AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Implementacja maszyny wirtualnej dla funkcyjnych języków programowania wspierających przetwarzanie współbieżne.

Kajetan Rzepecki

Wydział EAIiIB
Katedra Informatyki Stosowanej

21 listopada 2013

Wprowadzenie

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

Wprowadzenie

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

W skład pracy wchodzą:

- Implementacja interpretera kodu bajtowego.

Wprowadzenie

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

W skład pracy wchodzą:

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

Wprowadzenie

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

W skład pracy wchodzą:

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

Motywacja pracy - Problemy Erlanga

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

Motywacja pracy - Problemy Erlanga

```
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.
```

Motywacja pracy - Problemy Erlanga

```
transmogrify(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.
```

Motywacja pracy - Problemy Erlanga

```
transmogrify(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.

Motywacja pracy - Problemy Erlanga

```
transmogrify(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.
```

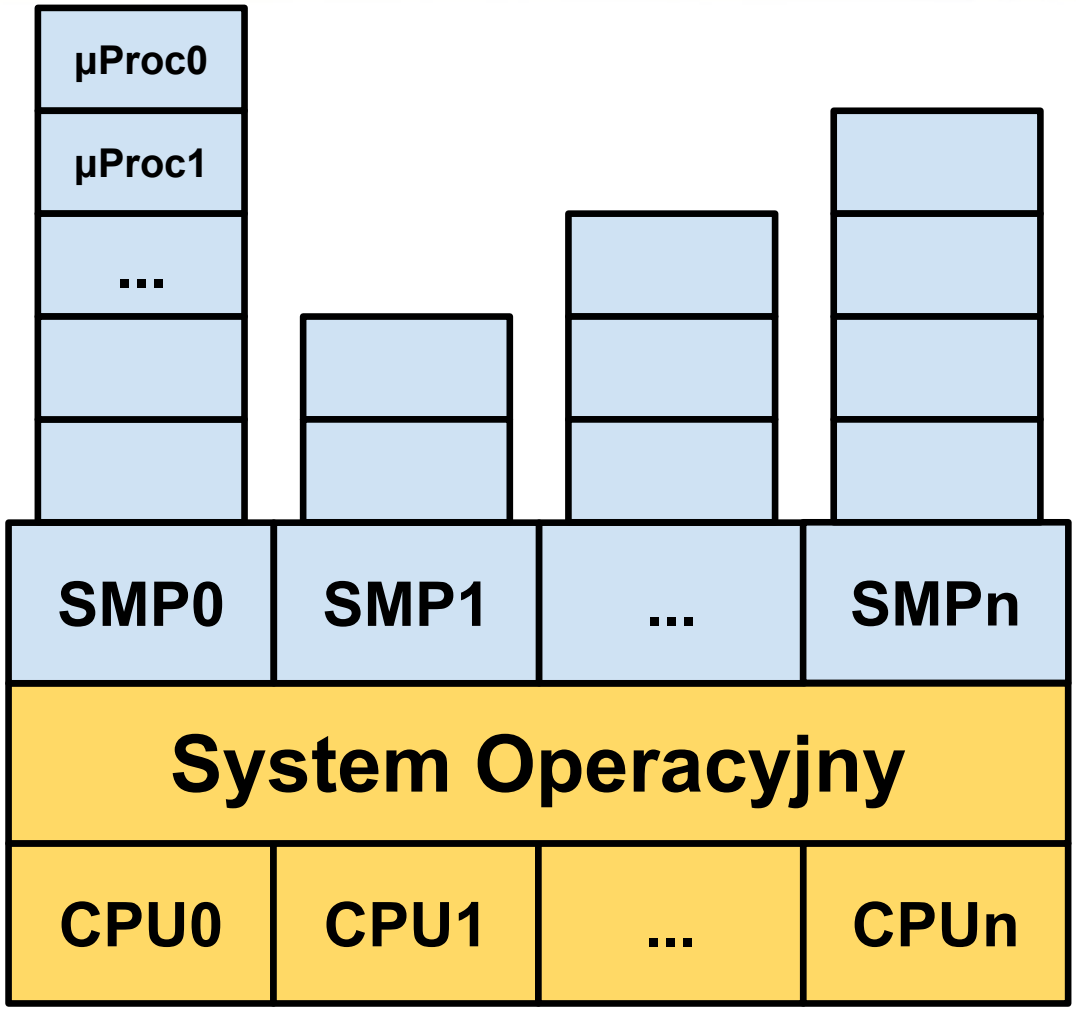
- Mniejsza logika przepływu danych.
- Zwielokrotnienie parsowania pliku JSON.

Motywacja pracy - Problemy Erlanga

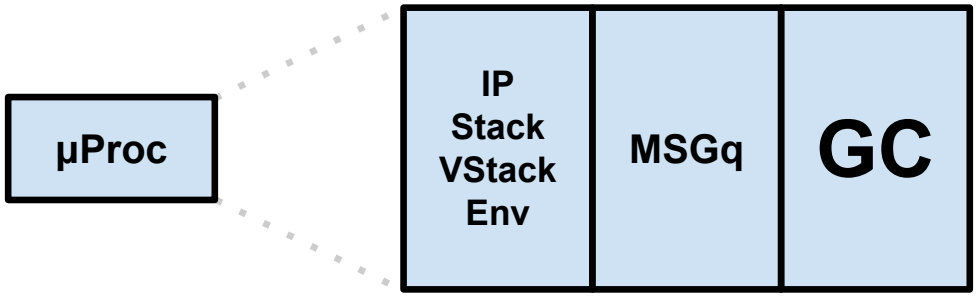
```
transmogrify(Data) ->
    Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
    framework:map_reduce(Pids, Data).

