

Implementacja maszyny wirtualnej dla funkcyjnych języków programowania wspierających przetwarzanie współbieżne.

Kajetan Rzepecki

**Wydział EAIiB
Katedra Informatyki Stosowanej**

21 listopada 2013

Maszyna wirtualna - środowisko uruchomieniowe języków programowania niezależniające je od platformy uruchomieniowej.

Maszyna wirtualna - środowisko uruchomieniowe języków programowania niezależniające je od platformy uruchomieniowej.

W skład pracy wchodzi:

- ✚ Implementacja interpretera kodu bajtowego.

Maszyna wirtualna - środowisko uruchomieniowe języków programowania uniezależniające je od platformy uruchomieniowej.

W skład pracy wchodzi:

- ✚ Implementacja interpretera kodu bajtowego.
- ✚ Implementacja kolektora obiektów nieosiągalnych.

Maszyna wirtualna - środowisko uruchomieniowe języków programowania uniezależniające je od platformy uruchomieniowej.

W skład pracy wchodzi:

- ✦ Implementacja interpretera kodu bajtowego.
- ✦ Implementacja kolektora obiektów nieosiągalnych.
- ✦ Implementacja Modelu Aktorowego (ang. Actor Model).

```
start() ->
```

```
Data = file:read("file.json"),    %% <<"Dane ...">>  
transmogrify(Data).
```

```
start() ->
    Data = file:read("file.json"),      %% <<"Dane ...">>
    transmogrify(Data).

transmogrify(Data) ->
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
    JSON = json:decode(Data),           %% {[Dane ...]}
    framework:map_reduce(Pids, JSON).  %% $#%~@

do_stuff(JSON) ->
    %% Operacje na danych.
    result.
```

```
transmoglify(Data) ->  
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),  
    framework:map_reduce(Pids, Data).  
  
do_stuff(Data) ->                                %% <<"Dane ...">>  
    JSON = json:decode(Data), %% {[Dane ...]} * bajylion  
    %% Operacje na danych.  
    result.
```



```
transmoglify(Data) ->
```

```
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),  
    framework:map_reduce(Pids, Data).
```

```
do_stuff(Data) ->                                %% <<"Dane ...">>  
    JSON = json:decode(Data), %% {[Dane ...]} * bajilion  
    %% Operacje na danych.  
    result.
```

✚ Mniejsza logika przepływu danych.

```
transmoglify(Data) ->
```

```
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),  
    framework:map_reduce(Pids, Data).
```

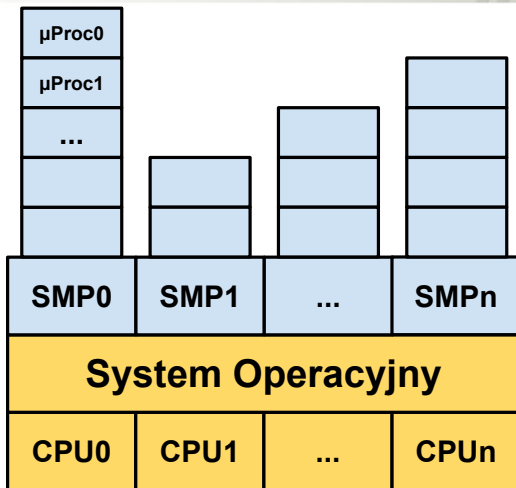
```
do_stuff(Data) ->                                %% <<"Dane ...">>  
    JSON = json:decode(Data), %% {[Dane ...]} * bazylion  
    %% Operacje na danych.  
    result.
```

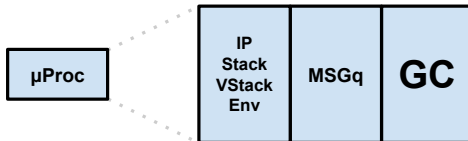
- ✦ Mniejsza logika przepływu danych.
- ✦ Zwielokrotnienie parsowania pliku JSON.

