Формальная модель партицированной операционной системы реального времени на Promela

Сергей Старолетов
serg_soft@mail.ru

АлтГТУ им. И.И. Ползунова

Online, 5 ноября 2020 г.

IoT, киберфизические системы и Индустрия 4.0

Модели киберфизических систем:

- Кибер-часть: дискретная система переходов;
- Физическая часть: непрерывная физическая модель.

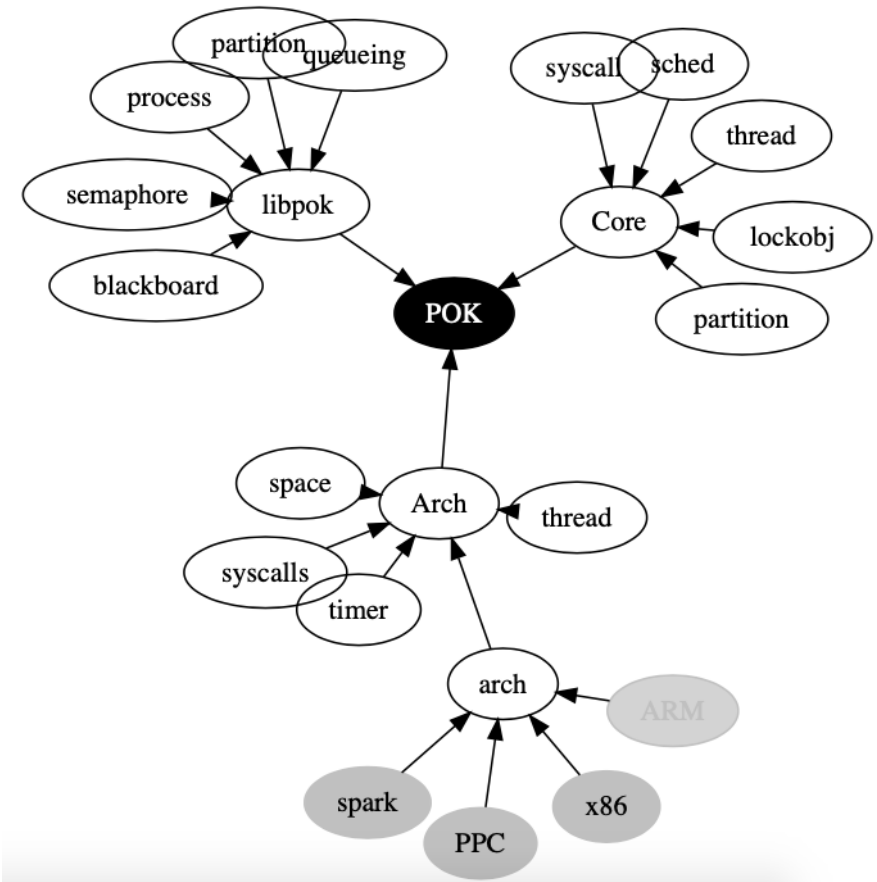
Системная часть – embedded ОС!

POK

Это лицензируемая по BSD открытая ОС, в некоторой мере соответствующая стандартам ПО для авионики, была создана в научно-исследовательском институте во Франции в качестве диссертации Julien Delange, цель создания – применить инженерию на основе моделей для описания конфигурации системы:

- это отличный proof-of-concept, состоящий из ядра и множества примеров;
- частично соответствует ARINC 653 стандарту real-time ОС для бортовых систем;
- защищенные партиции с разделением времени и памяти;
- real-time планировщики с возможностью задания различных стратегий планирования двух видов: (a) планировщик партиций (b) планировщик процессов внутри партиции.

Схема внутренней архитектуры POK



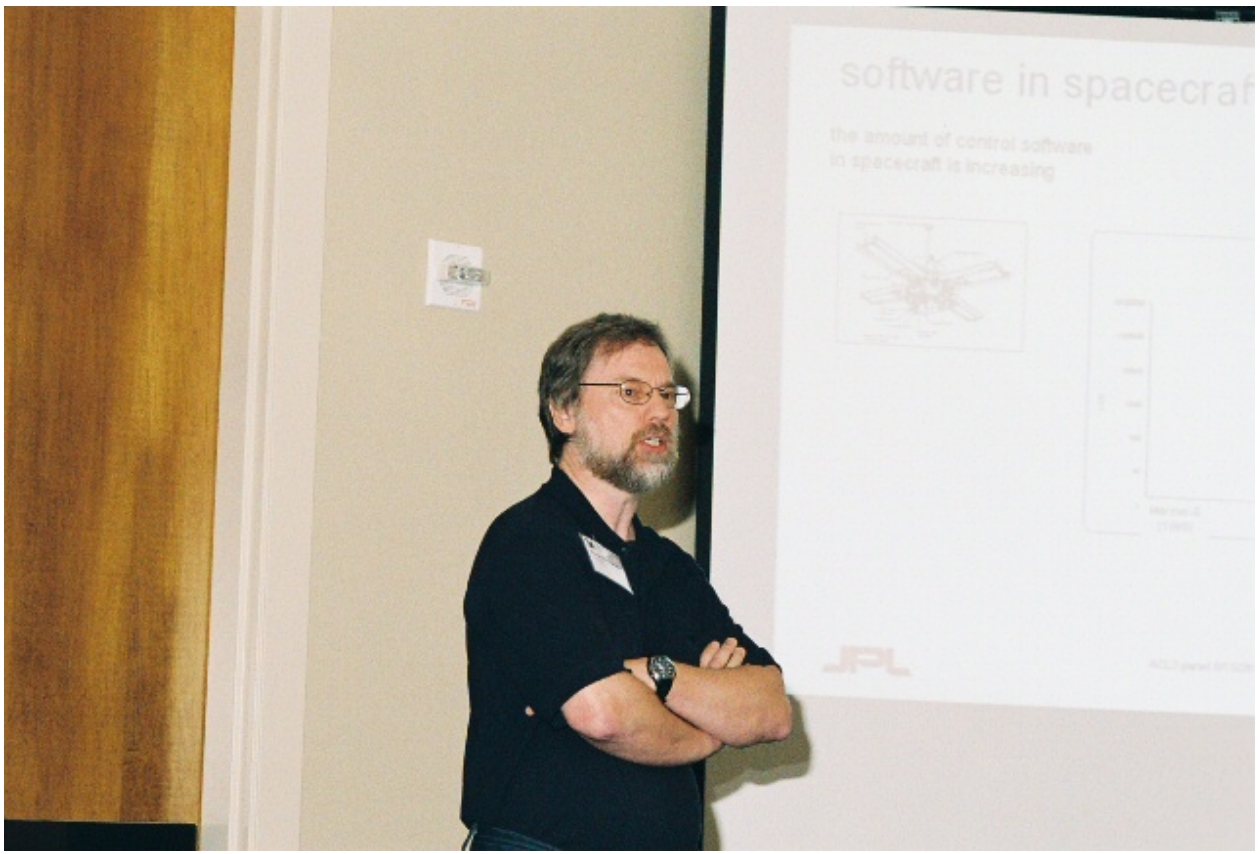
* ARINC 653-совместимое API позволяет работать с партициями, процессами, примитивами, портами, очередями и сообщениями стандартизированным способом.