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

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

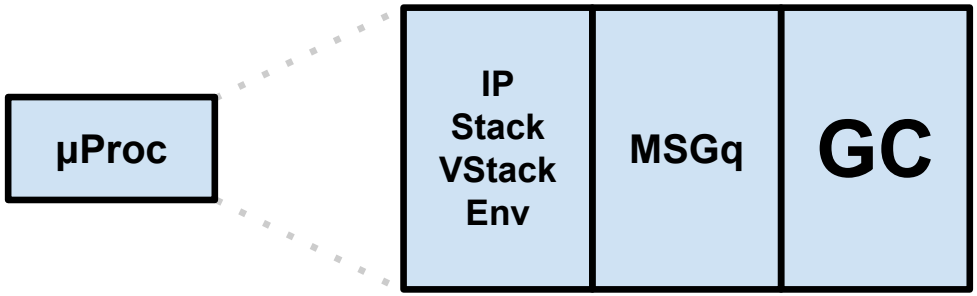
ThesisVM - Architektura



ThesisVM - Interpreter kodu bajtowego



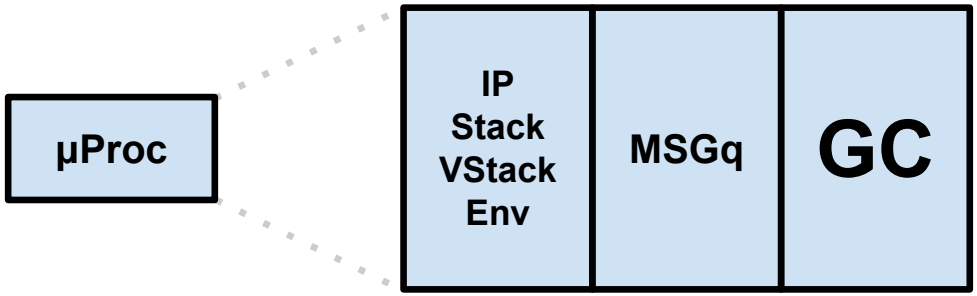
ThesisVM - Interpreter kodu bajtowego



Oparty o **Three Instruction Machine**:

- Niewielka ilość rejestrów.

