodhur *deadlock* duhet të plotësohen këto 4 kushte : *Mutual Excluzive* : Vetëm një process mund të qaset në një reurs të përbashkët në të njëjtën kohë. *Hold and Wait* – Një proces ka një resurs të rezervuar por në të njëjtën kohë është duke pritur për një resurs tjetër i cili është i rezervuar nga një proces tjetër. *Circular Wait* – Le të themi se kemi N procese nga P0,P1,P2,..,PN, nësë P0 pret për P1 ky pret për P2 e kshtu me rradhë deri tek PN , e PN prejt për P0, kemi të bëjm me *circular wait*.

*Resource Allocation Graph* :



*Graph me deadlock :*  *Graph pa deadlock* :



Nëse nuk ka cikle në graf atëher nuk ka as *deadlock*, nësë ka cikle dhe nuk ka instanca të mjaftueshme ndodh *deadlock*, por nëse ka instanca të mjaftueshme nuk ka *deadlock*.

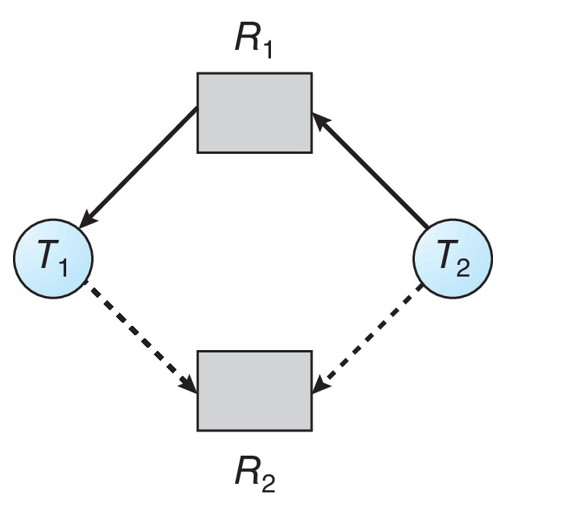
Për ta trajtuar problemin me *deadlock* kemi 3-4 metoda kryesore : *Preventation* – Parandalimi i deadlock. *Avoidance* – Menjanimi i deadlock. *Të lejohen deadlock nga sistemi* e pastaj të bëhet recover. *Të injorohen* dhe të sille sikurse nuk ndodhë asnjëher deadlock.

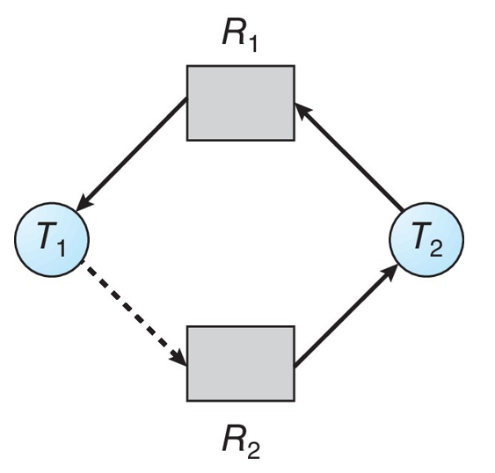
Për të *prevent* (parandalu) *deadlock* duhet të mos plotësohet njëri nga kushtet e ekzistimit të deadlock : *Mutual Excluzive* : Të aplikohet vetëm në të resurset non-shared, ndërsa në resurset *read-only* mos të aplikohet. *Hold and Wait* : Të sigurohet se procesi që kërkon resurse mos të ket të rezervuara resurse tjera. *Circular Wait* – Nëse procesi ka disa resurse të rezervuara dhe bën një request edhe për disa resurse tjera, mirpo ky request refuzohte atëher procesi duhe ti liroj të gjitha resurset që i ka në posedim.

