О восстановлении программ из контрольной точки*

А.Ю. Поляков

В работе описаны два подхода к проблеме восстановления распределенных программ из контрольной точки. Предложен алгоритм восстановления взаимосвязей типа "родитель-потомок" и алгоритм принадлежности к группам и сеансам для набора процессов в рамках элементарной машины распределенной вычислительной системы. Предложен алгоритм координированного восстановления набора связанных процессов, перезапускаемых раздельно (на различных элементарных машинах или терминалах). Описанные подходы реализованы в системе создания контрольных точек DMTCP (Distributed MultiThreaded CheckPointing).

1. Введение

Распределенные вычислительные системы (ВС) - это важнейший вычислительный инструмент, который используется для проведения научных, инженерных и экономических расчетов [1]. Такие ВС являются большемасштабными, они состоят из сотен тысяч процессорных ядер и имеют производительности порядка PetaFLOPS [2]. Однако даже на таких высокопроизводительных системах многие современные задачи требуют для своего решения дни, недели и месяцы. Несмотря на высокий уровень развития элементной базы и схемотехники, аппаратные ресурсы распределенных ВС не являются абсолютно надежными. В связи с их большемасштабностью вероятность выхода из строя одной или нескольких составляющих становится достаточно высокой. Отказы процессоров, жестких дисков, сетевых адаптеров, кабелей и шин передачи данных могут повлечь за собой потерю значительного количества промежуточных вычислений, что приведет к снижению технико-экономической эффективности ВС. Таким образом, актуальной задачей является обеспечение отказоустойчивого выполнения программ на распределенных ВС.

Наиболее распространенным подходом к решению данной проблемы является создание контрольных точек (КТ) [3]. В процессе выполнения программы происходит периодическое сохранение ее состояния на надежный носитель данных. В случае отказа производится "откат" к ближайшей доступной контрольной точке, и работа возобновляется. При этом теряется незначительное количество промежуточных вычислений.

В данной работе описаны два подхода к проблеме восстановления распределенных программ из контрольной точки. Предложен алгоритм координированного восстановления набора связанных процессов, перезапускаемых раздельно (на различных элементарных машинах или терминалах). Разработан алгоритм восстановления взаимосвязей типа "родитель-потомок" и алгоритм принадлежности к группам и сеансам для набора процессов в рамках элементарной машины (ЭМ) распределенной ВС. Данные алгоритмы реализованы в программном пакете создания КТ DMTCP (Distributed MultiThreaded Check-Pointing) [4], который позволяет создавать КТ для последовательных, параллельных и распределенных программ в ОС GNU\Linux.

2. Классификация средств создания контрольных точек

Существует достаточно много средств создания КТ (ССКТ) [4-7], каждое из них имеет свои преимущества и недостатки. Рассмотрим несколько подходов к классификации ССКТ.

Существует две основные схемы взаимодействия ССКТ с защищаемой программой: *явная* и *прозрачная* (неявная). ССКТ, построенные на основе *явной схемы*, позволяют задать ограниченный набор информации, которую необходимо сохранить в КТ. Это позволяет снизить объем дискового ввода/вывода, т.е. значительно уменьшает накладные расходы таких ССКТ. Недостатком явной схемы является необходимость модификации исходного кода, что не позволяет

* Работа выполнена при поддержке РФФИ (гранты 08-07-00018, 08-07-00022, 08-08-00300, 09-07-00185, 09-07-12016, 09-07-13534, 09-07-90403) и Совета по грантам Президента РФ (грант НШ-2121.2008.9)

применять ее к программам, доступным только в бинарном виде. Кроме того, КТ могут создаваться только в моменты времени, определяемые программой и связанные с завершенностью определенного периода вычислений.

ССКТ, построенные на основе *прозрачной схемы*, выполняют сохранение КТ незаметно для программы, что обеспечивает простоту и универсальность их использования. Недостатком этой схемы является больший объем дискового ввода/вывода, так как сохраняется все пространство памяти.

По классам поддерживаемых программ ССКТ можно разделить на *сосредоточенные* и *распределенные*. Сосредоточенные ССКТ обеспечивают отказоустойчивость выполнения одного или нескольких процессов в рамках вычислительной машины. Распределенные ССКТ обычно строятся на базе сосредоточенных и позволяют выполнять создание КТ для распределенных и параллельных программ, что делает их важным инструментом организации функционирования ВС. Для создания распределенной КТ (РКТ) необходимо:

1) создать сосредоточенные КТ для всех процессов, входящих в состав распределенной программы (РП);
2) сохранить граф связей между процессами РП;
3) сохранить сообщения, которые были отправлены, но не доставлены на момент создания РКТ (такие сообщения также называют in-transit).

Для распределенных ССКТ различают *координированный* и *некоординированный* подходы. При создании РКТ каждый процесс РП сохраняет свое состояние в КТ. Целостной РКТ [3] называется набор из N локальных КТ, формирующих допустимое состояние программы. Такая РКТ может быть использована для восстановления программы после сбоя. При координированном подходе создание КТ происходит синхронно, что гарантирует целостность РКТ. При некоординированном подходе каждый процесс создает КТ независимо от других. Следовательно, при восстановлении необходимо выполнять поиск целостного состояния программы на основе набора независимых КТ. Это вносит дополнительные накладные расходы. Для некоординированного подхода существует опасность возникновения "эффекта домино", когда в процессе поиска целостного состояния происходит откат к начальному состоянию программы.

