INFORMATION SYSTEMS IN MANAGEMENT

Information Systems in Management (2016) Vol. 5 (2) 237−247

ELIMINATION OF TASK STARVATION IN CONFLICTLESS SCHEDULING CONCEPT

MATEUSZ SMOLIŃSKI

Institute of Information Technology, Lodz University of Technology

New concept of conflictless task scheduling is an alternative approach to existing solutions in concurrency. Conflictless task scheduling includes data structures and algorithm that prevents occurrence of resource conflict between tasks executed in parallel. The range of applications the conflictless task scheduling includes different environments like transactions processing in database management systems, scheduling of processes or threads in operating systems or business processes management. Task scheduling without any resource conflicts is dedicated to high contention of limited resources environments and its algorithm can be implemented in modern GPU. This paper presents concept of local task scheduling without resources conflicts occurrence, discusses features of new approach and focuses on problem of task starvation. Elimination of task starvation is included in conflictless task scheduling concept, detailed explanation are contained in this paper.

Keywords: Resource conflict elimination, conflict free task schedule, deadlock avoidance, concurrency control, mutual exclusion, transaction processing, OLTP

1. Introduction

Resource conflict occurs in multitasking environment when many tasks executed in parallel use the same instance of resource and at least one of them perform operation, which changes this resource state. Elimination of negative phenomenon as resource conflicts can be performed in many ways, for example using competitive or cooperative concurrency. All concurrency methods

synchronize global resource allocation. In competitive concurrency various synchronization mechanism can be used (i.e. semaphores, locks or barriers), however tasks do not communicate each other to determine order of resource allocation. A significant problem in competitive concurrency is proper selection and correct application of synchronization mechanism for access to global resources. Another problem is to choose fine or coarse grain strategy in resource synchronization. In cooperative concurrency tasks communication is possible, also dedicated structures can be used to store environment state. Supervision mechanism bases on task communication or dedicated structures that support prevention of unsafe environment state [3, 5].

The elimination of resource conflict between executed tasks ensures avoidance of deadlock. But there is no universal synchronization method that guarantees task execution without resource conflict in any environment. Even when task specification is known, selection and use proper concurrency method are not always apparent to programmer.

In next chapters concept of universal solution of conflictless task scheduling will be presented. This concept bases on dedicated structures and algorithms, that guarantee fairness and liveliness in task processing without resource conflicts. New concept can be used in various environments of task processing that meet fixed assumptions. In further discussion a task starvation problem in conflictless task scheduling has been examined.

2. Assumptions for conflictless task processing environment

Task processing environment has own specifics resulting from the number of task sources and characteristics of requested tasks. Any requested task is single unit of work and is defined by sequence of operations and resources, which are required to finish its execution. Resources required by task can be local or global. Resource instance is local when is used by only one task, in other case resource is global.

New concept of conflictless task scheduling is dedicated to any high contention environment that meets assumptions for requested tasks:

- all resources required by requested task exists,
- set of all global resources required by requested task is known before its execution begins,
- many requested tasks can be executed in parallel,
- execution time for task is not known and its finish time is not limited,
- each task requires minimum one global resource,
- task are independent and its execution order is not fixed,
- task execution is not depended on interaction with external objects,

- tasks are equivalent and there is no task priorities,
- number of global resources is limited and any global resource has only one instance.

In applied task definition single task with long sequence of operations can be divided into many smaller sub-tasks, which have to be requested by task source one by another. This technique can be also used for interactive task to isolate subtasks that do not require any global resource. The range of application of proposed conflictless task scheduling includes various environments like scheduling of processes or threads in operating systems, transaction processing systems or business process environments. Regardless of the type of task in high-contention environments each requested task reveals a specific representation of all required global resources before its execution begin and reports finish of execution. Even task execution interrupted by error has to be reported to resume other waiting conflicted tasks.

