Scavenger jobs operator

Cieľ

- vyrobiť Custom Resource Definition (CRD) *ScavengerJob* a k nemu controller, ktoré budú spolu slúžiť na zvyšovanie využitia zdrojov clustru pomocou Operator Pattern (https://kubernetes.io/docs/concepts/extend-kubernetes/operator/)
- black-box prístup - operátor nerieši, čo beží
- vytvoriť prototyp aplikácie, ktorá sa dá spustiť cez *ScavngerJob* (GROMACS, AMBER, PYTORCH/TENSORFLOW - tu nejaký malý mám)
- ide o prototypové no **plne funkčné** riešenie, počíta sa s ďalším vývojom a úpravami

Technológie:

- https://tag-app-delivery.cncf.io/whitepapers/operator/
- odporúčam použiť Operator SDK https://operatorframework.io pretože vygeneruje celú kostru
- intro do Operator pattern od IBM https://developer.ibm.com/articles/introduction-to-kubernetes-operators/ , krátky tutorial na memcached operátor https://developer.ibm.com/learningpaths/kubernetes-operators/develop-deploy-simple-operator/create-operator/ (fajn si prejsť na pochopenie tvorenia CRD Spec, Status a controller reconcile() loop)
- https://cloud.redhat.com/blog/kubernetes-operators-best-practices
- https://github.com/operator-framework/operator-sdk/issues/6035 … ako spracovávať vŠetky CRD naraz
- https://omerxx.com/k8s-controllers/ good to know
- kľudne aj iné zdroje
- potrebné si naštudovať Priory Classes a pod preemption: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/

Motivácia:

Cloudové prostredia sú notoricky známe tým, že trpia nízkou vyťaženosťou oproti rezerváciam. Scavenger Job (ďalej už len SJ) je job, ktorý slúži na zvýšenie utilizácie zdrojov clustru tým, že konzumuje voľné zdroje (ktoré ale môžu byť rezervované, no sú aktuálne nepoužívané), no ak príde do clustru požiadavka na použitie týchto rezervovaných zdrojov od "vlastníka" rezervácie, (pre nás abstrakcia "akýkoľvek workload, ktorý má vyššiu prioritu než SJ"), tak je SJ prerušený, uvoľní zdroje a tie si vezme workload s vyššou prioritou. Všetky SJ majú nižšiu prioritu než všetky iné druhy workloadov v clustri.
SJ môžu byť aj multipodove = jeden SJ môže chcieť spustiť Joby vo viacerých rôznych podoch a potencialne viacerych nodoch zároveň. Ak je takýto SJ prerušený, všetky jeho Joby musia byť zrušené.

SJ pomáhajú s využítím clustru a neplytvaním zdrojmi no aj samotne SJ môžu trpieť niektorými týmito problémami :

1. môžu plytvať zdroje, ak SJ by teoreticky dobehol o 3 minúty a už beží 2 hodiny a my ho ukončíme -> zbytočné prepočítaný CPU čas, úloha sa musí počítať znova
2. môžu plytvať zdroje ak si SJ požiadajú o zdroje, ktoré nevyužijú z väčšej časti -> rovnaký problém nevyužitia zdrojov

Nasledujúce komponenty budú potrebné, no 4tá je už additional.

Komponenta 1: Operátor

Návrh CRD:

CRD *ScavengerJob*:

ScavengerJob bude iba abstrakcia, reálne bude kontrolerom spúšťaný klasicky *kind: Job*, ktorý preberie informácie zo specu SJ (a Job spúšťa samozrejme Pod). Job spustený jedným SJ musí mať ownerRef na SJ, prípadne inú jasnú viazanosť na to, ku ktorému SJ patrí.