Распределенные ССКТ также можно разделить на *универсальные* и *MPI-ориентированные*. Первые позволяют создавать РКТ для любых распределенных и параллельных программ, в том числе для различных реализаций модели передачи сообщений (PVM, MPI). Что касается вторых, то существует несколько ССКТ, построенных на базе конкретных реализаций MPI. Например, OpenMPI [8], MVAPICH2 [9], LAM-MPI [10]. Все они используют ССКТ BLCR [5] для создания сосредоточенных КТ и реализуют собственные механизмы сохранения графа связей и транзитных сообщений.

Для создания КТ сосредоточенного процесса необходимо сохранить информацию о его состоянии. Это может быть реализовано на различных программных уровнях:

1. **Уровень операционной системы (ОС)**. Предусматривает сохранение содержимого пространства ядра и пространства пользователя для всех процессов ОС. Такой подход может быть реализован с использованием систем виртуализации (например, VMWare);
2. **Уровень ядра ОС**. Предусматривает внедрение дополнительных компонентов, позволяющих сохранить необходимую информацию: содержимое памяти конкретного процесса и состояние ядра, относящиеся к нему.
3. **Уровень системных библиотек.** Предусматривает сохранение содержимого памяти и состояния ядра с использованием средств, предоставляемых ОС для управления процессами.
4. **Прикладной уровень.** Предусматривает сохранение минимального объема информации, необходимого для восстановления каждой конкретной программы.

ССКТ уровней ОС, ядра и системных библиотек реализуются в рамках прозрачной схемы. Кроме того, некоторые ССКТ уровней ядра и системных библиотек предоставляют программе возможность влиять на процесс обеспечения отказоустойчивости, например, выбирать наиболее удобные моменты для создания КТ. Прикладной уровень предусматривает только явную схему.

Преимуществом первого уровня является простота реализации, а недостатком - значительный объем дискового ввода/вывода и отсутствие гибкости. Второй уровень позволяет получать прямой доступ к внутренним структурам ядра и памяти процесса и выполнять сохранение необходимой для восстановления информации при меньшем объеме ввода/вывода. Недостатком данного подхода является зависимость от изменений в ядре ОС (новые версии ядра Linux выходят в среднем с частотой раз в 3-4 месяца). Также данный подход требует привилегий суперпользователя для установки и управления, а ошибки, допущенные в программном обеспечении уровня ядра, приводят к нарушению работы всей ОС.

Третий уровень позволяет обеспечить создание КТ, не требуя при этом привилегий суперпользователя и не подвергая угрозе функционирование всей ОС. Однако при данном подходе невозможно осуществить прямой доступ к внутренним структурам ядра, которые описывают защищаемый процесс. Для этого требуется перехват и обработка системных вызовов.

На четвертом уровне сохраняется лишь содержимое буферов, которые явно указываются в программе.

3. Distributed MultiThreaded Checkpointing - DMTCP

Программный пакет DMTCP реализован на уровне системных библиотек и является универсальной координированной распределенной ССКТ. DMTCP разработан в Северо-западном университете (Northeastern University) США под руководством профессора Дж. Купермана.

Наиболее распространенной сосредоточенной ССКТ на данный момент является пакет BLCR. Кроме того, как было отмечено ранее, он используется во многих распределенных MPI-ориентированных ССКТ. Таблица 1 отражает сравнение ССКТ DMTCP и BLCR по поддерживаемым функциям ОС. BLCR используется для создания сосредоточенных КТ в нескольких MPI-ориентированных распределенных ССКТ.

Таблица 1. Функции, поддерживаемые ССКТ

Поддерживаемые компоненты ОС	DMTCP		BLCR	
	Полностью	Частично	Полностью	Частично
Обработка сигналов	X		X	
Сокеты	X		-	-
Многопоточные приложения	X		X	
Идентификаторы ресурсов ОС (процессы, группы, сессии)		**X**	**X**	
Именованные и неименованные каналы	X		X	
Открытые файлы	X		X	
Отображенные (mapped) файлы	X		X	
/proc файлы		X		X
Статически скомпилированные программы	**-**	**-**		**X**
Отлаживаемые программы		X	-	-

Из таблицы 1 видно, что DMTCP уступает BLCR по двум параметрам. Во-первых, нет поддержки статически скомпилированных программ, т.к. для перехвата системных вызовов используется "предзагрузка" служебной динамической библиотеки dmtcphijack.so. Однако данный пункт не полностью поддерживается и в BLCR. Во-вторых, отсутствует восстановление идентификаторов ресурсов ОС, таких как идентификаторы групп и сессий. В пространстве ядра в связи с прямым доступом к его внутренним структурам данная задача является более простой. В DMTCP (на уровне системных библиотек) была реализована частичная виртуализация идентификаторов процессов. В данной работе предложен алгоритм, позволяющий более полно восстанавливать идентификационную информацию. Он был интегрирован и используется в DMTCP в настоящее время.