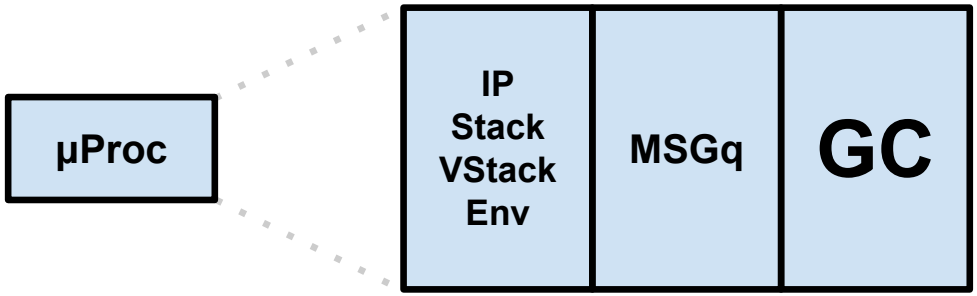
ThesisVM - Interpreter kodu bajtowego



Oparty o **Three Instruction Machine**:

- Niewielka ilość rejestrów.
- Niewielka ilość instrukcji.

ThesisVM - Interpreter kodu bajtowego



Oparty o **Three Instruction Machine**:

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

ThesisVM - Przykład kodu bajtowego

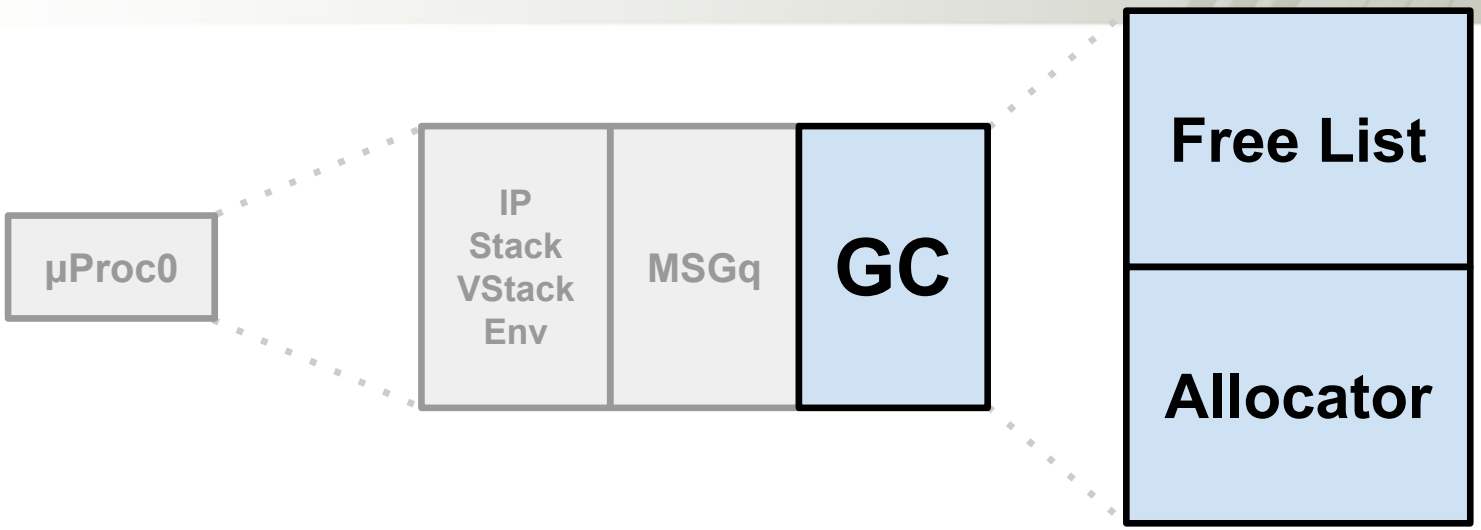
```
(define (add a b)
  (+ a b))
(add 2 2)
```