Spec musí obsahovať:

3. image, ktorý bude použitý jobom
4. požiadavky na zdroje (pripraviť na requests i limits, používať sa naozaj budú guaranteed pody, ktoré majú req==lim)

5. referenciu na PVC, CM, Secrets ktoré sa majú použiť atď (môže ich byť aj viac, može to byť štrukturované ako pre normálne Joby čiže meno objektu a mountPath kde ho pripojiť)
6. command na spustenie
7. či ide o multikontajnerovú úlohu, ak áno koľko kontajnerov (resourcz su rovnaké pre každý kontajner) (melo by vyplnit spise MPIJob kind)
8. runAsUser
9. ďalšie sa možno vyskytnú

Status musí obsahovať:

10. informáciu o stave SJ v rámci clustru: čaká, running, prerušený, completed, failing
11. informáciu o tom, koľko krát bol prerušený *interruptedCount*
12. creation Timestamp (to je možné, že je by default vygenerované v rámci CRD)
13. interruptedTimestamp: možno pole? značiace kedy bol SJ prerušený?
14. runningTimestamp: možno pole? značiace kedy bol SJ začatý (pole preto bo može byť prerušený a potom musí začať znova dakedy)?
15. finishedTimestamp (to je možné, že je by default vygenerované v rámci CRD)
16. priorityStartOverTimestamp: značí, kedy dobehlo posledných duration/*interruptionAfter* minutes (kvoli jednoduchšiemu počítaniu + 30m)
17. voliteľný message keby náhodou

Tieto informácie by mohli byť i v nejakej DB z ktorej by kontroler čítal, no riešenie chceme cloud-native (meaning?). Naviac, použitie CRD je vhodné aj vzhľadom na to, že tieto objekty môžu byť potom uvažované ako ďalší aktéri clustru či použité do metrík rovno z clustru.

Aby sme vyriešili bod 1. s plytvaním zdrojov:

18. implementujeme hodnotu *duration (interruptionAfter, ap.)*, ktorá značí dobu z ktorej sa počíta vhodnosť na preempciu
19. požadujeme od workloadov, ktoré chcú bežat ako SJ aby implementovali checkpointy (v nejakej forme) a aby po reštarte tej istej úlohy neskôr sa vedel workload reštartovať od daného checkpointu. Tieto checkpointy by sa mali diať každých x<*duration* aby sa zaručilo, že daný čas nebol prepočítaný zbytočne a prípadne ukončený workload môže pokračovať
20. S plynutím tejto doby sa zvyšuje “dopočítanosť” workloadu a teda je menej a menej vhodný na ukončenie. Keďže môže bežať viac SJ (Jobov ale v skutočnosti Podov) zároveň a môžu byť v rôznej fázi rozpočítanosti, je potrebné aby bežiace SJ (Joby ale v skutočnosti Pody) mali medzi sebou tiež nejaké priroty aby kube-scheduler (default plánovač v K8s) vybral v prípade potreby ten SJ (Job ale v skutočnosti Pod), ktorý je zatiaľ v najnižšej fáze rozpočítanosti
21. Počíta sa s tým, že buď sa MUSÍ zrušiť nejaký workload alebo nie. Netreba teda deliť dobu rozpočítania na kusy ktoré (ne)sú preeemptible lebo buď třeba alebo nie
    a. ak sú v clustri iba SJ s vysokými prioritami a je nutné niečo zrušiť proste sa to musí zrušiť alebo **počkať 5 minút**
22. Ak je SJ prerušený alebo zrušený
    a. Ak končí Job (a teda Pod pod ním) s exit 1 a pod.status.phase-=Failed, SJ sa už nebude považovať ďalej za vhodný pretože v ňom očividne niečo nefunguje a nastaví sa celý SJ ako Failed alebo čosi také
    b. ak bol Pod prerušený z von (čiže na pod s pod.status.phase=Running sa aplikovala operácia DELETE) tak sa proste zrušia jeho Job(y) no SJ neprestane existovať, je iba označený ako prerušený a navýši sa mu *interruptedCount*
        i. *interruptedCount* určuje prioritu medzi čakajúcimi SJ. Ak sa znova uvoľní miesto a nejaký SJ môže bežať, najprv sa spustia (ak je miesto) SJ ktoré majú najvyšší *interruptedCount*. (Prístup “better finish what we started than start something else”). Ak nie je miesto na žiadny z už začatých a prerušených SJ, spustí sa nejaký čo ešte nebežal a je na nenho dostatok zdrojov

Aby sme vyriešili bod 2. s plytvaním zdrojov:
Dajú sa použiť tunery alebo ML techniky nastavenia vhodných zdrojov pre SJ. To nie je zatiaľ scope of this work.