На рисунке 1 показан запуск программы с применением DMTCP. В процессе ее работы автоматически осуществляется контроль над созданием новых процессов с использованием сис-

темного вызова *fork()*. Как было сказано ранее, DMTCP реализует координированное создание КТ. На каждую вычислительную группу создается один координатор (*dmtcp_coordinator*). Он может быть запущен явно, как показано на рисунке 2. Если при запуске программы (рисунок 1) процесс координатор не обнаружен, то он будет запущен автоматически.

```
host1$ dmtcp_checkpoint ./program1
```

Рис. 1. Запуск программы *program1* под управлением DMTCP на узле *host1*

```
host1$ dmtcp_coordinator
```

Рис. 2. Запуск процесса-координатора на узле *host1*

```
host1$ dmtcp_checkpoint ./program2
```

Рис. 3. Запуск программы *program2* под управлением DMTCP на узле *host1*

```
host2$ DMTCP_HOST="host1" dmtcp_checkpoint ./program3
```

Рис. 4. Запуск программы *program3* под управлением DMTCP на узле *host2*

Возможно создание РКТ для нескольких взаимодействующих программ, запускаемых с разных терминалов. Например, как показано на рисунках 1 и 3. В этом случае вспомогательный модуль DMTCP, интегрированный в каждую из программ, выполнит соединение с координатором.

Также возможно создание РКТ для процессов, работающих на разных узлах сети. Для этого необходимо указать через переменную окружения *DMTCP_HOST* адрес узла, на котором выполняется координатор. Так, на рисунке 4 показан запуск программы *program3*, которая подключается к вычислительному процессу, уже содержащему программы *program1* и *program2*.

Создание РКТ происходит следующим образом: каждый процесс сохраняет свое состояние в отдельном файле, а координатор формирует shell-скрипт, содержащий последовательность действий, необходимых для запуска вычислений из данной РКТ.

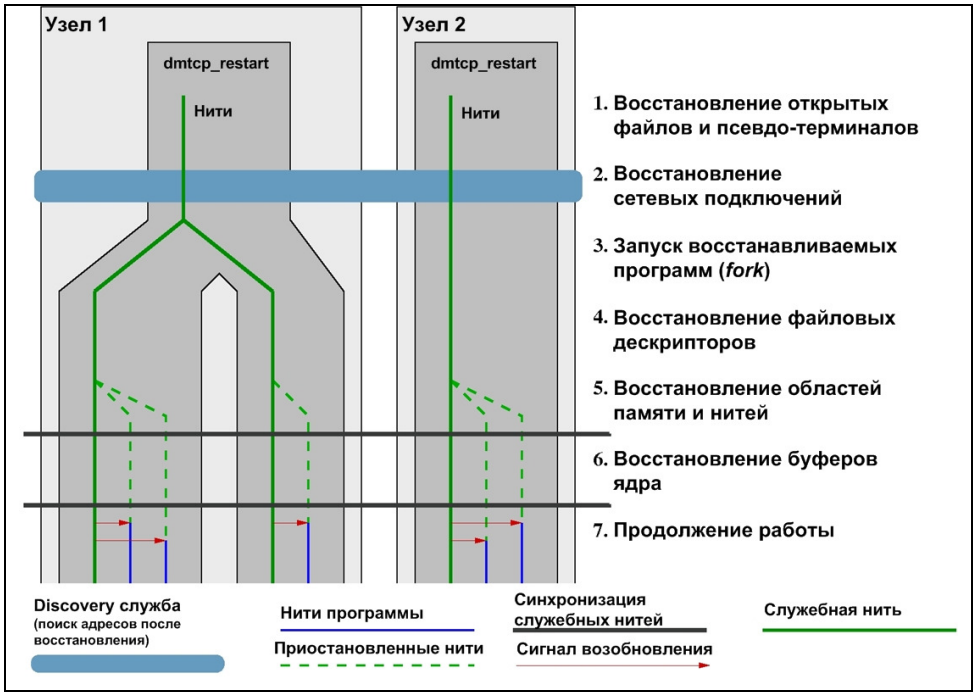

Рис. 5. Восстановление вычислений из РКТ DMTCP

Как показано на рисунке 5, на этапе восстановления на каждом узле запускается один служебный процесс, использующий РКТ для воссоздания компонентов программы (этап 3). Для синхронизации узлов используется этап восстановления сокетов (этап 2).

Недостаток данной схемы заключается в том, что процессы, выполнявшиеся на разных терминалах, будут перезапущены уже на одном. Например, DMTCP используется в качестве

основы для универсального реверсивного отладчика URDB [11]. Типичным сценарием применения URDB является подключение (attach) к уже выполняющейся программе и ее отладка. При восстановлении такой отладочной сессии необходимо сохранить принадлежность к разным терминалам, однако отсутствие средств синхронизации не позволяет этого сделать.

Рассмотрим другой пример: восстановление из контрольной точки группы процессов, распределенных по разным узлам сети. Процессы разбиты на подгруппы, не связанные между собой постоянными сетевыми соединениями. В этом случае барьер, образованный этапом 2 (Recreate and reconnect sockets), не является достаточным для синхронизации. Если одна подгруппа была запущена значительно раньше остальных, ее выполнение будет продолжено, а остальные подгруппы не будут иметь возможности возобновить работу.

Для устранения указанных недостатков был предложен дополнительный компонент схемы синхронизации, который представлен в данной работе.

4. Дополнительные компоненты схемы синхронизации

В DMTCP предусмотрен барьер, позволяющий синхронизировать восстановление из РКТ только для процессов, связанных постоянными сетевыми соединениями. Как было показано в разделе 3, существуют программы, для которых это условие не выполняется. Для устранения этого недостатка возникла необходимость расширения схемы синхронизации.