ThesisVM - Przykład kodu bajtowego

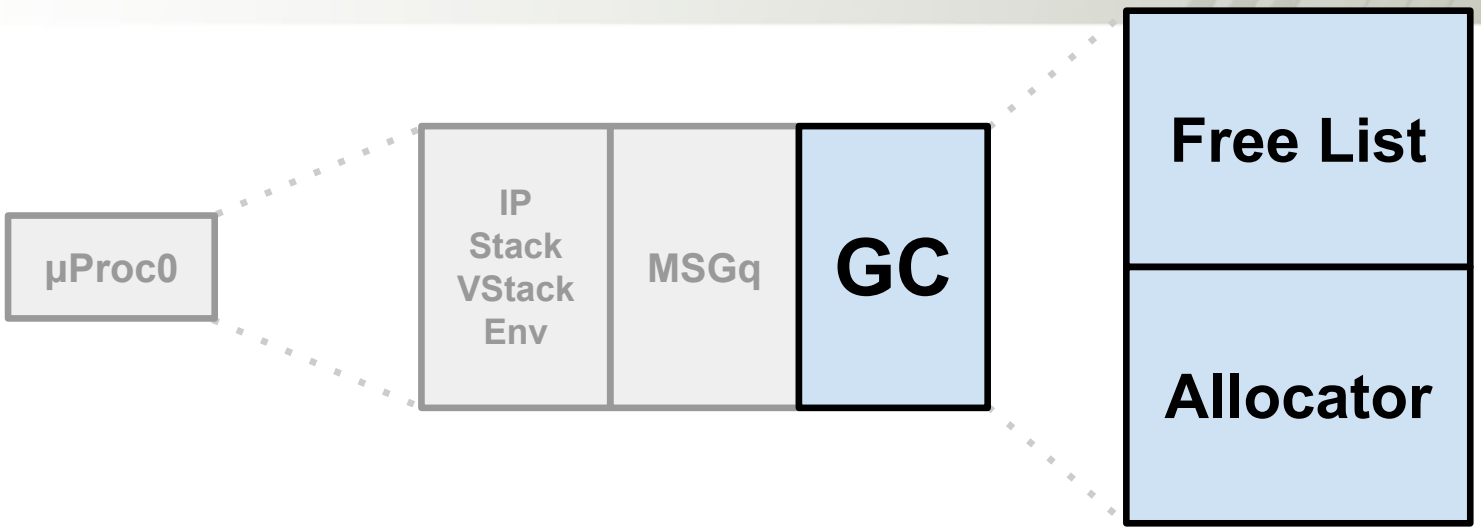
```
(define (add a b)
  (+ a b))
(add 2 2)
```

```
__start: PUSH 2        # Połóż "a" na stosie.
         PUSH 2
         ENTER add     # Wejdź 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
```