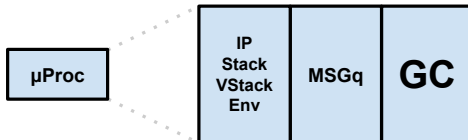
```
transmoglify(Data) ->  
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),  
    framework:map_reduce(Pids, Data).
```

```
do_stuff(Data) ->                                %% <<"Dane ...">>  
    JSON = json:decode(Data), %% {[Dane ...]} * bazylion  
    %% Operacje na danych.  
    result.
```

- ✦ Mniejsza logika przepływu danych.
- ✦ Zwielenokrotnienie parsowania pliku JSON.
- ✦ Działa szybciej. (!?)

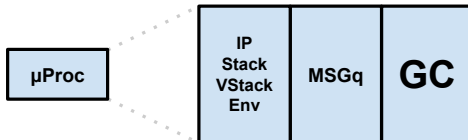






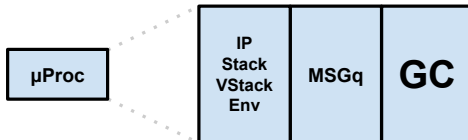
Oparty o **Three Instruction Machine**:

✚ Niewielka ilość rejestrów.



Oparty o **Three Instruction Machine**:

- ✚ Niewielka ilość rejestrów.
- ✚ Niewielka ilość instrukcji.



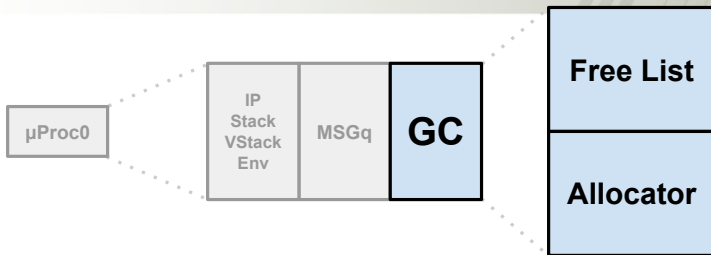
Oparty o **Three Instruction Machine**:

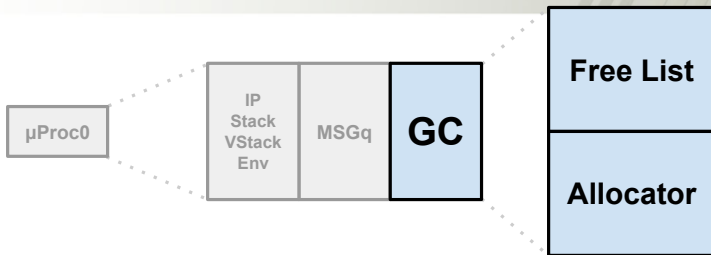
- ✚ Niewielka ilość rejestrów.
- ✚ Niewielka ilość instrukcji.
- ✚ Architektura **CISC**.


```
(define (add a b)
  (+ a b))
(add 2 2)
```

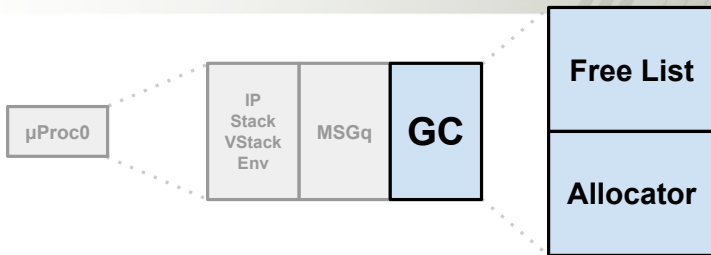
```
(define (add a b)
  (+ a b))
(add 2 2)
```

```
__start: PUSH 2      # Połóż "a" na stosie.
          PUSH 2
          ENTER add   # Wejdz do domknięcia "add".
add:      TAKE 2      # Pobierz dwa argumenty ze stosu.
          PUSH __add0
          ENTER 1
__add0:   PUSH __add1
          ENTER 0     # PUSHC 2, RETURN
__add1:   OP_ADD
          RETURN
```