Cca návrh controlleru (reconcile loop):

23. Reconcile by ~~snáď mohlo vrátiť nie 1 SJ per time instant ale celý list všetkých objektov typu *ScavengerJob* aby sa z nich dalo vyberať, ktorý bude vykonaný~~ ~~by mal byť triggered na základe zmeny Podu~~ mal byť triggered na
    a. Základe updatu z Jobu owned by some SJ (ktorý bol spôsobený zmenou Podu)
    b. Periodicky ak je SJ vo fáze Running a teda je potrebné mu zvyšovať running prioritu

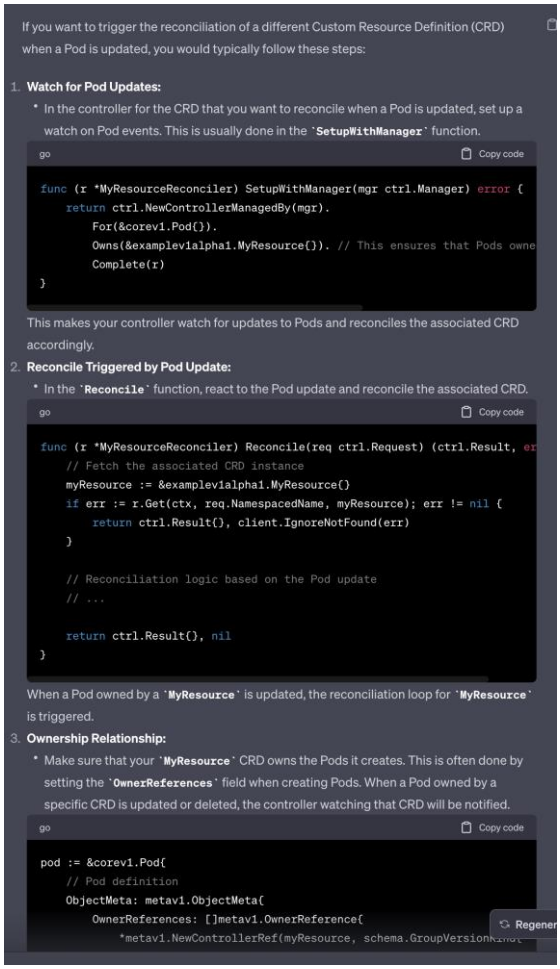
24. Keď sa vytvorí novy SJ tak
    a. DUMB: FIFO, proste pop z queue a prvý sa spustí

b. LEPŠIE:
   i. Implementovať default workingQueue jako priroty Queue, implementovať korešpondujúci enqueue a dequeue, ukážkový kód (thx ChatGPT) na konci dokumentu
   ii. Dequeue: SJ na spustenie sa vyberajú podľa popísanej priority *interruptedCount* . Vybere sa ten, s najvyšším *interruptedCount*. Ak v heap nie je žiadny s *interruptedCount*, vyberie sa FIFO čiže najstarší, ktorý je v queue. Priority Queue pre položky s rovnakou prirotou neimplementuje žiadny sofistikovanejší mechanizmus no funguje to tak, že sa zachováva poradie v akom prišli položky do queue. To nám vyhovuje, keďže sprcovávme inak položky v poradí v akom prišli a neuprednostňujeme napr. Najstaršie
      1. Príklad: Queue s SJ 9,8,7,6,5,4
      2. Príklad: nový SJ X vytvorený v čase T, nesú na neho zdroje, je requeued o 10 časových jednotiek
      3. Medzitým queue : 2,3 (9,8,7,6,5,4 z bodu 1 sú spracované)
      4. Prišiel čas na SJ X o 10 časových jednotiek čiže queue je 10,2,3
      5. Má nasledovať dequeue. 2,3 sú novšie než 10 (v queue) a majú rovnakú prioritu. Pustí sa 3, neuprednostní sa 10 len preto, lebo je staršie ---> zachováva sa poradie vloženia
   iii. Reconcile loop dostane po zaznamenaní CREATE v clustri ctrl.request v ktorom je namespace ktorý spustil reconcile a objekt ktorý ho spustil.
   iv. treba vytvoriť Job object, pripojiť všetky PVC, Secrets, ConfigMaps atď. Skontrolovať že existujú, su validne hodnoty atd Ak nie, nastaviť SJ na Failed so správou prečo a neskúšať ďalej, ak chyba je napr v neexistujúcom PVC apod.
   v. Dá sa vytvoriť customPopFromWorkQueue, dotaz na ChatGPT "***implement custom logic of popping from work queue"*** , predtym uz mal kontext ohladom CRD a operatorov

