Reliable, Fair and Efficient Concurrent Software with Dynamic Allocation of Identical Resources

Claude Kaiser and Jean-François Pradat-Peyre

Conservatoire National des Arts et Métiers
Laboratoire CEDRIC
kaiser@cnam.fr, peyre@cnam.fr

1 Introduction

1.1 Presentation

Dynamic allocation of a class of identical resources, such as memory slots, is considered for concurrent software. Resources are allocated at run time to processes in a context prone to deadlock [Coffman 71] : the resources are allocated to one process at a time, a process may hold allocated resources while awaiting assignment of others, no resource can be forcibly removed from a process holding it. Safety of concurrent software, i.e. absence of deadlock, can be obtained by avoidance policies, called in this paper beforehand precaution and dynamic prevention. Both rely on the banker's algorithm [Habermann 69] based on a priori process claims and on service postponement when a request can lead to deadlock once processes proceed requesting resources up to their claims. Fairness of concurrent software, i.e. starvation avoidance cannot always be based on a pure FIFO service of requests since it may block all the processes and therefore introduce another source of deadlock.

We present a safe and fair implementation which rely on the concurrent semantic of protected objects in Ada and which has been proven with the model of colored Petri nets. However, safety and fairness may reduce concurrency and therefore the efficiency of concurrent software. It may also imply a different service for small claim and large claim processes.

We consider the case of large systems and how to partition the resources and/or the processes and construct a composite allocator, in order to limit the concurrency reduction or to limit the differences in quality of service. This partition is tuned by the results of simulations.

The resource utilization is simulated with different policies. Global criteria are computed such as a concurrency factor and a resource retention ratio. The simulation results show the effectiveness of the partionning for some case studies.

1.2 Concurrent software

Concurrency is getting more and more present in applications with the extend of synchronized multimedia man machine interfaces and of client-server intranet architectures supporting concurrent servers. Developing reliable concurrent software for these applications is therefore a growing challenge. Especially one must

be able to verify the reliability of programs when buying some on the shelf, or when performing acceptance trials for some piece of software realized by sub-contractors.

Recent programming languages such as Ada95 and Java provide features for concurrency and distribution and they favor the development of concurrent and distributed software. These object oriented languages allow dynamic creation of objects, threads or tasks, which request and remove dynamically resources at their creation and deletion. Threads and tasks may also request and remove resources during their execution. Dynamic allocation of resources is also playing a central role in object oriented distributed applications, which allow a variable and evolving number of objects, tasks, threads, processes, stations, sites or servers.

This arms software developer's with powerful but dangerous tools. Systems composed of cooperating agents can be concisely specified, but the resulting program may contain concurrency errors such as deadlock or starvation. Most of these errors cannot be prevented statically at compile time since they will depend of the system behavior resulting from hazardous requests and dynamic allocation of resources. Thus considerable care must be taken to provide reliable dynamic allocation policies such as deadlock prevention or avoidance policy. Moreover one must take care that the usual two stage development, which first deals with avoiding deadlock and second deals with introducing fairness, does not reintroduce deadlock when coping with fairness.

1.3 Resource allocation

When resources are allocated at run time, the allocation policy may consider various aspects such as:

- the state of the resources,
- the nature of the requests, such as size, duration of use, individual or co-ordinated requests (for example a communication channel may need to be requested by both end users ; another example is an allocation for a client presenting a request which has to be authenticated by a third party ; in object oriented concurrent applications the cooperation of agents introduces some dependencies between the requests or the releases.),
- the priority of the requesting client or of the group of clients,
- the history of individual clients, of the group of clients, of resource allocation,
- the declaration of total resource claims and of the behavior of future resource requests,
- postponing the calling client until the resource is available or just returning a denial of service,
- controlling that the client respects the assumption of resource allocation for a finite period (detecting faulty clients when their deadline is over),
- preempting or not the client allotment.

The allocation policy must take care of clients requests, of quality of service (efficiency of allocation management, availability of resources during a given

time period) ; it must also guarantee global invariant properties of the system behavior such as absence of deadlocked clients, absence of infinitely postponed clients and absence of clients never releasing their allocated resources (or simply not releasing the resources in the contractual delay).

2 Reliable allocation of identical resources

Let a pool of m identical resources to be allocated to a set of n independent processes.

A process P_i $(1 \leq i \leq n)$ is represented by :

- $claim(i)$: the claim of process P_i, i.e. the maximum number of resources it ultimately needs,
- $grant(i)$: the total number of resources currently granted to process P_i,
- $dist(i)$: the distance of process P_i to its claim, i.e. $dist(i) = claim(i) - grant(i)$,
- $add(i)$: the current number of additional resources requested by P_i,
- $rank(i)$: the rank of process P_i when it is necessary to totally order the set of processes

An allocation state is **realizable** if the following conditions hold :

$$(R1) \ : \ \forall i \in 1..n, 0 \leq dist(i) \leq claim(i) \leq m$$

$$(R2) \ : \ \sum_{i=1..n} grant(i) \leq m$$

Let $Stock$ be the current number of available resources; $Stock$ is defined by $Stock = m - \sum_{i=1..n} grant(i)$.