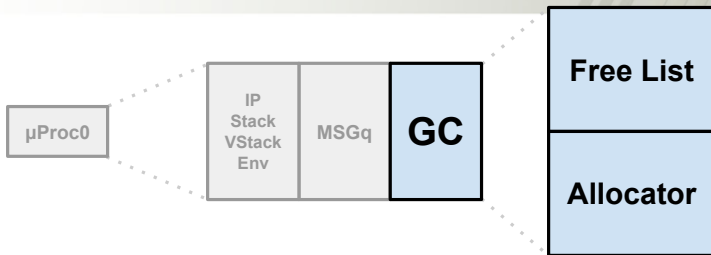




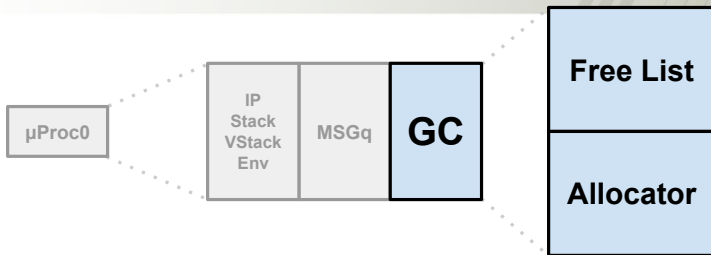
✚ Wykorzystuje zliczanie referencji.



- ✚ Wykorzystuje zliczanie referencji.
- ✚ Kolekcja pamięci procesu nie zależy od innych procesów.



- ✚ Wykorzystuje zliczanie referencji.
- ✚ Kolekcja pamięci procesu nie zależy od innych procesów.
- ✚ “Ostatni gasi światło.”



- ✚ Wykorzystuje zliczanie referencji.
- ✚ Kolekcja pamięci procesu nie zależy od innych procesów.
- ✚ “Ostatni gasi światło.”
- ✚ Proste obiekty (≤ 8 bajtów) są kopiowane.

```
TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}
```



```
TVMObj* alloc(uProc* context, size_t size) {  
    TVMObj* newObject;  
    if(hasSuitableObject(context, size)) {  
        newObject = popFreeList(context, size);  
        collect(context, newObject);  
    } else {  
        newObject = alloc(context.allocator, size);  
    }  
    atomicIncrement(&newObject.refCount);  
    return newObject;  
}  
  
void collect(uProc* context, TVMObj* object) {  
    foreach(TVMObj* pointer; object) {  
        free(context, pointer);  
    }  
}
```

```
TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}
```

```
void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}

void free(uProc* context, TVMObj* object) {
    if(atomicDecrement(object.refCount) == 0) {
        pushFreeList(context, object);
    }
}
```

```

TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}

```

```

void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}

void free(uProc* context, TVMObj* object) {
    if(atomicDecrement(object.refCount) == 0) {
        pushFreeList(context, object);
    }
}

```

✚ Szybka dealokacja.

```

TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}

```

```

void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}

void free(uProc* context, TVMObj* object) {
    if(atomicDecrement(object.refCount) == 0) {
        pushFreeList(context, object);
    }
}

```

✚ Szybka dealokacja.

✚ Szybka alokacja zamortyzowana listą wolnych obiektów.

```

TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}

void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}

void free(uProc* context, TVMObj* object) {
    if(atomicDecrement(object.refCount) == 0) {
        pushFreeList(context, object);
    }
}

```

- ✦ Szybka dealokacja.
- ✦ Szybka alokacja zamortyzowana listą wolnych obiektów.
- ✦ Pamięć nie jest natychmiastowo zwracana do Systemu Operacyjnego.

```

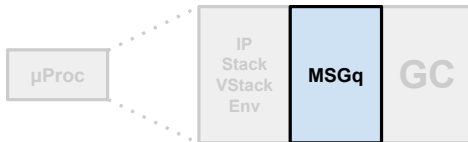
TVMObj* alloc(uProc* context, size_t size) {
    TVMObj* newObject;
    if(hasSuitableObject(context, size)) {
        newObject = popFreeList(context, size);
        collect(context, newObject);
    } else {
        newObject = alloc(context.allocator, size);
    }
    atomicIncrement(&newObject.refCount);
    return newObject;
}

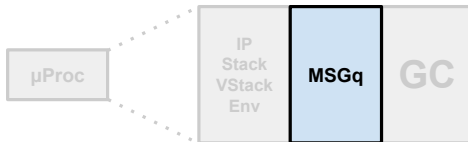
void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}

void free(uProc* context, TVMObj* object) {
    if(atomicDecrement(object.refCount) == 0) {
        pushFreeList(context, object);
    }
}

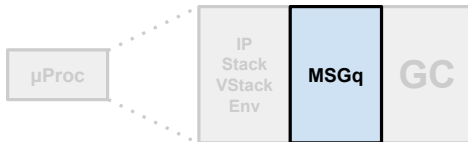
```