3. Task representation and resource conflict detection

The concept of conflictless task scheduling requires special task resources representation model which is used to fast verification existence of resource conflict between tasks. In presented concept all global resources required by task are represented in its binary identifiers *IRW* and *IR*, they are granted to task by central resource controller. In those binary identifiers single bit represents only one global resource, the length of resources identifiers is limited by number of global resources. This task resource representation is scalable [4]. Identifier *IRW* represents all global resources used by task that are read or written in its sequence of operations. However *IR* represents all global resources used by task that are only read. As opposite to write read operation do not change global resource state. Besides of resources identifiers each requested task t_k has assigned a logical time T_k number. Order of logical time values represents sequence of granting binary resource identifiers by central resources controller to tasks.
Detection of resource conflict between two tasks using their binary resource identifiers requires to check simple condition:

$$(IRW_i \ and \ IRW_j) \ xor \ (IR_i \ and \ IR_j) \neq 0 \quad (1)$$

If above condition is satisfied then exists at least one resource conflict between tasks. This means that two task can not be executed in parallel and its execution order must be determined. Logical time values can be used to determine sequence of conflicted task execution, longer waiting task can be executed first.

4. Structures for conflictless task schedules generation

Controlling tasks execution without resource conflicts requires to maintain two sets: active task set R that are executed and waiting task set W that are suspended. In any n-th point in time active task set R^n and waiting task set W^n do not have any common elements $W^n \cap R^n = Ø$. Effective task processing requires maximization of cardinality of the active task set R^n in any point of time. Resource conflict detection between newly requested task and any active task and also between suspended tasks is supported by dedicated structures task classes C and conflict array M. State of this structures may vary over time, so state of task class C_k in n-th point of time will be marked $C^n{}_k$ and respectively state of conflict array in n-th point of time represents M^n. Single task class groups all tasks with the same values of resources binary identifiers:

$$C_k = \{t_i: IRW_i = IRW_k \wedge IR_i = IR_k\} \tag{2}$$

In n-th point in time task class $C^n{}_k$ groups only supended tasks. If task from class C_k is executed (this task is included in set R^n) then class $C^n{}_k$ is marked as active, but active task $t \in C_k \cap R^n$ not belongs to $C^n{}_k \subseteq C_k$. Therefore in n-th point in time suspended task set $W^n = C^n{}_1 \cup \ldots \cup C^n{}_N$. For any active class $C^n{}_k$ there has to be determined task class C^*_{nk} which includes oldest suspended task t^*_k that has resource conflict with active task $t_k \in C_k \cap R$. Each non-empty task class has its own FIFO queue, which determines order of execution of their suspended tasks. Each FIFO queue is assigned to other resource group represented by task class, which has at least one task suspended. This multi queues approach can be treated as alternative to use many synchronization mechanisms with coordination of emptying their queues for waiting tasks. In n-th point of time cardinality of task class $n(C^n{}_k)$ show number of suspended tasks that are located in class $C^n{}_k$ queue. According to prepared conflictless schedule the task class $C^n{}_k$ queue leaves only single the oldest waiting task, only if condition is not satisfied:

$$IRW_k = IR_k \tag{3}$$

If condition 3 is satisfied then all tasks from task class C_k queue can be executed in parallel. The conflict array M is another structure, which stores conflicts detected between task classes bases on condition 1. If in two dimension array value $m_{ij} = 0$ then there is no conflict between tasks belonging to different classes C_i and C_j. Conflict array M is always symmetric $M = M^T$, so only half of it should be calculated. Dimension of array M changes every time when new task class is added because in n-th point of time $M^n = g(R^n, C^n{}_1, C^n{}_2, ..., C^n{}_N)$. If dimension of array M changes then all conflict values must be calculated between new added and any other existing task class. If there is no resource

conflict between the new class and any other active class, then execution of task from new class has begin and new class becomes active. Otherwise new task is suspended in class FIFO queue and is waiting to resume according to prepared conflictless schedule of task execution.