ThesisVM - Kolekcja śmieci

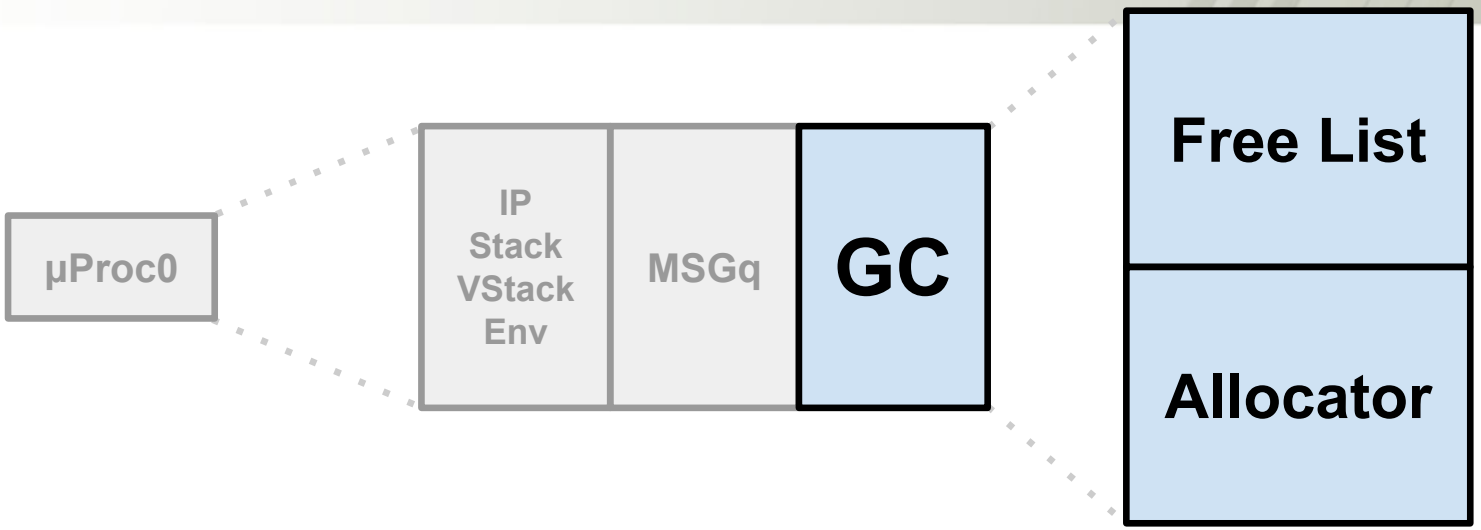


ThesisVM - Kolekcja śmieci



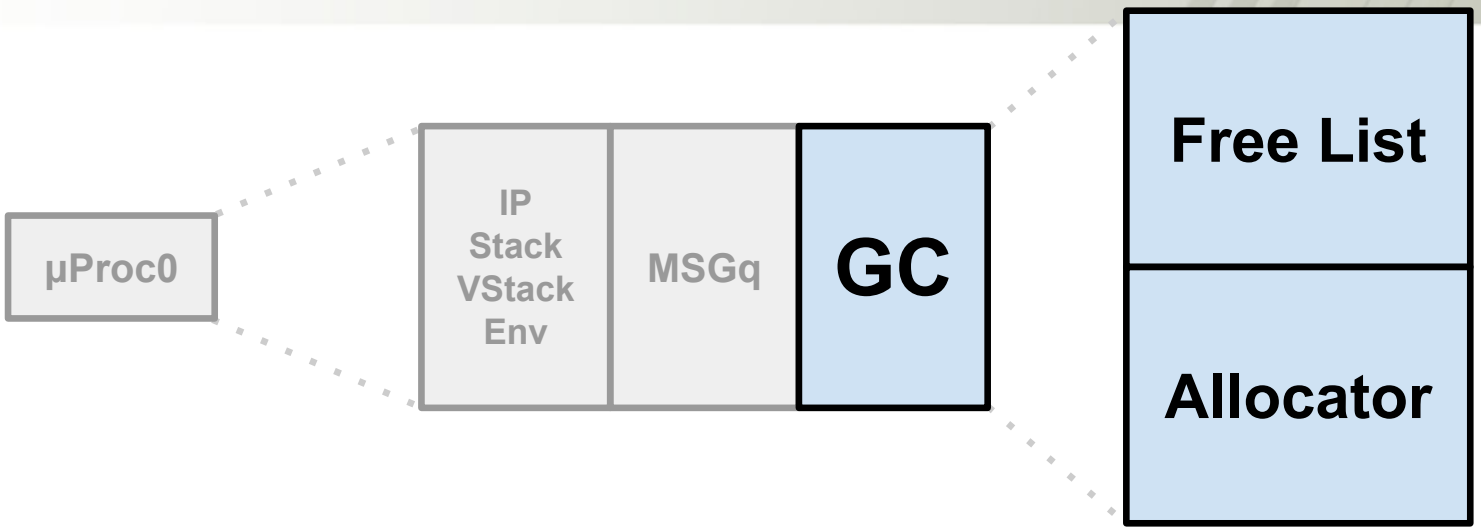
- Wykorzystuje zliczanie referencji.