Model Checking с помощью SPIN

- SPIN – это свободная утилита для проверки корректности моделей распределенных программ. Аббревиатура SPIN означает Simple Promela INterpreter (простой интерпретатор языка Promela).
- Система SPIN верифицирует не сами программы, а их модели.
- Для построения модели оригинальной параллельной программы или алгоритма, инженер (обычно вручную) строит такую модель на C-подобном входном языке, называемым Promela (PROtocol MEta-LAnguage).

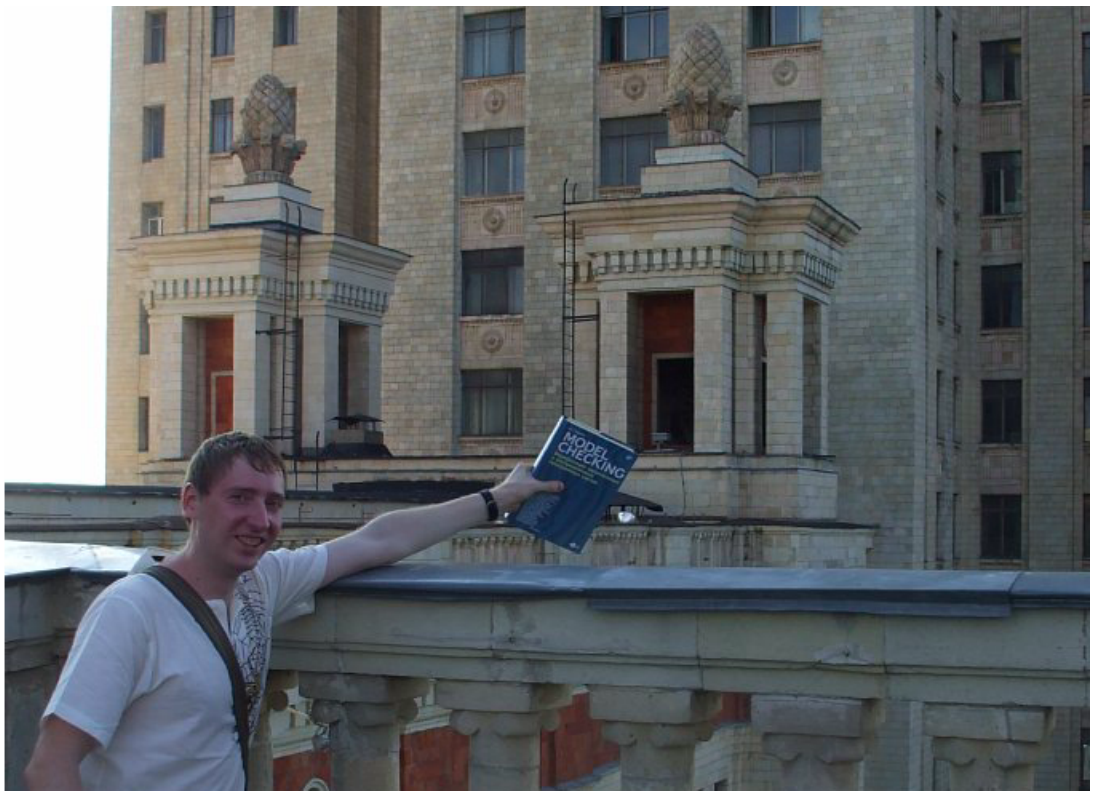
Model Checking с помощью SPIN

Автор SPIN Gerard J. Holzmann



Model Checking с помощью SPIN

С книгой “Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем” на МГУ

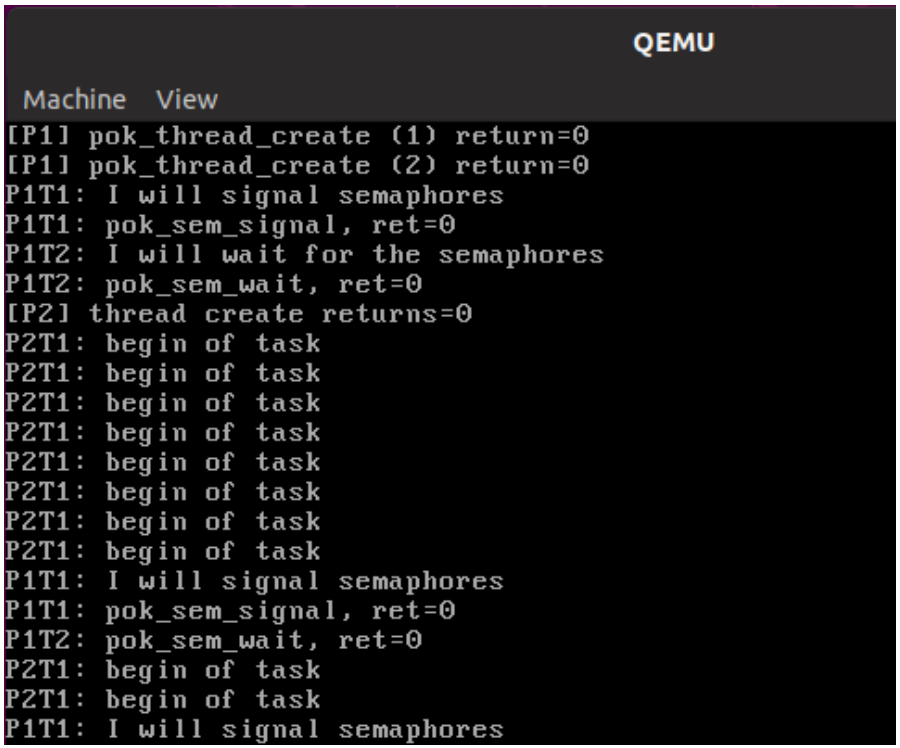


Promela

Для формализации наших задач, нам понадобятся следующие особенности языка Promela:

- он является акторным (процесс- и сообщение-ориентированным) языком;
- он специально предназначен для описания протоколов и взаимодействующих систем;
- он имеет C-стилизированный синтаксис и типы данных ограниченного размера;
- имеются inline-функции, похожие на макросы в C;
- имеется возможность описания своих типов (с использованием typedef как в C);
- имеются “atomic”-секции для моделирования кода без возможности переключения процессов внутри секции.

Пример для изучения



Минимальный пример клиентского кода POK, работающего в партициях, запущен в среде QEMU.

Пример для изучения

Листинг 1: Многопоточный пример

```
void* pinger_job ()
{
pok_ret_t ret;
 while (1)
 {
  printf ("P1T1: I will signal semaphores\n");
  ret = pok_sem_signal (sid);
  printf ("P1T1: pok_sem_signal, ret=%d\n", ret);
  pok_thread_sleep (2000000);
 }
```

- это реально минимальный пример поведения партицированной ОС;
- он содержит мнопоточность, несколько партиций, а также примитивы блокировки и засыпание процессов, поэтому подходит для моделирования алгоритмов динамического планирования.

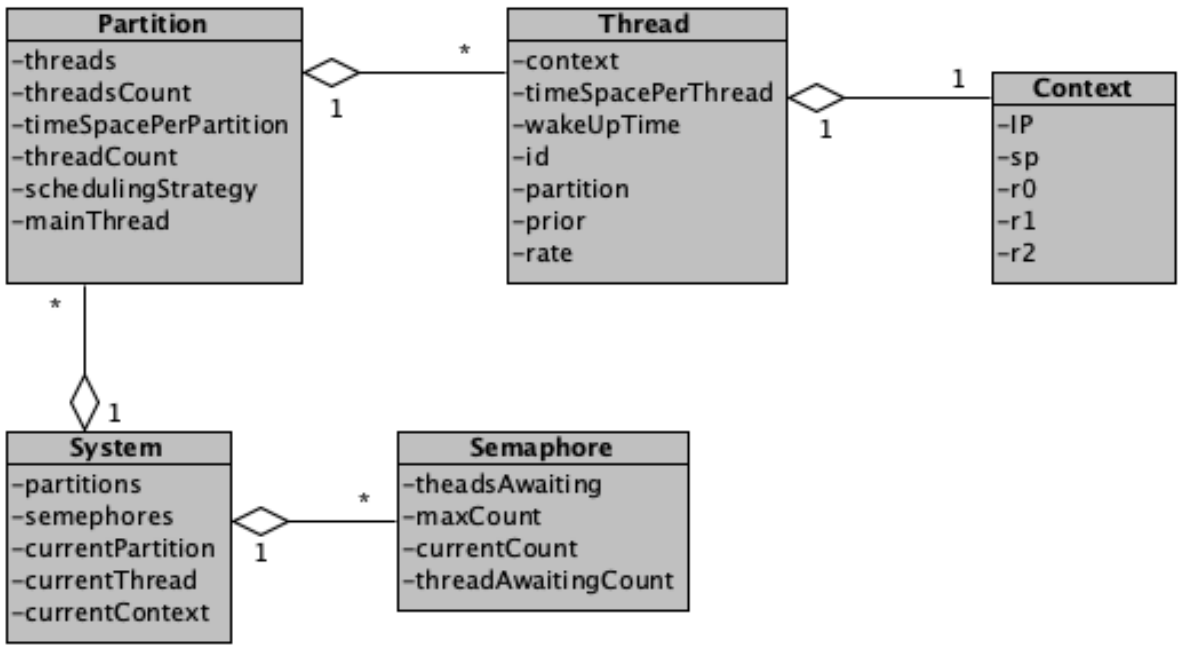
Модель для минимального примера

```
proctype threadP1T1(short myPartId; short myThreadId) {
do
::(osLive == 1) ->
atomic {
 if ::(currentPartition == myPartId
  && currentThread == myThreadId && currentContext.IP == 0) ->
     {
      pok_print(P1T1_I_will_signal_semaphores);
      currentContext.IP++;
     }
   ::else ->
  if ::(currentPartition == myPartId &&
 currentThread == myThreadId && currentContext.IP == 1) ->
     {
      pok_sem_signal(sid, currentContext.r0);
      currentContext.IP++;
     }
  ::else ->
 if ::(currentPartition == myPartId &&
  currentThread == myThreadId && currentContext.IP == 2) ->
    {
     pok_printf(P1T1_pok_sem_signal_ret, currentContext.r0);
     currentContext.IP++;
    }
  ::else ->
 if ::(currentPartition == myPartId &&
  currentThread == myThreadId && currentContext.IP == 3) ->
    {
     pok_delay(2000);
     currentContext.IP = 0; /* inf loop */
    }
  ::else -> skip;
   fi
  fi
 fi
fi
}
::else -> break;
od
}
```

Свойства модели минимального кода

- Для каждого процесса в системе создается соответствующий процесс в Promela.
- Для всех своих вычислений, данный процесс должен использовать данные из переменных-регистров из текущего контекста.
- После каждого значимого действия увеличивается регистр IP.
- Каждый оператор выполняется в операторе “switch” по регистру IP, текущему процессу и текущей партиции.