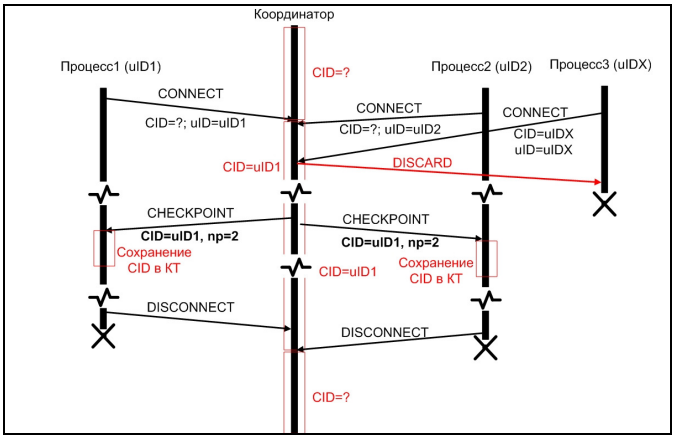

Рис. 6. Создание РКТ

Каждый процесс в DMTCP имеет уникальный идентификатор uID (*unique ID*), который формируется из трех компонент: <*хеш-код имени сетевого узла*>-<*PID*>-<*временная метка*>.

Координатор играет роль службы, предоставляющей сервис синхронизации. Его состояние подстраивается под выполняемые задачи и не сохраняется в РКТ. В качестве синхронизационного условия выбрано число процессов, принадлежащих вычислительной группе (ВГ) на момент создания контрольной точки. Как показано на рисунке 6, для идентификации ВГ (*CID – computational group ID*) используется uID процесса, который выполнил подключение первым. Если приходит запрос на подключение от другой ВГ (процесс 3 на рисунке 6), то оно отклоняется. На этапе создания РКТ координатор рассылает *CID* текущей ВГ и число ее участников (*np – number of process*). Эта информация сохраняется в каждой локальной КТ. При отключении последнего процесса из текущей ВГ координатор переходит в состояние *CID=?* и готов принимать новые запросы на услуги синхронизации от других ВГ.

На этапе восстановления (рисунок 7) процесс считывает *CID* и *np* из КТ и отправляет координатору при подключении. Если координатор не занят обслуживанием других заявок, он устанавливает параметр *CID* в значение, которое содержится в сообщений. Также запоминается количество клиентов, которое должно выполнить подключение до того, как можно будет продолжить вычислительных процесс.

Если подключение выполняет клиент, не имеющий *CID* или имеющий *CID*, который отличается от текущего, то такое соединение отклоняется.

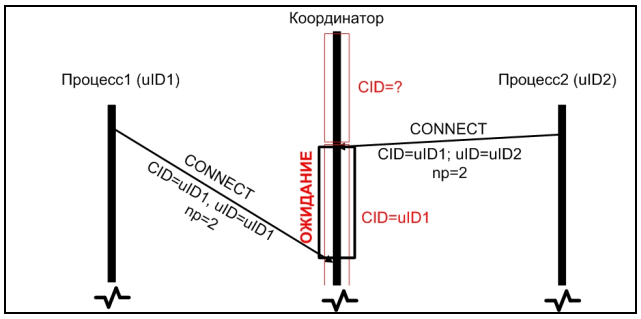

Рис. 7. Восстановление из РКТ

5. Алгоритм восстановления идентификационной информации

Как было сказано ранее, восстановление идентификационной информации на уровне системных библиотек затруднено отсутствием прямого доступа к внутренним структурам ядра ОС GNU\Linux. Необходима имитация процесса первоначального запуска программы. Рассмотрим подробнее восстанавливаемые идентификационные ресурсы.

5.1 Идентификационные ресурсы

В ОС GNU\Linux процесс описывается набором идентификаторов. Первый из них – идентификатор процесса PID (*process ID*). *PID* назначается при создании системными вызовами *fork()* или *vfork()* и используется для того, чтобы указать на процесс в ряде важных системных вызовов, таких как *kill()*, *ptrace()*, *setpriority()*, *waitpid()*.

Отношение родитель-потомок строится на основе *PID*. Процесс, выполнивший системный вызов *fork*, становится родителем созданного процесса. Для доступа к информации об идентификаторе родителя (*parent PID - PPID*) используется системный вызов *getppid*. Если процесс завершается, а потомки продолжают существование, их родителем становится системный процесс *init*, имеющий *PID=1*.

Каждый процесс принадлежит к одной и только одной сессии, для создания новой используется системный вызов *setsid*. Идентификатор сессии SID (*session ID*) равен идентификатору процесса-создателя (или лидера). Принадлежность к сессии наследуется потомком от родителя.

Каждый процесс принадлежит к одной и только одной группе. Если его идентификатор совпадает с идентификатором группы, то он называется ее лидером. Все процессы группы принадлежат одной и только одной сессии. Данные механизмы используются командными интерпретаторами при организации конвейеров, некоторыми отладчиками для управления отлаживаемыми программами и т.д.

5.2 Постановка задачи

Пусть имеется множество контрольных точек $C = \{c_i\}, i = 1...N$, каждая из которых однозначно соответствует восстанавливаемому процессу $p_i \in P$. КТ описывается четырьмя параметрами $c_i = (pid_i, ppid_i, sid_i, img_i)$, где: pid_i - уникальный идентификатор ($\forall i, j = 1...N, i \neq j, pid_i \neq pid_i$) в рамках ЭМ ВС; $ppid_i$ - идентификатор процесса, создавшего p_i через системный вызов *fork()*; sid_i - идентификатор сессии, если $pid_i = sid_i$, то КТ c_i содержит процесс-лидер сессии sid_i; img_i - сохраненное состояние процесса, необходимое для его перезапуска.