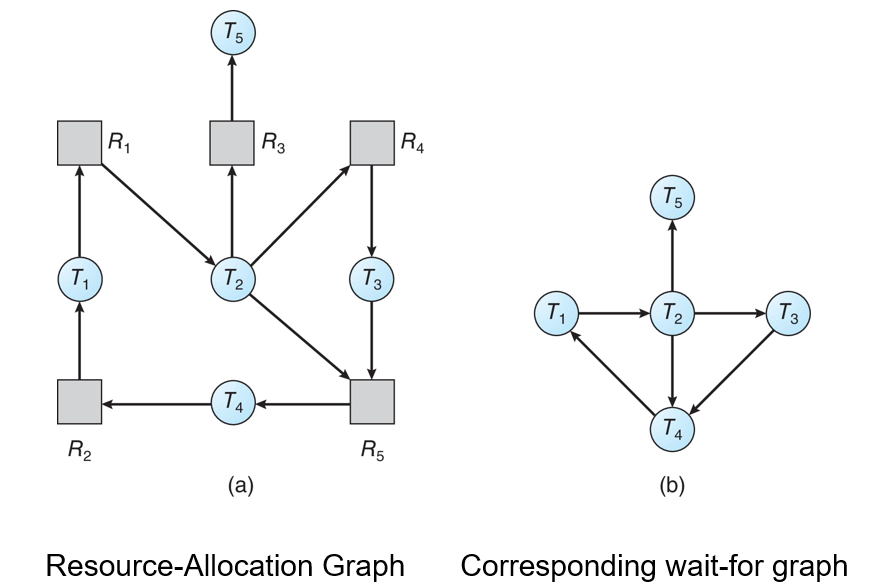
*Deadlock Avoidance* ose menjanimi i deadlock-ut kërkon informata shtesë, informata kto që kanë të bëjn me proceset që janë në *read queue* qfarë kërkesa kanë dhe sa resurse ekzistojnë për ti shërbyr kto procese. Kto informacione krahasohen dhe na mundësojn të kuptojmë se kur gjindemi në *safe state* e kur jo, safe state – jemi kur kemi numër të resurseve më të madhë se numri i kërkesave apo proceseve. Nëse numri i instancave për një resursë është më i vogël se numri i kërkesave ai resurs nuk u shërbehet proceseve.

Me këtë kuptojmë se *avoidance* e parandalon kalimin në *unsafe state* kët e bën duke përdorur dy algoritme : *Single instance of resource* : Përdor resource allocation graph të modifikuar. *Multiple instance of resource* : Përdorë Banker Algoritmin.

Më poshtë është i paraqitur një graf i resurseve të alokuara i modifikuar, shigjetat e ndërprea tregojn se T1 dhe T2 mund të rezervojnë R2 mirpo nuk është e sigurt kjo nuk na dërgon në *unsafe state,* ndërsa në të djathtë është shfaqur rasti kur kemi *unsafe state*. Shigjeta request kthete në claim kur lirohet resursi, ndërsa claim kthehet në request kur rezervohet resursi.





Detektimi i deadlock bëhet në dy forma : *Resource allocationg graph* dhe *Wait for Graph*. 

Nëse nuk arrijm të bëjm as parandalim e as menjanim të *deadlock* atëher është e nevojshme të kryejm një *recovery* prej *deadlock*, kjo mund të bëhet në dy mënyra : *Abort all* : Të terminohen të gjitha proceset që janë në deadlock. *Abort one by one until deadlock ends* : Terminohen proceset nga një nga një deri sa të largohet deadlocku, zgjedhja e procese për terminim bëhet në bazë të prioriteti të tyre. Për të prioritizuar proceset përdorim *resource preemitve* së pari duke e përzgjedhur *viktimën* – zakonisht procesi me prioritetin më të ultë. *Rollback* – kur kthehemi në safe state i rialokojm resurset proceseve të terminuara. *Starvation* – mund të shfaqet nësë gjithmon terminohet procesi i njëjtë pra *viktima* e njëjtë edhe pas disa *rollbacks*.