Определение модели данных



Определение модели данных

Чтобы смоделировать выполнение кода, мы явно вводим регистры процессора как переменные Promela и помещаем их в текущий контекст выполнения, который имитирует один процессор с его памятью. В первую очередь это регистр для указателя текущей инструкции (IP), регистр стека (sp) и несколько арифметических регистров (r0-rn), см. Листинг 3:

```
typedef Context {
    int IP;          //instruction pointer
    int sp;          //stack pointer - for further modeling
    int r0;          //arithmetic registers
    int r1;
    int r2;
}
```

Листинг 3. Модель состояния текущего процесса..

IP используется для регистра потока выполнения в процессе (см. Листинг 2), арифметические регистры должны использоваться в вычислениях, sp добавляется для будущего использования, например, для моделирования локальной памяти, процедур и передачи параметров. Затем мы включаем такой контекст в определение потока (см. Листинг 4):

```
typedef Thread {
    Context context;  //thread context to save
    short timeSpacePerThread; //count of ticks to thread work
    bit isLocked;     //1 if it has been locked on a semaphore
    int wakeUpTime;   //wake up time to schedule using sleep
    short id;         //unique thread id
    short partition;  //number of parent partition
    short prior;      //for further model with priorities
    short rate;       //current execution time - for rms
}
```

Листинг 4. Определение данных для потоков (клиентских процессов).

Поток характеризуется своим контекстом, некоторыми параметрами и временем работы в тиках, рассчитываемых планировщиком. В конце концов, мы представляем определение раздела, как показано в Листинге 5:

```
typedef Partition {
    short timeSpacePerPartition; //count of ticks to run
    short threadCount;
    Thread threads[MAXTHREADS];  //threads of this partition
    short schedulingStrategy;    //type of sched for threads
    short mainThread;            //first thread to run
}
```

Листинг 5. Определение данных для разделов (партиций).

Моделирование планировщика

Для первой итерации моделирования, представим простейший недетерминированный планировщик, который случайно выбирает партицию и процесс внутри нее, см. Листинг 6:

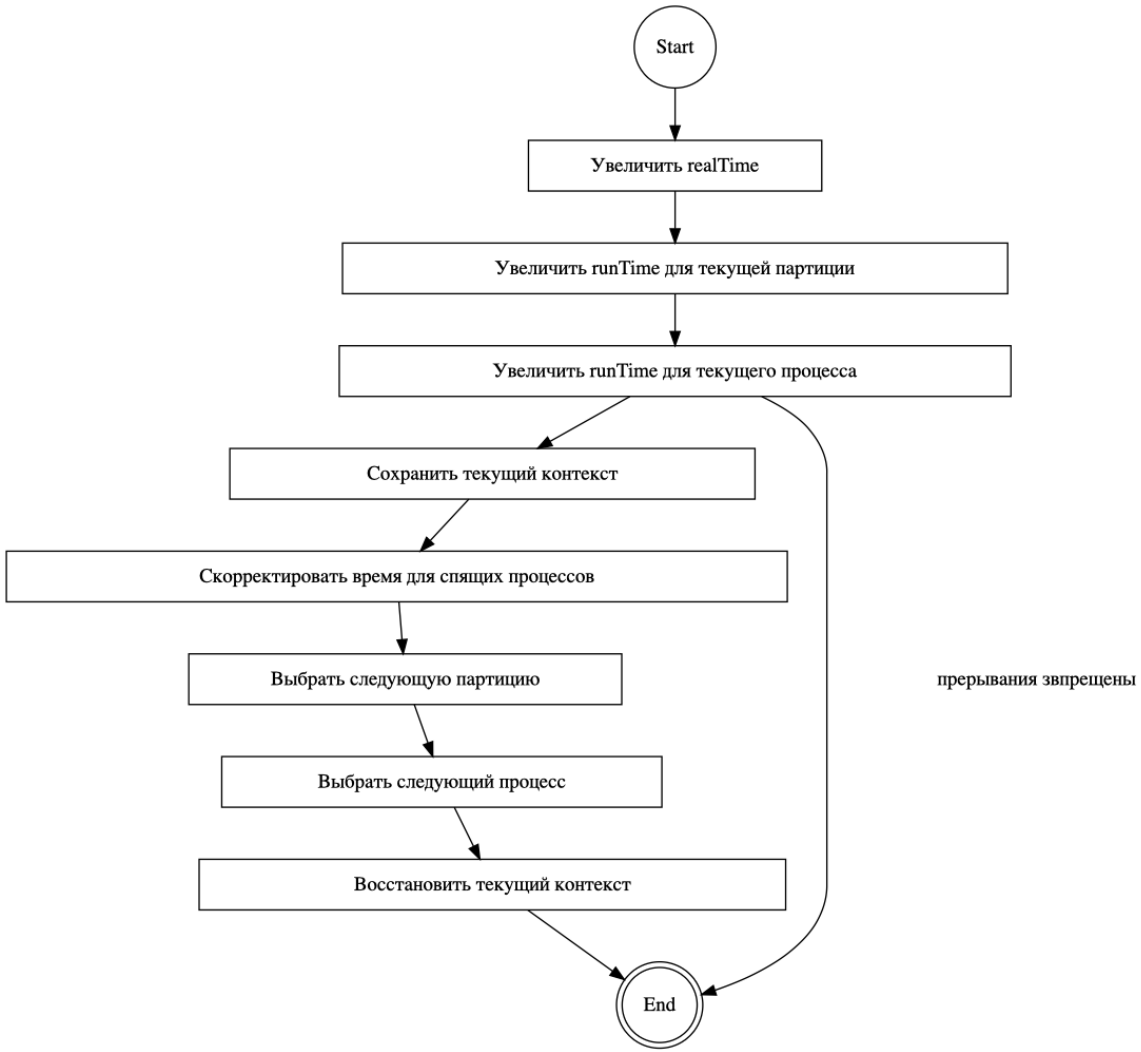
```
proctype schedNonDeterministicInstance() {
 do
 :: realTime < MAXTIMESIM -> {
  atomic {
    saveCurrentContext();

    //non-deterministic partitions scheduler
    if
      ::true -> currentPartition = 0;
      ::true -> currentPartition = 1;
    fi

    if
      ::(currentThread == 0) -> currentThread = 1; //stub
      :: else -> currentThread = 0;
    fi
      realTime++;
      restoreCurrentContext();
   }
  }
 :: else -> {
              printf("Simulation time is over!\n");
              osLive = 0;
              break;
             }
 od
}
```

Listing 6. Simple scheduler that peaks random partitions and threads.