ThesisVM - Kolekcja śmieci



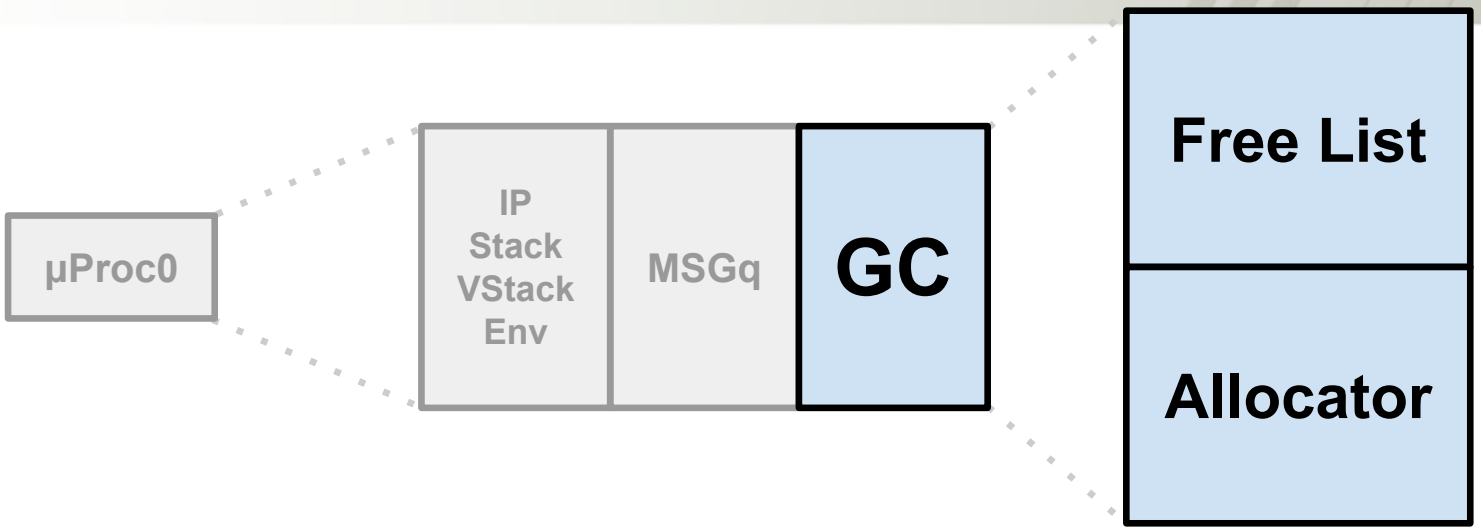
- Wykorzystuje zliczanie referencji.
- Kolekcja pamięci procesu nie zależy od innych procesów.

ThesisVM - Kolekcja śmieci



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

ThesisVM - Kolekcja śmieci



- Wykorzystuje zliczanie referencji.
- Kolekcja pamięci procesu nie zależy od innych procesów.
- "Ostatni gasi światło."
- Proste obiekty ($\leq$ 8 bajtów) są kopiowane.

ThesisVM - Opóźnione zliczanie referencji

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

ThesisVM - Opóźnione zliczanie referencji

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

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

ThesisVM - Opóźnione zliczanie referencji

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

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

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

ThesisVM - Opóźnione zliczanie referencji

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

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


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

- Szybka dealokacje.

ThesisVM - Opóźnione zliczanie referencji

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

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


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

- Szybka dealokacje.
- Szybka alokacja zamortyzowana listą wolnych obiektów.

ThesisVM - Opóźnione zliczanie referencji

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

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


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

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

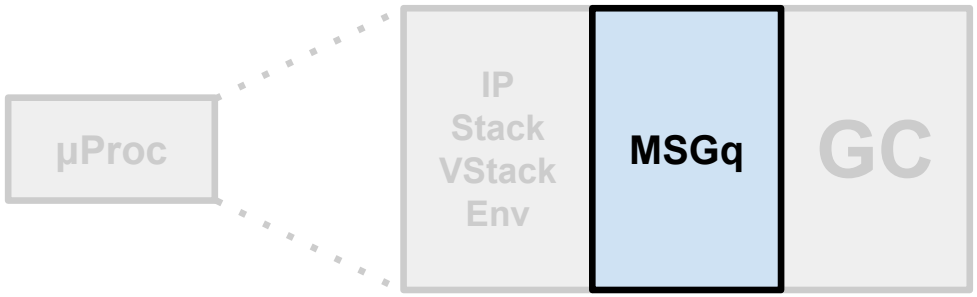
ThesisVM - Opóźnione zliczanie referencji