5. Preparation of adaptive conflictless schedule

The conflicless task processing includes determination the task execution order for waiting tasks, which ensures parallel task execution without global resource conflicts. Appropriate set of tasks is fixed by conflicless schedule, which is adequate to task environment state. In n-th point in time number of conflictless schedules to prepare is limited by cardinality of active task set R^n because time of task execution is not known. The conflictless schedule should be prepared to each situation that execution of active task $t_k \in R^n$ finishes and requires to determine set S^n_k of suspended tasks that can be executed in parallel without resource conflicts. Any task set S^n_k has to fulfill all conditions:

- all tasks belonging to S^n_k has conflict with finished task t_k,
- no resource conflict between task belonging to S^n_k and any active task from set $R^n / \{t_k\}$,
- no resource conflict between task belonging to S^n_k and the oldest suspended task t^*_k for any active task from $R^n \cup \{t_k\}$ or when resource conflict exists between task t_i belonging to S^n_k and oldest suspended task t^*_k for any active task from $R^n \cup \{t_k\}$ and $T^n_i < T^{*n}_k$,
- no resource conflict between any two tasks that belong to S^n_k,

Above restrictions can cause that non-empty set S^n_k does not exist and when execution of active task t_k is finished no other suspended task from task classes will be resumed to start its execution.

In situation when various sets S^n_k exists arbitration rule should be used to choose conflictless one schedule S^{*n}_k, in example should be chosen most numerous set of suspended task with the lowest sum of logical time values for task belonging to S^n_k. Each preparation of conflictless schedule in n-th point of time bases on detection of resource conflicts between many selected task couples stored in conflict array M^n. Resource conflict between tasks from different classes can be simply detected by reading a single value from conflict array M i.e. resource conflict between tasks $t_x \in C_i$ and $t_x \in C_j$ exist only if $m_{ij} \neq 0$. The conflictless schedule has to adapt to task class cardinality and logical time of its longest waiting task, therefore it is a adaptive conflictless schedule which can be applied only in fixed state of task processing environment.

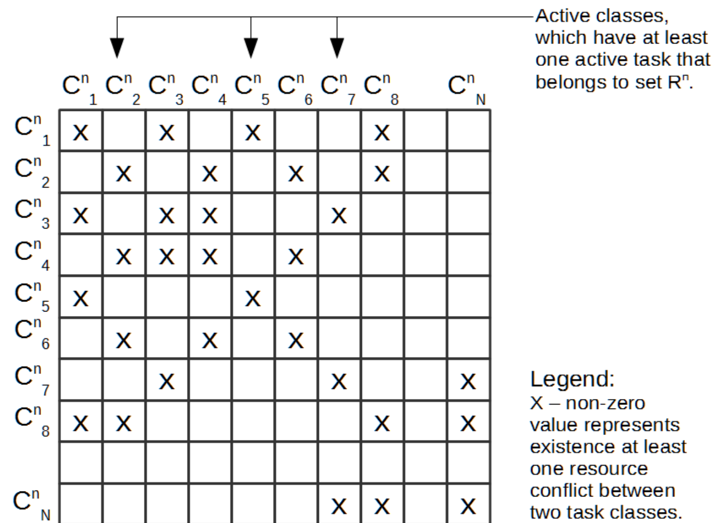

Figure 1. Example of conflict array in selected n-th point in time

Figure 1 presents example state of conflict array in fixed *n*-th point in time for number *N* of existing task classes. The presented conflict array contains additionally indication of active classes according to elements in active task set R^n.

6. Elimination of task starvation in adaptive conflictless schedule preparation

Conflictless schedule is prepared step by step, which allows to adapt to changing conditions in task processing environment. Presented concept of conflictless scheduling determines set of tasks that execution can be started when fixed active task finish its execution. The multipath conflictless task scheduling is result of the assumption, that task execution time is not known. Lack of long-term schedule could cause a repeating pattern in task class selection, which can exclude resuming of task from other classes. If task has never been included in prepared schedule it waits infinitely for execution and task starvation problem occurs. Concept of conflicless task scheduling that allows occurrences of task starvation problem is not correct, because it does not ensure fairness in task execution.

The main role in elimination of task starvation problem in conflictless task scheduling concept is to often change members of active class set. However this rule is not sufficient to eliminate the task starvation problem. Therefore another rule takes into consideration logical time of waiting tasks, in conflictless schedule preparation longer waiting tasks are preferred to begin its execution first.