- ✦ Szybka dealokacja.
- ✦ Szybka alokacja zamortyzowana listą wolnych obiektów.
- ✦ Pamięć nie jest natychmiastowo zwracana do Systemu Operacyjnego.
- ✦ Wymaga atomowych operacji na liczniku referencji oraz barier pamięci.



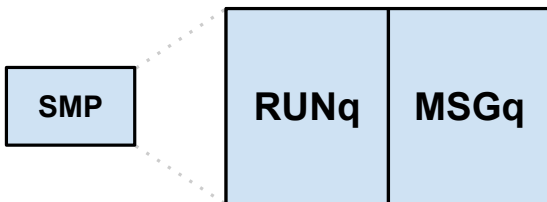


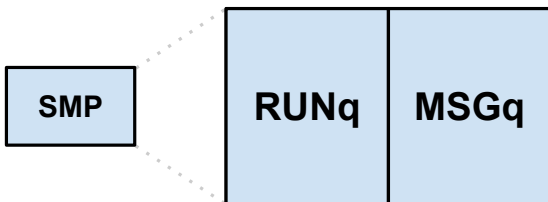
✚ **Pid** == wskaźnik na kontekst procesu.



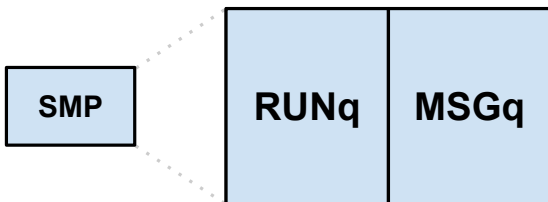
✚ **Pid** == wskaźnik na kontekst procesu.

✚ Wykorzystuje kolejki nieblokujące.

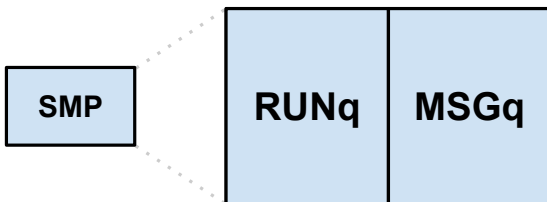




✚ Wykorzystuje Model Aktorowy!



- ✚ Wykorzystuje Model Aktorowy!
- ✚ Procesy są wywłaszczane (ang. preemptive concurrency).



- ✦ Wykorzystuje Model Aktorowy!
- ✦ Procesy są wywłaszczane (ang. preemptive concurrency).
- ✦ Obecnie brak automatycznego balansowania obciążenia procesorów.

Projekt implementuje:

- ✚ Interpreter kodu bajtowego oparty o **Three Instruction Machine**.

Projekt implementuje:

- ✦ Interpreter kodu bajtowego oparty o **Three Instruction Machine**.
- ✦ Kompilator kodu bajtowego.

Projekt implementuje:

- ✦ Interpreter kodu bajtowego oparty o **Three Instruction Machine**.
- ✦ Kompilator kodu bajtowego.
- ✦ Kolektor obiektów nieosiągalnych oparty o **opóźnione zliczanie referencji**.

Projekt implementuje:

- ✦ Interpreter kodu bajtowego oparty o **Three Instruction Machine**.
- ✦ Kompilator kodu bajtowego.
- ✦ Kolektor obiektów nieosiągalnych oparty o **opóźnione zliczanie referencji**.
- ✦ Model Aktorowy oparty o **kolejki nieblokujące**.

Dziękuję za uwagę.

Kajetan Rzepecki