Моделирование планировщика

- Планировщик работает как процесс Promela; он активируется в некоторое случайное время.
- Время работы системы ограничено константой, текущее время считается в переменной realTime, то есть мы увеличиваем текущее время в одном месте, когда активизируется процесс планировщика (соответствует обработке прерываний от аппаратного таймера).
- Макросы saveCurrentContext and restoreCurrentContext используются для сохранения текущего контекста текущего процесса и его восстановления.
- То есть с использованием этих идей, можно реализовать простую модель планировщика.

Моделирование партицированного планировщика

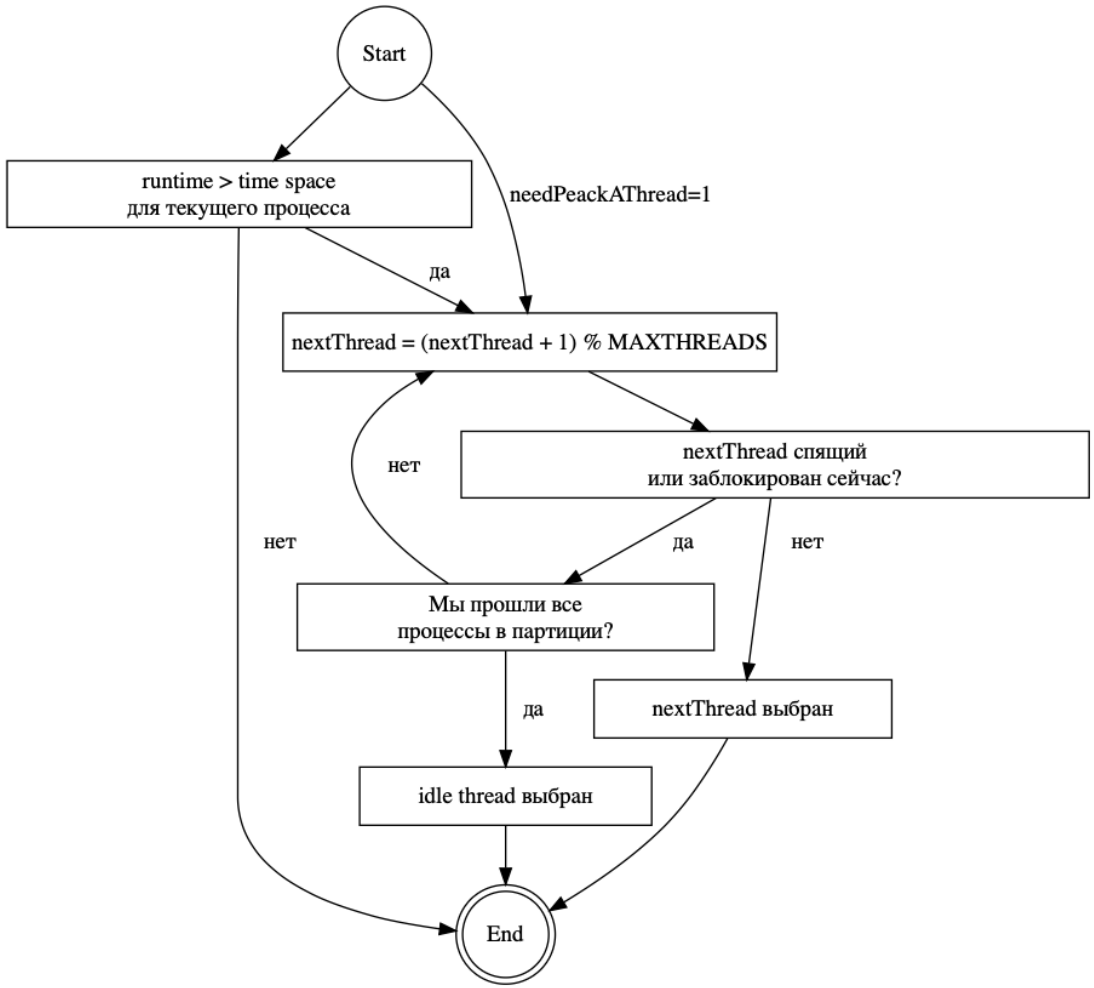


Стратегии планирования

```
mtype = {sched_part_rms_strategy, sched_part_rr_strategy,
sched_part_edf_strategy, sched_part_llf_strategy}

inline elect_next_thread(needPeakAThread) {
if
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_rms_strategy) -> sched_part_rms(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_rr_strategy)  -> sched_part_rr(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_llf_strategy) -> sched_part_llf(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_edf_strategy) -> sched_part_edf(needPeakAThread);
::else -> skip;
fi
}
```

Listing 7. An extensible thread election.

Модель round-robin планировщика с возможностью блокировки и засыпания процессов



Моделирование системных вызовов (syscalls)

- В системе POK, все API, которое предоставляется ОС ее клиентским процессам (например, создание семафора, ожидание его или блокировка), осуществляется посредством системных вызовов (syscalls).
- Это означает, что для каждого взаимодействия с объектом ядра будет выполнена генерация программного прерывания.
- Настоящий подход позволяет контролировать такие вызовы со стороны ОС, возможность приоритизировать их, выполнять в защищенном контексте.

Моделирование системных вызовов (syscalls)

```
//syscalls types
mtype = {syscall_sem_p, syscall_sem_v, syscall_delay,
syscall_printf}

//library available to user
inline pok_sem_signal(sid, ret) {
    printf("pok_sem_signal\n");
    pok_do_syscall(syscall_sem_v, sid, NOPARAM, ret);
}

inline pok_sem_wait(sid, ret) {
    printf("pok_sem_wait\n");
    pok_do_syscall(syscall_sem_p, sid, NOPARAM, ret);
}
```

Listing 8. Syscalls user library.