c. Job sa tupo vytvorí a to je všetko. Čokoľvek čo sa ďalej stane s Jobom triggerne reconcile (malo by) , napr.
   i. ak Job ostane pending kvôli nedostatku zdrojov spôsobí to reconcile kde sa zistí dôvod (nedostatok zdrojov) a SJ ostane v stave Pending. Reconcile na taký SJ napr s return ctrl.Result(RequeueAfter: time), nil Zároveň třeba zmazať vytvorený Job pretože bude visieť v systéme a čakať na zdroje kým nebudú dostupné a to nechceme. SJ sa nezmaže, ten ostane Pending a keď na neho znova príde rad, spustí sa Job, ktorý potom triggerne reconcile atď
      1. Otázka - má mať SJ, na ktorý neboli zdroje časom vyššiu prioritu? Prevent starving. Alebo je to nám jedno a má si to riešiť ten, kto ich submitol že jako dlho kvasia v queue a že možno žiadajú príliš veľa zdrojov. Alebo ich my requeue náhodne po nejakom čase a možno budú vtedy potrebné zdroje dostupné? ... -> **do začiatku stačí že sa takýto Job reconcile(requeuAfter: nejaký čas)**
   ii. Ak Job zfailuje, nastaviť SJ jako failed a už na ňom neoribť žiadne akcie (check v controlleri ci SJ.status == failed a ak hej rovno return)
   iii. Ak Job Succeded, podobne jak fail, nič s nim už nerobiť
   iv. Ak Job Running, tak pre SJ platí že sa im musí cyklicky meniť *runningPriority - dôležité je, že ak nastane čas* priorityStartOverTimestamp, musí byť na Job priradená preemptible priorita

d. Funkcie na zaregistrovanie eventu pre pody
   i. SetupWithManager to watch for Job/Pod updates .. kedze pod je spusteny jobom a job moze mat owner reference na SJ .. Job.. ScavengerJob
   ii. EnqueueRequestsFromMapFunc https://pkg.go.dev/sigs.k8s.io/controller-runtime/pkg/handler#EnqueueRequestsFromMapFunc viac komplikované predikáty
   iii. https://sdk.operatorframework.io/docs/building-operators/golang/references/event-filtering/ možno budeme potrebovať toto

e. AKO MNOŽSTVO VOĽNÝCH ZDROJOV SA BERIE DO ÚVAHY THRESHOLD=70% CELKOVO REÁLNYCH ZDROJOV CLUSTRU ABY OSTALO NEJAKÉ VOĽNÉ MIESTO DO ISTOTY.
   i. threshold sa dá tunovať podľa výsledkov, heuristikou sa možno príde na vhodnú hdnotu pre nás a naše zloženie workloadov (a im podobné setupy)
   ii. otázka je, že či sa má počítať aktuálny usage a z toho 70% alebo aktuálny usage plus SJ ktorý sa uvažuje a z toho 70%



If you want to trigger the reconciliation of a different Custom Resource Definition (CRD) when a Pod is updated, you would typically follow these steps:

1. **Watch for Pod Updates:**
   * In the controller for the CRD that you want to reconcile when a Pod is updated, set up a watch on Pod events. This is usually done in the `SetupWithManager` function.

```go
func (r *MyResourceReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&corev1.Pod{}).
        Owns(&examplev1alpha1.MyResource{}). // This ensures that Pods owne
        Complete(r)
}
```

   This makes your controller watch for updates to Pods and reconciles the associated CRD accordingly.
2. **Reconcile Triggered by Pod Update:**
   * In the `Reconcile` function, react to the Pod update and reconcile the associated CRD.

```go
func (r *MyResourceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, er
    // Fetch the associated CRD instance
    myResource := &examplev1alpha1.MyResource{}
    if err := r.Get(ctx, req.NamespacedName, myResource); err != nil {
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // Reconciliation logic based on the Pod update
    // ...

    return ctrl.Result{}, nil
}
```