Обозначим через $s = \bigcup_{i=1}^{N} sid_i$ множество уникальных идентификаторов сеансов, к которым принадлежат процессы из P. Пусть $S = \{S_k\}, k = 1...|s|$ - множество сеансов, где

$S_k = \{c_i \mid c_i \in C, sid_i = s_k\}$ - подмножество КТ, содержащих процессы одного сеанса. Очевидно, что $C = \bigcup_{S_k \in S} S_k$ и $\forall k_1, k_2 = 1...|s|,\ k_1 \neq k_2,\ S_{k_1} \cap S_{k_2} = \varnothing$. Пусть также определено множество $R = \{c_i \mid \forall j = 1...N, j \neq i, ppid_i \neq pid_j\}$ независимых КТ.

Требуется, используя программный интерфейс ОС GNU/Linux, выполнить запуск процессов из контрольных точек так, чтобы восстановить их исходную иерархию и принадлежность к сеансам. При этом для изменения сеанса имеется только системный вызов *setsid()*, который позволяет процессу создать собственный сеанс, в котором он становится лидером.

5.3 Алгоритм восстановления иерархии процессов и сеансов

На вход алгоритма подается множество контрольных точек C. Алгоритм состоит из следующих шагов:

1. **Формирование отношений типа родитель-потомок**. Строится лес деревьев $T = \{T_l\}, l = 1...|R|$ (рисунок 8) , для которого определена функция однозначного соответствия f между КТ и узлами деревьев леса: $f = \{(t,c) \mid \exists l = 1...|R|, t \in T_l, c \in C, t \Leftrightarrow c\}$. Справедливы следующие утверждения:

1) $\forall T_l \in T$, если t - корень T_l, то $f(t) \in R$;

2) $\forall T_l \in T, \forall t_1, t_2 \in T_l$ то t_1 - непосредственный потомок $t_2 \Leftrightarrow \exists i, j : c_i = f(t_1), c_j = f(t_2)$ и $pid_i = ppid_j$.

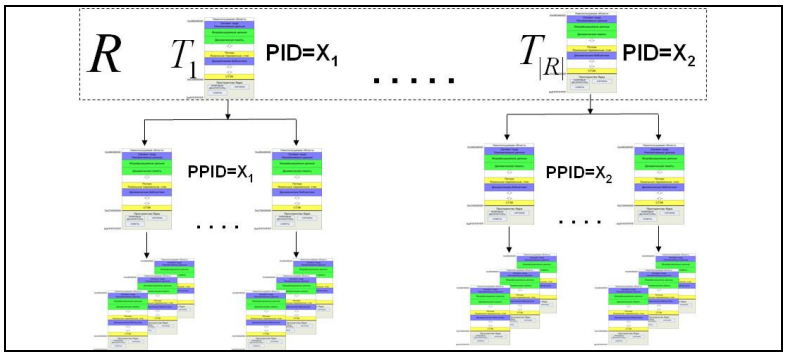

Рис. 8. Лес деревьев T, соответствующий восстанавливаемым КТ

2. **Построение метаинформации**. Для того чтобы восстанавливать принадлежность к одной сессии процессов, которым соответствуют узлы различных деревьев, выполняется построение метаинформации. Для каждого дерева $T_l \in T$ выполняется его обход в глубину. Для каждого обрабатываемого узла $t \in T_l$ определяется номер соответствующей ему КТ $i : c_i = f(t)$. Строится метаинформация, которая представляется в виде пары $(x^i{}_k, y^i{}_k)$, где $x^i{}_k \in x^i$, $x^i = \{x^i{}_k \mid x^i{}_k = sid_j,\ j = 1...N, f^{-1}(c_j) \in \{t\} \cup \{потомки\ t\}\}$,

$$y^i{}_k = \begin{cases} 1, & \exists c_j \in C : f^{-1}(c_j) \in \{t\} \cup \{потомки\ t\}\ и\ pid_j = x^i \\ 0, & иначе \end{cases}.$$

Второй компонент пары ($y^i{}_k$) указывает на наличие или отсутствие лидера сессии с идентификатором $x^i{}_k$.

На рисунке 9 показан пример сбора метаинформации. Рассмотрим узел X, которому соответствует метаинформация, состоящая из одной пары *(SID_3,0)*. X является листовым и не является лидером SID_3. Корень дерева (узел Y) содержит метаинформацию из четырех пар: *(SID_1,1)*, *(SID_2,1)*, *(SID_3,1)*, *(SID_4,1)*. Это означает, что на текущем и нижележащих уровнях дерева имеет-

ся четыре сессии с идентификаторами SID_1,SID_2,SID_3,SID_4, для каждой из них были найдены лидеры.