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

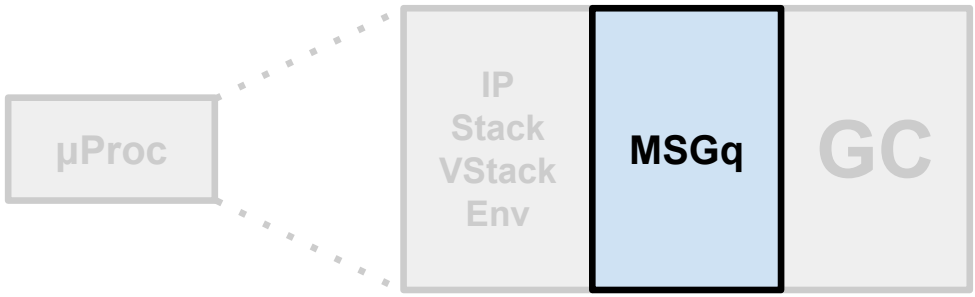
```
void collect(uProc* context, TVMObj* object) {
    foreach(TVMObj* pointer; object) {
        free(context, pointer);
    }
}


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

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

ThesisVM - Model Aktorowy

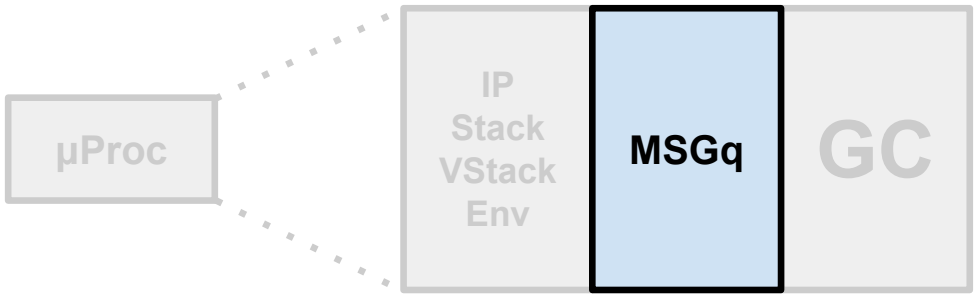


ThesisVM - Model Aktorowy



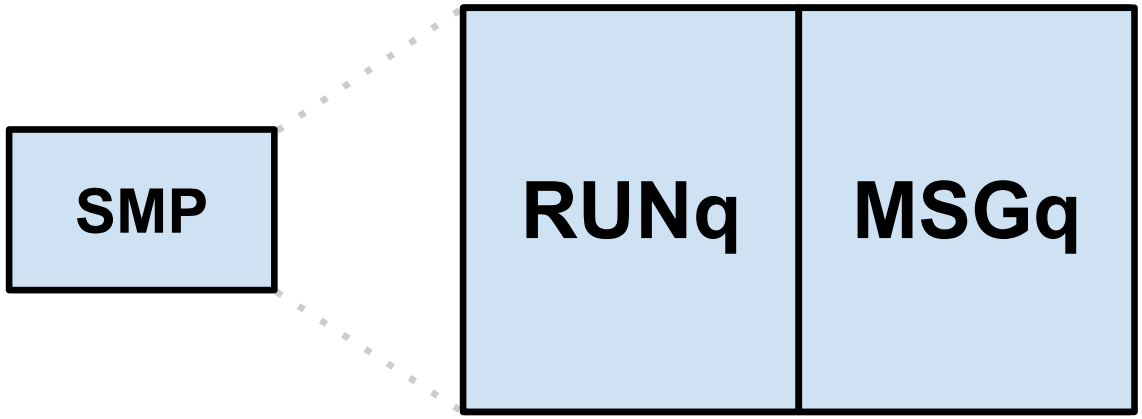
- **Pid** == wskaźnik na kontekst procesu.

ThesisVM - Model Aktorowy

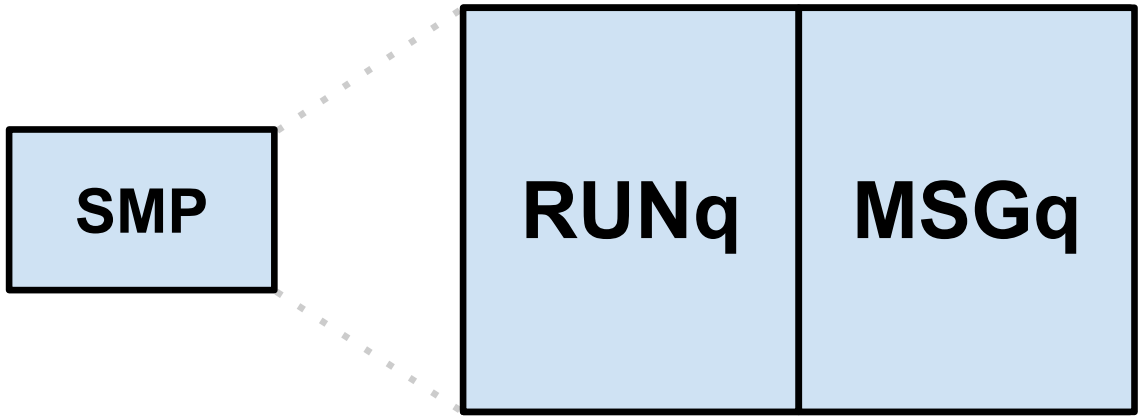


- **Pid** == wskaźnik na kontekst procesu.
- Wykorzystuje kolejki nieblokujące.

ThesisVM - Symmetric Multiprocessing

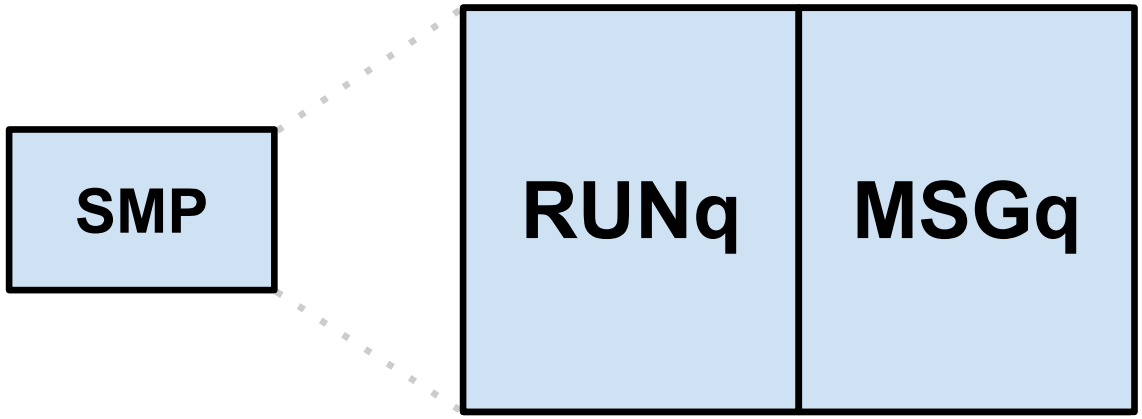