Algorithm of determination of conflicless schedule S^n_k in n-th point of time for situation when active task $t_k \in C_k \cap R^n$ execution is finished:

1. Determine collection Q^n_k of all non-empty task classes C^n_i that has conflict with C^n_k, where for $i \neq k$ $m_{ik} \neq 0$ in conflict array.
2. If task class C^n_k queue is not empty $n(C^n_k) > 0$, add class C^n_k to determined class collection Q^n_k.
3. Among task classes from collection Q^n_k find the task t^*_k with the lowest value of logical time T^*_k, this task has the oldest waiting time in head of all class queues from collection Q^n_k. Class that include task t^*_k is marked C^{*n}_k.
4. Remove from class collection Q^n_k each class that has at least one resource conflict with any other active class than C^n_k.
5. Remove from class collection Q^n_k each class C^n_i that has at least one resource conflict with recently determined C^{*n}_j for any active class (including C^n_k only if C^{*n}_k is included in collection Q^n_k) that fulfills the conditions $C^n_i \neq C^{*n}_j$ and $T^n_i > T^{*n}_j$.
6. If C^{*n}_k does not belongs to class collection Q^n_k then go to step 7, otherwise go to step 8.
7. In class collection Q^n_k find the most numerous subset of classes without resource conflict and the lowest sum of logical time values for head task belonging to queues of classes from selected subset. Go to step 10.
8. If $C^n_k \neq C^{*n}_k$ then go to step 9 otherwise go to step 10.
9. In class collection Q^n_k find the most numerous subset of classes without conflict and including C^{*n}_k. When many collections exist choose that one, that has lowest sum of logical time values for head task from class queues.
10. Add to S^n_k each next task from queue of any class from collection Q^n_k. In situation when for C^n_j condition $IRW_j = IR_j$ is also satisfied add to S^n_k all tasks waiting in queue of class C^n_j.

Presented algorithm uses conflict array structure with additional information like cardinality of any task class or logical time of head task in all class queues. Presented algorithm can be run in many instances, which allows for fixed n-th point in time parallel preparation of many conflictless schedules S^n_k. Each one conflictless schedule S^n_k is prepared for other case, when execution one of active task $t \in R^n$ is finished. In practice only one from all prepared schedules will be used. Theoretically all calculations required to determine conflictless schedule S^n_k can be performed ahead before active task $t \in R^n$ execution finish, than all required calculations for conflictless schedule preparation have to be performed in environment isolated from task execution. If all calculation related with preparation conflictless schedule S_k will be completed before finish of active task $t \in C_k$ then there will be no additional delay in task processing.

Validity of determined conflictless schedules is time limited. Anyone of presented situations can cause useless of earlier determined conflictless schedules:

- change in the active task set,
- change in task class cardinality,
- adding a new task class.

Conflictless schedules have to be prepared often and quickly. It requires dedicated environment for manage structures and algorithm executions to prepare conflictless schedules, which is discussed in chapter 7.

Example situation of conflictless task processing in environment with high contention resource utilization in *n*-th point in time presents state of conflict array on figure 2. Additionally for any detected resource conflict between non-empty task classes logical time values for classes are presented on figure 2, each one is determined by the longest waiting task in class queue in *n*-th point of time. In presented on figure 2 situation three conflictless schedules are determined using proposed algorithm. Only one from prepared conflictless schedule $S*^n_2$, $S*^n_5$, $S*^n_7$ will be used depending on which active task from R^n will be finished first. All prepared conflictless schedules $S*^n_2$, $S*^n_5$, $S*^n_7$ are valid only in *n*-th point of time.