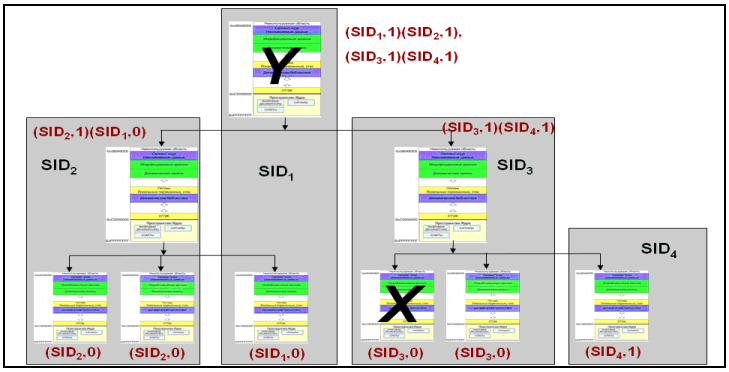

Рис. 9. Построение метаинформации

3. **Построение зависимостей между элементами** T. Каждому узлу $t \in T$ ставится в соответствие множество $Dep(t) = \varnothing$. Далее выполняется обработка метаинформации, соответствующей корням деревьев из леса T. Пусть $T_{M_1}, T_{M_2} \in T$, как показано на рисунке 10. Пусть t_1 - корень T_{M_1}, t_2 - корень T_{M_2}, i, j - соответствующие индексы КТ: $c_i = f(t_1)$, $c_j = f(t_2)$. Тогда T_{M_2} зависит от T_{M_1}, если $\exists k_1, k_2 : x^i{}_{k_1} = x^j{}_{k_2}$ & $y^j{}_{k_2} = 1$ & $y^i{}_{k_1} = 0$. В этом случае выполняется поиск лидера сессии $x^i{}_{k_1}$ - $\tilde{t} = f^{-1}(c_l) : pid_l = sid_l = x^i{}_{k_1}$ и модифицируется соответствующее множество $Dep(\tilde{t}) = Dep(\tilde{t}) \cup t_2$

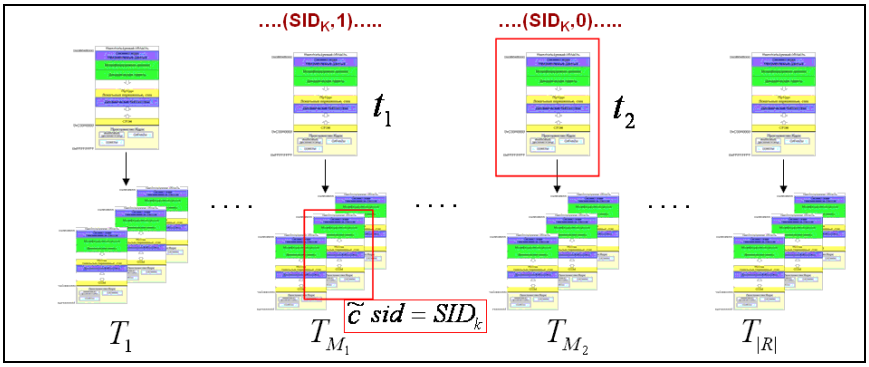

Рис. 10. Построение зависимостей между деревьями леса T

На рисунке 10 показано, что метаинформация корня дерева T_{M_2} указывает на отсутствие лидера сессии SID_k в T_{M_2}. Пара (SID_k,1), соответствующая корню дерева T_{M_1}, указывает на наличие узла $\tilde{t}$ и контрольной точки $\tilde{c} = f(\tilde{t})$, содержащей лидера SID_k. Тогда считаем, что дерево T_{M_2} зависит от узла $\tilde{t}$. То есть при запуске процессов из КТ сначала должна быть восстановлена КТ, соответствующая $\tilde{c}$, и создана новая сессия SID_k, а также все процессы дерева T_{M_2}.

4. Для всех независимых деревьев ($\forall T_l \in T : t_l = 0$) выполняется восстановление исходной структуры процессов с использованием *fork()* и *setsid()* по алгоритму, приведенному на рисунке 11.

```
procedure restore(t,psid)
    i = k : (c_k == f(t));
    if pid_i != sid_i then
        for t1 ← childs(t) do fork(); restore(f(t1), psid) done
        start(f(t));
    else
        for t1 ← childs(t) do
            j = k : (c_k == f(t1));
            if sid_j <> pid_i then
                if fork() = 0 then restore(f(t1),psid) end if
             end if
        done
        psid = setsid();
        // Восстановление всех деревьев, зависящих от узла t
        for t1 ← Dep(t) do
             if fork() = 0 then
                if fork() = 0 then restore(f(t1), psid)
                else exit(0) end if  // теперь p – наследник init
             end if
         done
         for t1 ← childs(t) do
             j = k : (c_k == f(t1));
             if sid_j = pid_i then
                if fork() = 0 then restore(f(t1),psid) end if
                end if
         done
         start(f(t))
    end if
procedure end
```

Рис. 11. Алгоритм запуска программ из КТ

5.4 Восстановление принадлежности к группам

Процессы внутри одного сеанса могут менять свои группы в произвольное время. Для этого используется системный вызов *setpgid()*, которому передается идентификатор новой группы и процесса, для которого выполняется смена группы. Если какой-то из аргументов нулевой, то считается, что подразумевается текущий. Восстановление принадлежности к группам выполняется после того, как все процессы запущены и им сопоставлены сеансы. Если текущий процесс является лидером группы, то он создает ее с помощью вызова *setpgid(0,0)*. Остальные члены группы выполняют попытки подключиться к ней до тех пор, пока не истечет таймаут (рисунок 12)

```
int ret = 1, i = 0;
struct timespec ts = {0,100000};
// Trial Timeout = 2 seconds
while( ret && ((float)(i*ts.tv_nsec) / 1E9) < 2.0 ){
    ret = setpgid(0,_gid);
    if( ret ){
        nanosleep(&ts,NULL);
    }
    i++;
}
```

Рис. 12. Алгоритм восстановления принадлежности к группе для процесса, не являющегося ее лидером

6. Экспериментальные данные

На рисунке 13 (а,б) показаны структуры двух программ, для которых были созданы КТ с применением ССКТ DMTCP.