Моделирование системных вызовов (syscalls)

```
inline pok_do_syscall(N, param1, param2, ret) {
  atomic {
    //pass the params
    currentContext.r0 = N;
    currentContext.r1 = param1;
    if
        ::(param2 != NOPARAM) -> currentContext.r2 = param2;
        ::else -> skip
    fi
    }
    //emit the interrupt
    InterruptController ! POK_INTERRUPT;
    //wait for iret
    InterruptRet ? ret;
}
```

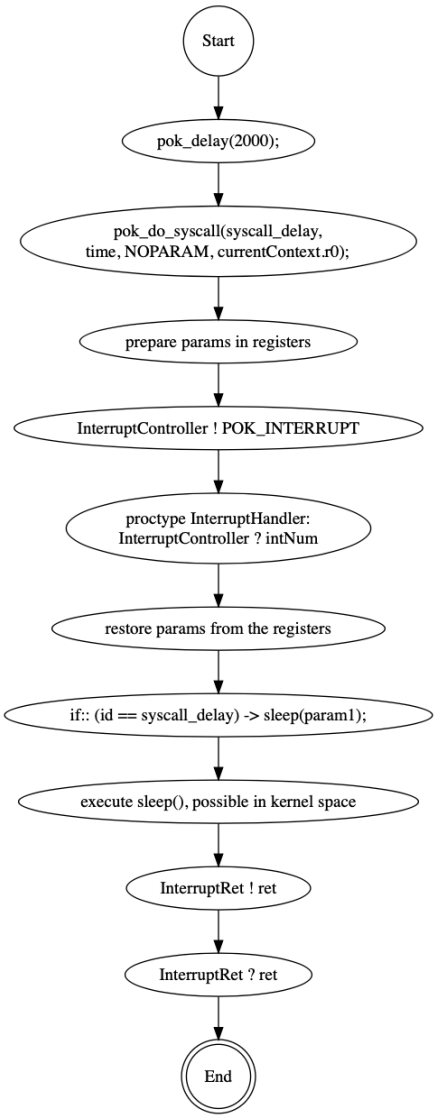
Listing 9. Syscalls executor.

Моделирование системных вызовов (syscalls)

```
interruptsDisabled = 1; //stop the scheduler
saveCurrentContext();
if
 ::(intNum == POK_INTERRUPT) -> {
   if
     ::(id == syscall_sem_v)   -> sem_signal(param1);
     ::(id == syscall_sem_p)   -> sem_wait(param1);
     ::(id == syscall_delay)   -> sleep(param1);
     ::(id == syscall_printf) -> print(param1, param2);
     ::else -> skip; //unknown syscall id
   fi
 }
 ::else -> skip;
fi
restoreCurrentContext();
```

Listing 10. A part of syscalls handling.

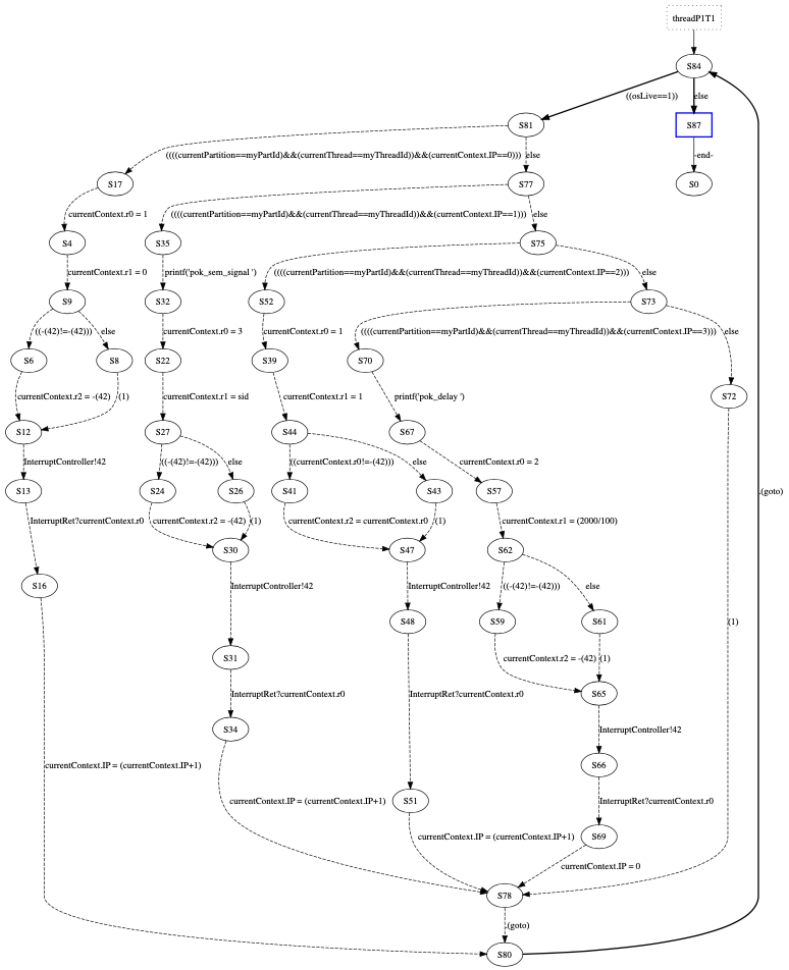
Моделирование системных вызовов (syscalls)