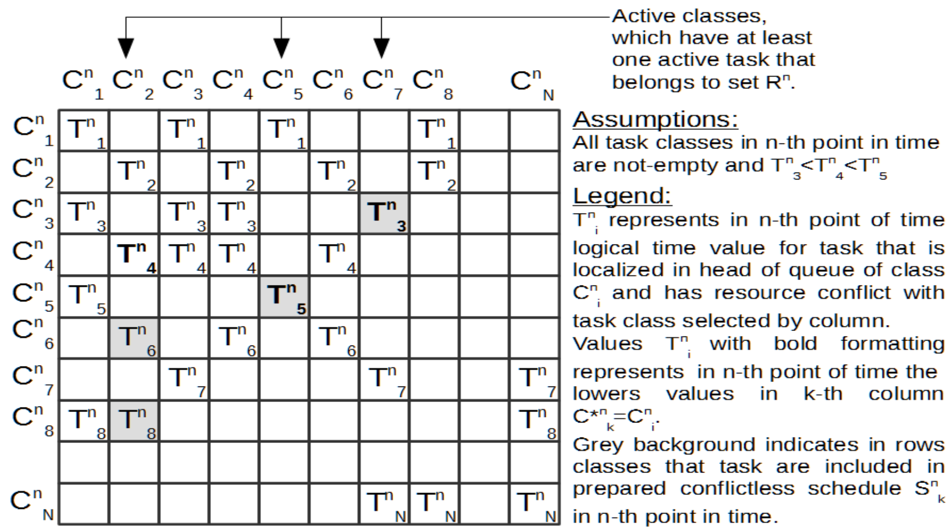

Figure 2. Conflictless schedules for example conflict array in selected *n*-th point of time

Conflictless schedule $S*^n_2$ includes at least one task from class C^n_6 and C^n_8. If condition 3 is satisfied for class C^n_6 or C^n_8, then all tasks waiting in this class queue can be assigned to $S*_2$. Class $C^n_4 = C*^n_2$ not belongs to $S*^n_2$ conflictless

schedule because it has resource conflict with $C^n_3 = C*^n_7$ and $T^n_4 > T^n_3$. This prevents starvation of tasks from class C^n_3 which is included in many cycles in WFG (Wait For Graph) presented in figure 3. Conflict array M is a representation of WFG for task classes.

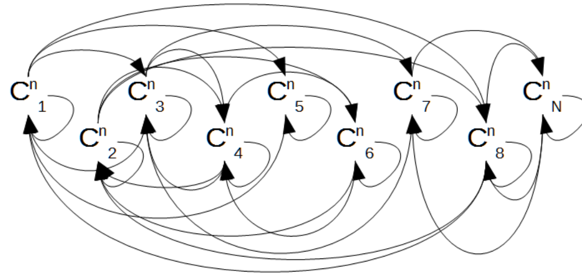

Figure 3. Wait for graph for task classes in selected n-th point in time

The conflictless schedule $S*^n_7$ includes at least one task from class C^n_3 and C^n_N. If condition 3 is satisfied for class C^n_3 or C^n_N, then all tasks waiting in class queue are assigned to schedule $S*^n_7$. When active task execution from class C_5 is finished then according to prepared schedule $S*^n_5$ another task from class C^n_5 begins execution. If condition 3 is satisfied then all suspended task can be removed from class C^n_5 queue and executed. The starvation of suspended task occurs if its execution will be stared in finite time [5]. Preparation of adaptive conflictless schedule bases on analysis of resource conflicts between not-empty classes and includes also logical time of waiting tasks. This ensures that set of active classes changes that execution of task from all classes will be resumed. This avoids occurrence of task starvation problem.

The preparation of conflictless schedule for selected time period requires preparation of adaptive conflictless schedules many times according to presented algorithm. Current environment state is taking into account when adaptive conflictless schedules are prepared in selected points of time. Those time points are determined by execution finish of any active task. Figure 4 presents usage of prepared schedules for example conflict array form figure 4 in $n + 1$, $n + 2$ and $n + 3$ time points. Those points of time are determined by each execution finish of active task. In $n + 1$ point of time task from class C_5 finish its execution and according to adaptive conflictless schedule another task from class C^{n+1}_5 can be executed. However finish this task does not cause execution next task from class C_5 which counteracts of task starvation for conflicted task from conflicted class C^{n+4}_1 (execution of task from C_1 is started later in $n + 7$ point in time). Identical situation occurs in $n + 5$ point of time, when execution of task form class C_6 is finished, which prevents starvation of tasks from classes C^{n+5}_2 and C^{n+5}_4.