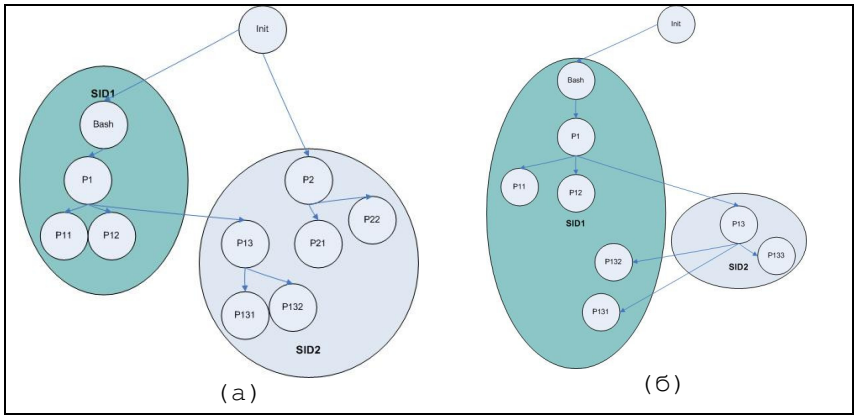

Рис. 13. Структуры тестовых программ

На рисунке 14 (а,б) показаны отношения между восстановленными процессами без применения разработанного алгоритма. Как видно из рисунка 14 (а), все процессы принадлежат одной сессии, а процесс *p2*, который должен быть потомком *init(PID=1),* является потомком *p1*. На рисунке 14 (б) мы также видим, что принадлежность к сессиям не восстанавливается.

На рисунке 15 (а,б) приведены результаты восстановления из КТ с применением предложенного алгоритма для соответствующих программ рисунка 13. Рассмотрим более подробно рисунок 15 (а). Очевидно, что процессы *p13*, *p131*, *p132*, *p2*, *p21*, *p22* принадлежат одной сессии, лидером которой является *p13*, как это и должно быть. Остальные процессы принадлежат сессии командного интерпретатора. Элемент *p2* восстановлен, как наследник *init* (*PID=1*).

На рисунке 15 (б) процессы *p13*, *p133* находятся в новой сессии, в которой *p13* является лидером. При этом потомки *p13 – p131* и *p132* остались в сессии командного интерпретатора, как это было в исходной программе.

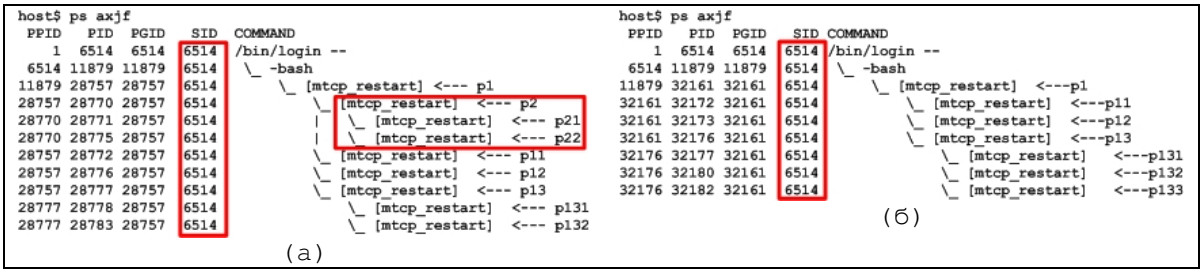

Рис. 14. Восстановление с неполным учетом идентификационной информации

(а) – для программы на рисунке 13(а); (б) – для программы на рисунке 13(б)

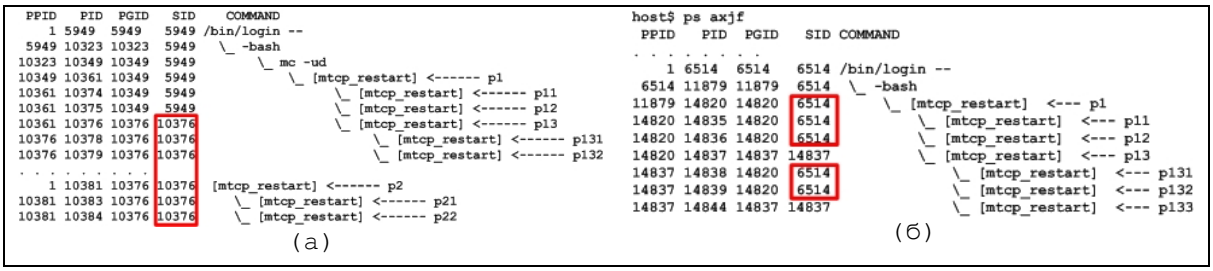

Рис. 15. Полное восстановление идентификационной информации

(а) – для программы на рисунке 13(а); (б) – для программы на рисунке 13(б)

На рисунке 16 (а) показана принадлежность к группам процессов исходной программы, для которой создавались КТ. Программа состоит из пяти процессов, которые принадлежат трем группам с идентификаторами 21226, 21231, 21239. При восстановлении из КТ исходными средствами DMTCP (рисунок 16 (б)) все компоненты остаются в группе, соответствующей некоторой внешней программе. Применение предложенного алгоритма, как показано на рисунке 16 (в), позволяет полностью восстановить исходную программу.