   When a Pod owned by a `MyResource` is updated, the reconciliation loop for `MyResource` is triggered.
3. **Ownership Relationship:**
   * Make sure that your `MyResource` CRD owns the Pods it creates. This is often done by setting the `OwnerReferences` field when creating Pods. When a Pod owned by a specific CRD is updated or deleted, the controller watching that CRD will be notified.

```go
pod := &corev1.Pod{
    // Pod definition
    ObjectMeta: metav1.ObjectMeta{
        OwnerReferences: []metav1.OwnerReference{
            *metav1.NewControllerRef(myResource, schema.GroupVersionKind{
```

Poznámky:
Od bodu existencie SJ v clustri musia mať VŠETKY workloady prioritu napr *ordinaryWorkloadPriority*. Vytvoriť webhook ktorý nastaví dostatočne vysoký default všetkým ostatným (dostatočne vysoký = aby ostalo dosť čísel reprezentujúcich priority SJ)

Ak sa má ScavengerJob skontrolovať na validitu, musí byť pred ním ValidatingAdmissionConfiguration/Webhook. Po prijatí do clustru už nie je možne odstrániť objekt.

Komponenta 2: Regulátor práhu 70-80-85%

Jako stanoviť vhodný práh?

Sčasti inšpirované článkom https://dl.acm.org/doi/pdf/10.1145/3357223.3362734

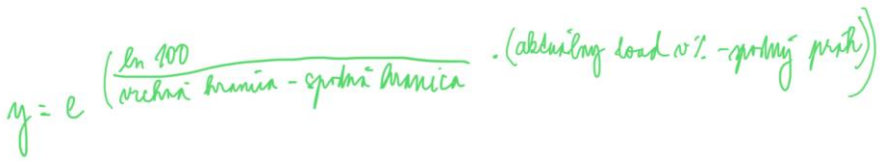
Práh môže byť určený štandardnou odchýlkou priemerného využitia clustru plus nejaká rezerva naviac. Použitie štandardnej odchýlky umožňuje miesto na manévrovanie spadajúce do sledovaného trendu. Ak by clustru poskočila spotreba, skok by sa mal nachádzať v rozmedzí priemerná spotreba clustru +- stdev

Priemerné využitie clustru (CPU) a stdev môžu byť počítané napr denne (prídavná komponenta zas?)
Rezerva je konfigurovateľný parameter

Celkové využitie clustru, ktoré chceme dosiahnuť , spodná hranica (v percentách): 100 - (rezerva + stdev)
Celkové využitie clustru, ktoré chceme dosiahnuť , vrchná hranica (v percentách): 100 - (rezerva /3 + stdev)

Pre interval medzi spodnou a vrchnou hranicou platí, že ide o špekulatívnu preempciu. Controller prestane vyrábať Joby ak presiahne aktuálne reálne využitie clustru (spodný práh+ max(rezerva, stdev) - min(rezerva, stdev))

Percento SJ (j teda P), ktoré majú byť preemtible v intervale medzi vrchnou a spodnou hranicou počítané jako



Prídavná komponenta regulujúca 70% využitie clustru

Controller nepustí nový SJ ak by mal presahovať kapacitu 70% (alebo kapacitu prepočítanú podľa predchádzajúceho). Na druhú stranu, workloady do clustru sa spúšťajú aj mimo SJ framework a teda je potrebné aby niekto udržiaval kapacitu clustru naplnenú na 70% (existujúce workloady plus SJ) ešte "zvonku". Ak kapacita naplnenia presiahne threshold, tak komponenta začne najprv praktizovať *speculative preemption a až od istého prahu* rušiť SJ od najnižšej priority až kým nedosiahne pôvodný, začiatočny treshold , príp kým už neexistujú žiadne preemptible SJ.

Komponenta sleduje práh X a balansuje ho mierkou toho, koľko % z bežiacich scavenger jobov má byť v preeemptible class.

