Logging

"VProc sleeping"

We enrich vproc sleeping by extending the wake event with a cause argument. The cause argument indicates the reason that the vproc woke from sleeping. There are four possibilities:

- remote - Either another vproc or the "ping" thread signaled the host vproc.
- time - The time limit expired.
- interrupt - The OS sent an interrupt to pthread corresponding to the host vproc.
- error - An error occurred that is related to the OS locks and synchronization variables.

```json
{  "name" : "VProcWakeup",  
   "args" : [   
      Unknown macro: {"name"}  
   ],  
   "attrs" : ["rt"],  
   "desc" : "wakeup vproc from sleep" 
}
```

"Work stealing"

Each work group receives a unique ID.

```json
{  "name" : "WSInit",  
   "args" : [   
      Unknown macro: {"name"}  
   ],  
   "desc" : "work-stealing group initialization" 
}
```

The per-vproc worker-initialization event is dependent on its enclosing work group.

```json
{  "name" : "WSWorkerInit",  
   "args" : [   
      Unknown macro: {"name"},   
      Unknown macro: {"name"}  
   ],  
   "attrs" : ["rt"],  
   "desc" : "work-stealing worker initialization" 
}
```

The following two events indicate a worker executing work items and a busy worker being preempted, respectively.

```json
{  "name" : "WSExecute",  
   "args" : [   
      Unknown macro: {"name"}  
   ],  
   "attrs" : ["rt"],  
   "desc" : "work stealing execution of work items" 
}

{  "name" : "WSPreempted",  
   "args" : [   
      Unknown macro: {"name"}  
   ],  
   "attrs" : ["rt"],  
   "desc" : "work-stealing worker has been preempted" 
}```
Below we introduce five events to mark the progress of the thief. The first occurs when the thief worker sends a its thief liaison process to a randomly-chosen victim. The second and third events mark the interval when the thief process executes on the victim vproc. The last two events indicate when the thief receives notification on the result of the steal.

```json
{
  "name": "WSThiefSend",
  "args": [
    Unknown macro: {"name"}
  ],
  "attrs": ["rt"],
  "desc": "work-stealing thief send"
}

{
  "name": "WSThiefBegin",
  "args": [
    Unknown macro: {"name"}
  ],
  "attrs": ["rt"],
  "desc": "work-stealing thief begins executing on the victim vproc"
}

{
  "name": "WSThiefEnd",
  "args": [
    Unknown macro: {"name"}
  ],
  "attrs": ["rt"],
  "desc": "work-stealing thief ends executing on the victim vproc"
}

{
  "name": "WSThiefSuccessful",
  "args": [
    Unknown macro: {"name"}
  ],
  "attrs": ["rt"],
  "desc": "work-stealing thief successfully stole a work item"
}

{
  "name": "WSThiefUnsuccessful",
  "args": [
    Unknown macro: {"name"}
  ],
  "attrs": ["rt"],
  "desc": "work-stealing thief failed to steal a work item"
}
```
When a thief has exceeded its maximum duration of consecutive steal attempts, the thief goes to sleep for a while. We do this to avoid flooding workers with steal attempts.

```
{ "name" : "WSSleep",
 "args" : [
     Unknown macro: {"name"}
 ],
 "attrs" : ["rt"],
 "desc" : "work-stealing worker goes to sleep"
}
```

The following group defines some state transitions and dependencies specific to the thief.

```
{ "desc" : "Work stealing thief events",
 "kind" : "GROUP",
 "events" : [],
 "groups" : [
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
     Unknown macro: {"desc"},
 ]
}
```

Finally, we define the state transitions that workers may make. Each worker is always in one of the four states:

- Inactive - Either uninitialized, terminated, or waiting to be scheduled.
- Busy - Working productively on part of the computation.
- Thief - Attempting to steal work items.
- Sleeping - Doing nothing for a while to avoid flooding busy workers with steal attempts.

The following group defines the allowable state transitions.

```
{ "desc" : "Ropes"
}
```

The primary event of interest related to ropes is that of rebalancing, since excessive rebalancing might be the cause of a rather subtle performance bug. We therefore introduce two rebalances events to denote the start and end point of the rebalancing. Both events carry the length of the rope being rebalanced.

```
{ "name" : "RopeRebalanceBegin",
 "args" : [
     Unknown macro: {"name"}
 ],
 "attrs" : ["pml"],
 "desc" : "A rope is being rebalanced."
 }
```

```
{ "name" : "RopeRebalanceEnd",
 "args" : [
     Unknown macro: {"name"}
 ]
}
```
A rope is finished being rebalanced.