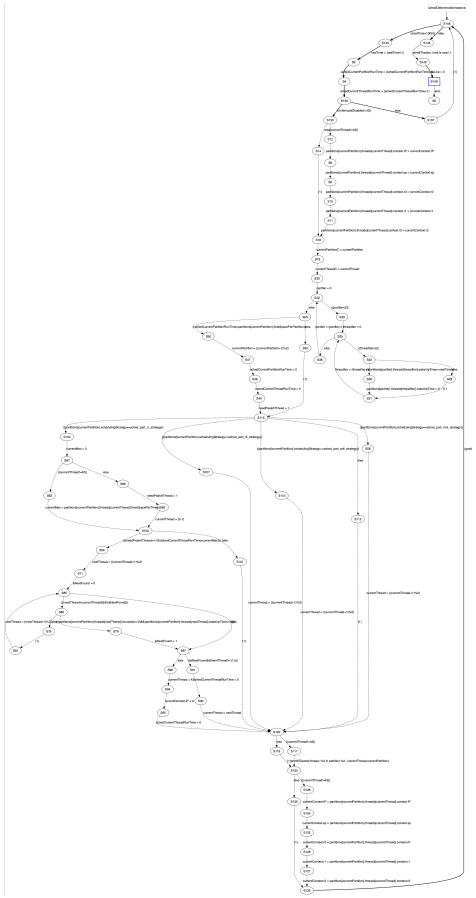
Дискуссия (система переходов по программе на языке с формальной семантикой)

- С использованием возможности SPIN экспортировать автомат модели как .dot диаграмму (./pan -D), были сгенерированы представления процессов, а также планировщика.
- Автомат процесса (соответствующего Листингу 1) содержит около 80 состояний, мы можем видеть, что часть состояний повторяется за счет inline макросов для моделирования syscalls.
- Автомат планировщика содержит примерно 150 состояний.
- Создание таких систем переходов вручную проблематично, поэтому выполнимая спецификация на Promela реально помогает описывать сложные формальные модели.

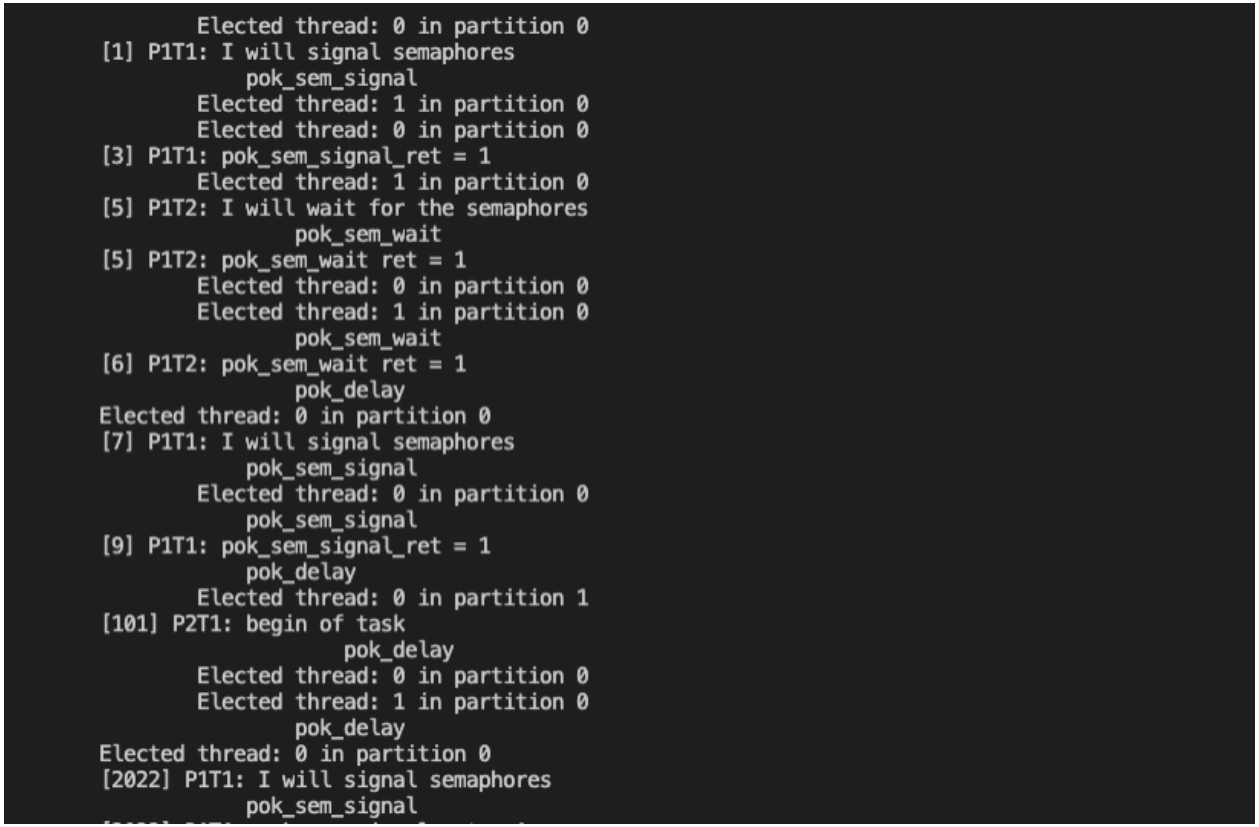
Пример автомата процесса



Пример автомата планировщика



Симуляция работы модели



Связанные работы

Самый известный подход к верификации ОС представлен в [Klein G. et al. seL4]. Авторы создали исполняемые спецификации микроядра L4 в Haskell на основе первоначальной реализации C, а затем уточнили их до модели Isabelle / HOL. В их статье также есть хороший обзор литературы в этой области. Представляемый подход в целом следует их подходу: создана исполняемая спецификация на Promela согласно C-коду POK, но отличием служит то, что мы собираемся использовать методы проверки моделей (Model Checking) вместо доказательства теорем.

К верификации свойств модели ОС

Хотя мы все еще думаем о свойствах для проверки, на данном слайде мы предоставляем способ проверить правильность переключения разделов в модели. Согласно нашей модели, мы используем некоторые строковые константы (см. параметры pok_print в листинге 1). Поскольку мы знаем, к какому разделу принадлежит каждая строка, мы можем установить ожидаемое соответствие такой строки ее разделу, см.:

```
//string constants
#define P1T1_I_will_signal_semaphores 0
#define P1T1_pok_sem_signal_ret 1
#define P1T2_I_will_wait_for_the_semaphores 2
#define P1T2_pok_sem_wait_ret 3
#define P2T1_begin_of_task 4

//map data-partition
short partitionByDataIndex[5] = {
        PARTITION1,
        PARTITION1,
        PARTITION1,
        PARTITION1,
        PARTITION2
};
```

К верификации свойств модели ОС

Затем, поскольку мы знаем ожидаемое соответствие разделов и будем знать текущие разделы при работе модели, мы предоставляем следующий макрос проверки, см.:

```
//check if we are in the correct partition
inline checkPointer(expectedPartition, actualPartition) {
    if
        :: (expectedPartition != actualPartition) ->  {
            pointersOk = 0;
            printf("segmentation fault!\n");
        }
        :: else -> skip
    fi
}
```

Listing 12. A macro to check the safety of data strings.