PseudoAlg

```
cluster_allocation = getClusterAllocation()
Initial_threshold = 70
preemptible_percetage = 33
preemptible_percetage_to_be = (cluster_allocation mod initial_threshold) + preemptible_percetage

if cluster_allocation >= 85% overall_capacity:
        While cluster_allocation > Initial_threshold:
                Get SJ of lowest priority
        If no job:
                  Return "sorry nothing to delete for you, make space yourself"
        Delete SJ of lowest priority
        cluster_allocation = getClusterAllocation()
Else:
Publish preemptible_percetage_to_be for prepočítavadlo priorít
```

Komponenta 3: Priraďovanie priorít pre bežiace SJ

Priraďovanie priorít pre bežiace SJ (teda Jobov teda Podov)

Nie je to zodpovednosť reconcile loopu ktorý by robil mnoho vecí naraz. Priorita pre Pody bežiacich SJ je rozdelená na 2 preddefinované PriorityClasses - ScavengerNonPreemptible*a ScavengerPreemptible* . Jednotlivé priority sa priradzujú podľa mierky, ktorú publikuje komponenta regulátor práhu ale do začiatku je 33% zo všetkých bežiacich SJ. Vykonáva iba to, že mení priority bežiacim podom ktoré patria pod bežiace SJ.

PseudoAlg

```
preemptible_percetage = 33
preemptible_percetage = Get_published_preemptible_percetage_from_regulator_prahu()
Sj = Get_all_running_SJ()
For each _sj in sj:
Priroty = CPUtime = CPU req * time elapsed from  last checkpoint
Assign priority to SJ
Sort(SJ on priority)
Set Preemptible Priority To All Pods of First X SJ corresponding to preemptible_percetage
Set NonPreemptible Priority To rest
```

Komponenta 4: Plugin do plánovaču ktorý preemptne i non-preemptible SJ

Prídavná komponenta ktorá preemptne i nonpreemptible v prípade že bežný workload s vyššou prioritou trvá na umiestnení na node ale je pending kvôli miestu

Posledný plugin do postFilter extension point kube-scheduleru. Ak by kube-scheduler rozhodol že třeba niečo preemptnúť pre workload na konkrétnom node (alebo list of nodes) a nie sú tam preemptible workloady, tak sa dá uvažovať nad Scavenger jobmi, ktoré sú nastavené jako non preemtpible ale aktuálne majú najnižiu prioritu - dajú sa zrušiť

Asi dobré uvažovať len pre bežné workloady ktoré sú highly interactive a nepočkajú ani chvíľu pretože eventualne sa niektorý z workloadov stane preemptible

Demo Aplikácia

Generická demo aplikácia

vytvoriť nejaký SJ ktorý dostatočne dlho trvá a tvorí checkpointy. je jedno čo, nech to otestuje že funguje framework

GROMACS

vytvoriť DB GromacsToTune ktoré budú obsahovať informácie o GROMACS výpočte, ktorý ďalej poputuje do tunera. Tuner snáď vypočíta vhodnú resource konfiguráciu (req, lim) a s informáciami GromacsToTune (a možno ešte extra, tie ktoré treba do SJ) vyrobiť predpisy ScavengerJobs a submitnúť do clustru.
Submit môže byť do personal namespacu ale na *ScavengerJobs* sa nevzťahuje namespace ani project kvóta. (zatiaľ)
Tento gromacsScavengerJob framework si môže držať zoznam napočítaných konfigurácií pre SJ a ak sa nejaká ne a ne spustiť (napr nedostatok zdrojov), odstrániť ten SJ a skúsiť inú, menšiu.

Vylepšenia na neskôr (hlavná funkcionalita implementovaná a otestovaná):

- ak SJ neimplementuje checkpointy, dá sa uvažovať nad budúcou funkcionalitou K8s *snapshot* kontajneru - zapnúť do alfa verzie, pri interuption vykonať snapshot ——> ako funguje znovuspustenie neviem
- zatiaľ predpokladáme, že dáta sú už na PVC a nemusia tam byť nakopírované ani stianuté (žiadny input filler ap.) —— —> potom možno nepotrebné žiadať lokaciu dát v SJ Specu inú než kam pripojiť PVC?
- metrics endpoint - push metrics ktoré hlásia kedy boli SJ spustené, dokončené, prerušené, koľkokrát, po koľkých minútach od prijatia do clustru sa spustili atď.
- kvóty na ScavengerJobs - čo ak si niekto prerobí všetky workloady do SJ a tým zahltí cluster a budú sa počítať na extra prístupných zdrojoch len jeho workloady? tým by obišiel personal kvótu
- prejsť z blackbox prístupu na white box kde sa sleduje využitie zdrojov SJ a ak to neni moc dobré, nejak poupraviť (napr podľa historickej usage)
- optimalizácia thresholdu:
    - 70% nodu
    - balancovať medzi vačšimi a menšími nodmi