ThesisVM - Symmetric Multiprocessing



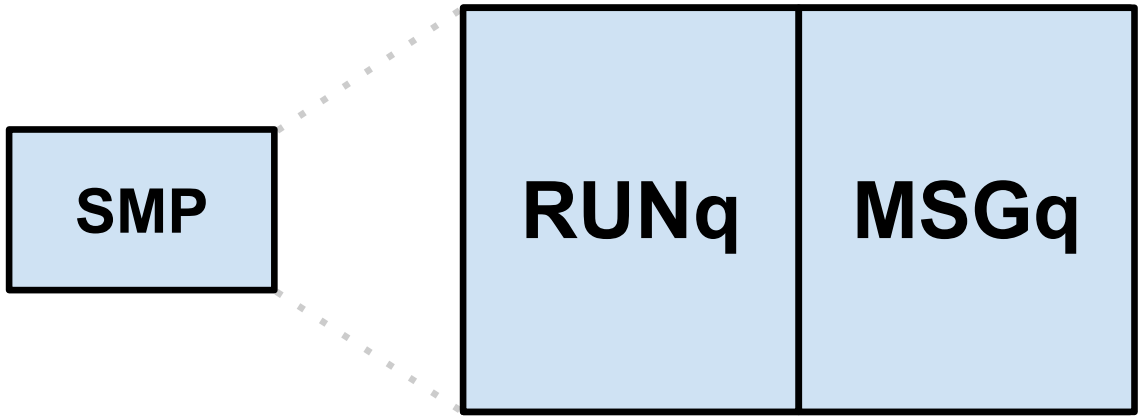
- Wykorzystuje Model Aktorowy!

ThesisVM - Symmetric Multiprocessing



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

ThesisVM - Symmetric Multiprocessing



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

Podsumowanie - TODO

Projekt implementuje:

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

Podsumowanie - TODO

Projekt implementuje:

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

Podsumowanie - TODO

Projekt implementuje:

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

Podsumowanie - TODO

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**.

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Dziękuję za uwagę.

Kajetan Rzepecki