И макрос checkPointer теперь используется в реализации системного вызова печати, см.:

```
inline print(string, param) {
 checkPointer(partitionByDataIndex[string], currentPartition);
 if
  ::(string == P1T1_I_will_signal_semaphores) ->
    printf("[%d] P1T1: I will signal semaphores\n", realTime);
  ::(string == P1T1_pok_sem_signal_ret) ->
    printf("[%d]  P1T1:  pok_sem_signal_ret  =  %d\n",  realTime,
param);
   ...
  ::else -> skip
 fi
}
```

Listing 13. A fragment of print syscall implementation.

К верификации свойств модели ОС

То есть для верифкации данной задачи процесс будет заключаться в проверке LTL формулы “всегда указатели корректны”

$$G(pointersOk)$$

где переменная pointersOk может быть изменена, когда строка запрашивается из неправильного раздела во время всех возможных прогонов модели.

К автоматическому построению модели на Promela из C кода клиентских процессов

Coccinelle:

- Средство для создания семантических патчей, одобренное сообществом Linux.
- Позволяет определять реакцию на конкретные синтаксические элементы AST C-программ, а также их последовательности.
- Внутри основана на CTL логике, что позволяет определять сложные темпоральные последовательности нужных операторов в коде.
- Кроме патчей, позволяет выполнять Ocaml или Python скрипты для проведения трансформации исходного кода.

К автоматическому построению модели на Promela из C кода клиентских процессов

```
//------------------------------
// rules for syscalls
//------------------------------

@print_rule@
expression E;
position pp;
@@
printf(E)@pp

@signal_rule@
identifier sid, i1;
position pp;
@@
i1 = pok_sem_signal(sid)@pp

@wait_rule@
identifier sem;
expression wait_value;
position pp;
@@
pok_sem_wait(sem, wait_value)@pp

@sleep_rule@
expression sleep_val;
position pp;
@@
pok_thread_sleep(sleep_val)@pp
```

```
//------------------------------
// reaction to a particular
// system call - scripts
//------------------------------
@script:python depends on print_rule@
ee <<  print_rule.E;
pos << print_rule.pp;
@@
fun = findFunByPos(pos[0].line, pos[0].file)
addTuple(fun, pos[0].line, (MySyscalls.PRINT, ee))
print("print on ->", ee, 'at', pos[0].line)

@script:python depends on signal_rule@
ssid << signal_rule.sid;
pos << signal_rule.pp;
@@
fun = findFunByPos(pos[0].line, pos[0].file)
addTuple(fun, pos[0].line, (MySyscalls.SEM_SIGNAL, ssid))
print("sem signal on ->", ssid, 'at', pos[0].line)

@script:python depends on wait_rule@
ssid << wait_rule.sem;
wval << wait_rule.wait_value;
pos << wait_rule.pp;
@@
fun = findFunByPos(pos[0].line, pos[0].file)
addTuple(fun, pos[0].line, (MySyscalls.SEM_WAIT, ssid, wval))
print("sem wait on ->", ssid, 'value', wval, 'at', pos[0].line)

@script:python depends on sleep_rule@
slval << sleep_rule.sleep_val;
pos << sleep_rule.pp;
@@
fun = findFunByPos(pos[0].line, pos[0].file)
addTuple(fun,pos[0].line, (MySyscalls.SLEEP, slval))
print("sleep on ->", slval, 'at', pos[0].line)
```

К автоматическому построению модели на Promela из C кода клиентских процессов

```
//finalize script

@finalize:python@
@@
print("fin!")
print("found functions:")
for key, val in map_file2fun.items():
    print (key, ':')
    for x in val:
        print (x)

print("generated processes:")

for fun in map_fun2loc:
    print("proctype %s(short myPartId; short myThreadId) {" % (fun))
    print("do\n::(osLive == 1) -> \natomic {")
    i = 0
    map = map_fun2syscalls[fun]
    for v in sorted(map.items()):
        //print(v[1])
        print(" if ::(currentPartition == myPartId \n  && currentThread == myThreadId && currentContext.IP == %d) -> \n {" % (i))
        type = v[1][0]
        param = v[1][1]
        if type == MySyscalls.SEM_WAIT:
            extra = v[1][2]
        if type == MySyscalls.PRINT:
            print("  pok_print(%s);" % (param.replace(':','').replace(' ','_').replace('\"','').replace('\\n','')))
        elif type == MySyscalls.SEM_SIGNAL:
            print("  pok_sem_signal(%s, currentContext.r0);" % (param))
        elif type == MySyscalls.SEM_WAIT:
            print("  pok_sem_wait(%s, %s, currentContext.r0);" % (param, extra))
        elif type == MySyscalls.SLEEP:
            print("  pok_delay(%s);" % (param))

        print("  currentContext.IP++;\n }")
```

Заключение

- В этом докладе были проанализированы назначение, состав и структура партицированной ОС реального времени.
- Обсужден возможный подход к созданию модели такой ОС на Promela.
- Предлагаемое решение позволяет перейти от сложного архитектурно-зависимого кода на языке C к общим моделям операционных систем, которые можно использовать в учебном процессе.
- Также, имея формализованную модель ОС, мы можем проверять свойства безопасности выполнения кода в партицированных системах с возможностью использования различных алгоритмов планирования.

Заключение

Возможные будущие шаги расширения:

- моделирование различных стратегий планирования;
- моделирование ARINC API;
- добавдение контрольных переменных, построение формул LTL и проверка модели;
- автомотическое создание моделей клиентских процессов по C-коду;
- многоядерные модели планирования;
- проверка простых киберфизических моделей, работающих в такой ОС.