- MPIJob

PriorityQueue in Go
In this example:

- The **SetupWithManager** function sets up the controller with the manager and initializes the priority queue.
- The **PriorityWorkQueue** field in the controller is used to hold the priority queue, and it is initialized in the **SetupWithManager** function.
- The **SetQueue** method is used to replace the default work queue with the priority queue.
- The **main** function initializes the manager, creates an instance of your custom controller, sets it up with the manager, and starts the manager.

```
import (
"container/heap"
"time"

ctrl "sigs.k8s.io/controller-runtime"
"sigs.k8s.io/controller-runtime/pkg/manager"
"sigs.k8s.io/controller-runtime/pkg/reconcile"
"sigs.k8s.io/controller-runtime/pkg/controller/controllerutil"
)

// PriorityItem represents an item in the priority queue.
type PriorityItem struct {
key      interface{} // Key used to identify the item
priority time.Time   // Priority of the item
}

// PriorityQueue implements a priority queue for work items.
type PriorityQueue []*PriorityItem

func (pq PriorityQueue) Len() int           { return len(pq) }
func (pq PriorityQueue) Less(i, j int) bool { return pq[i].priority.Before(pq[j].priority) }
func (pq PriorityQueue) Swap(i, j int)      { pq[i], pq[j] = pq[j], pq[i] }

// Push adds an item to the priority queue.
func (pq *PriorityQueue) Push(x interface{}) {
item := x.(*PriorityItem)
*pq = append(*pq, item)
}

// Pop removes and returns the highest-priority item from the priority queue.
func (pq *PriorityQueue) Pop() interface{} {
old := *pq
n := len(old)
item := old[n-1]
*pq = old[0 : n-1]
return item
}

// MyResourceReconciler is your controller with a priority work queue.
type MyResourceReconciler struct {
// Other fields...

PriorityWorkQueue PriorityQueue
}

// Enqueue a resource with a priority.
func (r *MyResourceReconciler) enqueueResourceWithPriority(resource *examplev1alpha1.MyResource,
priority time.Time) {
key, err := controllerutil.Key(resource)
if err != nil {
return
}
item := &PriorityItem{key: key, priority: priority}
heap.Push(&r.PriorityWorkQueue, item)
}

// Dequeue the highest-priority item from the work queue.
func (r *MyResourceReconciler) dequeueHighestPriorityItem() (interface{}, bool) {
if len(r.PriorityWorkQueue) == 0 {
return nil, true // Queue is empty
}
item := heap.Pop(&r.PriorityWorkQueue).(*PriorityItem)
```

```
return item.key, false
}

// Reconcile function
func (r *MyResourceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
// Fetch the resource using the request's NamespacedName
myResource := &examplev1alpha1.MyResource{}
if err := r.Get(ctx, req.NamespacedName, myResource); err != nil {
return ctrl.Result{}, client.IgnoreNotFound(err)
}

// Your reconciliation logic here...

// Example: Requeue the reconciliation after 1 hour with a priority
priority := time.Now().Add(time.Hour)
r.enqueueResourceWithPriority(myResource, priority)

return ctrl.Result{}, nil
}

// SetupWithManager sets up the controller with the manager.
func (r *MyResourceReconciler) SetupWithManager(mgr ctrl.Manager) error {
// Set the priority queue for the controller
r.PriorityWorkQueue = make(PriorityQueue, 0)
heap.Init(&r.PriorityWorkQueue)

// Create a new controller with the priority queue
c, err := ctrl.NewControllerManagedBy(mgr).
For(&examplev1alpha1.MyResource{}).
Complete(r)
if err != nil {
return err
}

// Replace the default work queue with the priority queue
c.(*ctrl.Controller).SetQueue(r.PriorityWorkQueue)

return nil
}

// Main function
func main() {
mgr, err := manager.New(cfg, manager.Options{})
if err != nil {
log.Fatal(err)
}

// Create an instance of your custom controller
myController := &MyResourceReconciler{}

// Setup the controller with the manager
if err := myController.SetupWithManager(mgr); err != nil {
log.Fatal(err)
}

// Start the manager
if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
log.Fatal(err)
}
}
```