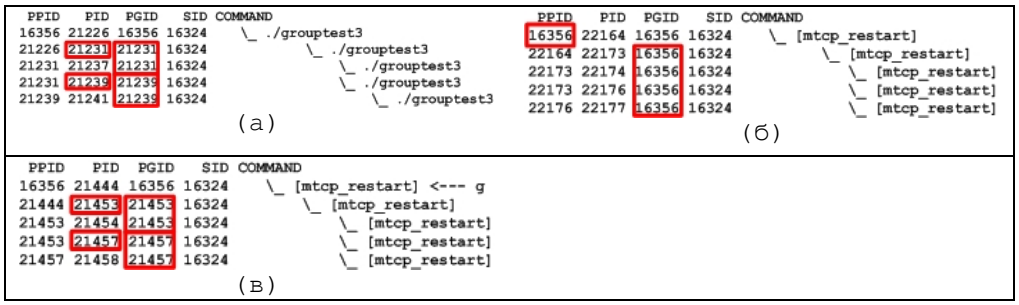

Рис. 16. Восстановление принадлежность к группам

(а) – исходная программа; (б) – восстановление алгоритмом DMTCP;

(в) - восстановление предложенным алгоритмом

На рисунке 17 показано восстановление тестовой программы. Она состоит из двух процессов, взаимодействующих через механизмы IPC. Запуск компонентов выполнен с двух различных терминалов: *pts/0* и *pts/2*. До реализации предложенного расширения схемы синхронизации подобное восстановление исходными средствами DMTCP было невозможно.

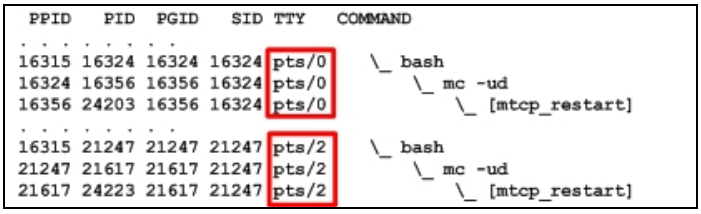

Рис. 17. Восстановление процессов на разных терминалах

7. Заключение

В работе были рассмотрены подходы к восстановлению программ из распределенных контрольных точек, реализованные в ССКТ DMTCP. Предложен алгоритм восстановления отношений родитель-потомок и алгоритм принадлежности к сессиям и группам с использованием стандартного механизма системных вызовов ОС GNU\Linux. Так как DMTCP реализован на уровне системных библиотек, он не имеет прямого доступа к внутренним структурам ядра. Следовательно, для восстановления указанных отношений между процессами требуется имитация основных шагов их запуска. Для этого выполняется построение древовидной структуры, отражающей родственные отношения между узлами. Далее происходит сбор метаинформации, содержащей описание принадлежности к сессиям. Разработанный алгоритм позволяет расширить диапазон программ, поддерживаемых DMTCP.

Расширена схема синхронизации, используемая в DMTCP: добавлен новый барьер, позволяющий выполнять восстановление процессов из РКТ на различных терминалах, расположенных на одном или разных узлах сети. Это позволяет использовать DMTCP для восстановления из РКТ программ, которые не связаны постоянными сетевыми соединениями. Предложенная доработка также необходима для организации реверсивной отладки, построенной на базе DMTCP.

Литература

1. Хорошевский В.Г. Архитектура вычислительных систем. – М.: МГТУ им. Н.Э. Баумана, 2008 . 520 с.
2. TOP500 supercomputer site [Электронный ресурс].- Режим доступа: http://www.top500.org/ . — Загл. с экрана. — яз. англ.
3. Elnozahy E. N., Alvisi L.,Wang Y.M.,Johnson D.B. A survey of rollback-recovery protocols in message-passing systems // ACM Computing Surveys . Vol. 34, No 3, 2002 . pp. 375-408.
4. J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop // Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS'09) . IEEE Press, 2009.
5. Paul H. Hargrove and Jason C. Duell Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters In Proceedings of SciDAC 2006: June 2006.
6. Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical report 1346, University of Wisconsin, Madison, Wisconsin, April 1997.
7. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In Proc. of the USENIX Winter 1995 Technical Conference, pages 213–323, 1995.
8. J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE Computer Society, March 2007.
9. Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. Parallel Processing, Jan 2006.
10. Dewolfs, D., Broeckhove, J., Sunderam, V., Fagg, G. "FT-MPI, Fault-Tolerant Metacomputing and Generic Name Services: A Case Study," Lecture Notes in Computer Science, Springer Berlin / Heidelberg, ICL-UT-06-14, Vol. 4192, Number 2006, pp. 133-140, 2006.
11. Ana Maria Visan, Artem Polyakov, Praveen S. Solanki, Kapil Arya, Tyler Denniston, Gene Cooperman Temporal Debugging using URDB .- Режим доступа: http://arxiv.org/abs/0910.5046v1