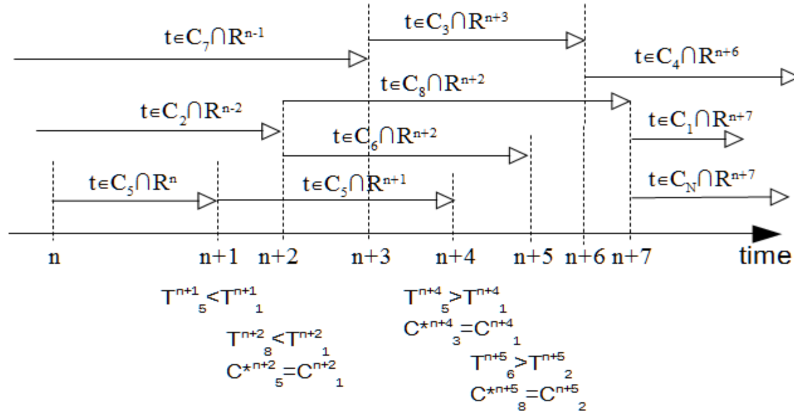

Figure 4. Example of conflictless schedule in period of time with environment conditions

Analysis of schedule from figure 4 show that for any point of time new adaptive conflictless schedules should be prepared, because different environment conditions cause that earlier determined schedules are useless. Frequent and parallel preparation of conflictless schedule need isolated computing environment, which resources are not used by tasks. Efficient task execution required preparation of all conflictless schedules before one of active tasks finishes its execution. Due to the high degree of parallelization and frequency of conflictless schedule preparation as isolated computing environment for preparation of adaptive conflicless schedules was proposed modern GPU.

7. Effective preparation of conflictless schedule by GPU

Modern GPU are isolated computing environment with own memory and multiple processing units, that architecture is categorized as SIMD in Flynn Taxonomy [2]. The GPU can be used to prepare and store a conflict array with so far detected resources conflicts between task classes. Any request of new type of task causes new task class creation and update of conflict array. There is no need in conflictless schedule preparation to transfer conflict array from GPU memory.

Using prepared algorithm GPU allows to prepare many conflictless schedule at the same time, each one for situation when other active task finish its execution. It is possible because during conflictless schedule preparation values stored in conflict array did not change. For conflicless scheduling implementation on GPU are recommended OpenCL or CUDA technology [1].

8. Conclusions

The proposed task scheduling concept is novel due to data model of task resources representation and methods of resource conflict detection between tasks, structures like task classes and conflict array and algorithms preparing schedule without resource conflict between tasks. Conflictless scheduling is designed to environment with limited number of global resources, that are acessed by task in high contention manner. The conflictless schedule eliminates resource conflict between executed tasks, therefore deadlock is not possible. Conflictless scheduling can be alternative to use other synchronization mechanism. The correctness of developed scheduling algorithm requires that it has to ensure liveness and fairness for tasks. Liveness requires that prepared conflictless schedule eliminate task starvation problem, which was discussed in this paper. In proposed task scheduling concept it was provided by using logical time for tasks and check its values in adaptive conflicless schedule preparation.

REFERENCES

[1] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., & Dongarra, J. (2012) *From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing*, 38 (8), 391–407.

[2] Flynn M.J., Rudd R. W. (1996) *Parallel architectures*, ACM Computing Surveys, Volume 28, Issue 1, 67–70.

[3] Silberschatz A., Galvin P.B., Gagne G. (2012) *Operating system concepts*, Wiley John Sons, 9th edition.

[4] Smoliński M. (2010) *Rigorous history of distributed transaction execution with systolic array support,* XXXI ISAT conference, Information Systems Architecture and Technology New Developments in Web-Age Information Systems, Oficyna Wydawcznicza PW, 235–254.

[5] Tanenbaum, A., Bos H. (2014) *Modern operating systems*. Prentice Hall, 4th edition.