We deal only with realizable states and we assume that a process cannot be forced to release a resource until it finishes (no resource preemption) but also that it releases all its allotted resources in due time.

For convenience, we say that a process P_i starts a new transaction when $grant(i) = 0$ and $add(i) \geq 1$. It ends its transaction when it releases all its resources ($grant(i) = 0$ and $add(i) = 0$). A process which has started a transaction is said to be transactive until it ends it.

An allocation state is **safe** if it cannot ever lead to deadlock.

2.1 Deadlock avoidance (safety) by prevention or by precaution

Deadlock avoidance rely on dynamic prevention or on beforehand precautions. Dynamic prevention controls the allocation and postpones request to avoid otherwise possible deadlock situations. Beforehand precautions make deadlock situations impossible to be reached for any resource request.

Dynamic prevention A policy of dynamic prevention of deadlock is the banker's algorithm [Dijkstra 67, Habermann 69, Holt 71, Holt 72].

The key of the banker's algorithm is to control the allocation such that, in every allocation state, the allocator stock is large enough to allow a feasible survival behavior of the allocator. In this survival behavior, the allocator would satisfy the claims of all the transactive processes in sequential order, refraining any concurrency.

More formally, given a list of n independent processes $Pj(1 \leq j \leq n)$, an allocation state is safe iff there exists at least one permutation of this list for which the following condition holds :

$$(R3) : \forall j \in 1..n, dist(j) \leq stock + \Sigma_{k|rank(k)<rank(j)} grant(k)^1$$

If the allocation is controlled such that the system evolves from safe state to safe state, deadlock is prevented.

Remarks Let us recall some known results [Habermann 69].

1. Let $available\_stock = min_{j \in 1..n}(stock + \Sigma_{k|rank(k)<rank(j)} grant(k) - dist(j))$. If the state is safe, $available\_stock$ is non negative. Then any process that makes a request that is lower than $available\_stock$ can be served safely, since the system remains in a safe state after the service of that request.
2. If a state is safe, any resource allocation $add(i)$ to a process P_i such as $dist(i) \leq stock$ is safe since it leads to a new safe state where $dist(i) - add(i) \leq stock - add(i)$ with $rank(i) = 1$.
3. If (R3) holds for a sublist of s processes ($s < n$) and if (R3) never holds for any of the $n - s$ remaining processes added to the list at rank $s + 1$, then no permutation of the n processes will hold (R3) for the whole list and the state is not safe. It is not necessary to check some other permutation of the sublist.
4. Suppose that the system is in a safe state and consider a new resource allocation $add(i)$ to a process P_i; this leads to a new state. Suppose now that there exists a permutation where P_i is placed at rank h. The new state is safe as soon as (R3) holds for $rank = 1..h$.
5. If a state is safe, adding a process with $claim < m$ and $grant = 0$ leads to another safe state. If a state is safe, a process releasing resources leads to another safe state.
6. An efficient ordering of transactive processes is to order them by increasing values of distances to their claims. (This is possible when the allocator has to deal with one class of resources only).
7. Combining the result above leads to an efficient algorithm for testing the request of P_i :
   - let the transactive processes of a safe state be ordered by increasing distances;
   - remove P_i from that list and change the data relative to P_i;

[1] where $rank(i)$ denotes the rank of the process P_i in the considered permutation

- insert P_i with its new value in the list; let $rank(i)$ be the new rank of P_i
- the new state is safe iff (R3) holds from $rank = 1$ to $rank = rank(i)$;
- from point 4, the new state is safe as soon as P_i can use the revised stock of $rank(j)$ such as $1 \leq rank(j) \leq rank(i)$; this leads to check first whether $dist(i) \leq stock + \Sigma_{k|rank(k)<rank(j)} grant(k)$.

Beforehand precautions A policy of beforehand precaution has been given by R. Holt [Holt 71]. Any realizable system state is safe and deadlock is not possible if even in the worst set of allotments there is enough resources left in stock to allow at least one process to finish and if this happy ending for that one process will afterwards allow the other processes to finish as well.

$$(R4) : \sum_{i=1..n} (claim(i) - 1) < m$$

Note that at least $m - \sum_{i=1..n}(claim(i) - 1)$ processes will never be postponed.

If postponed processes are served FIFO, allocation is fair and this service will not block all processes when postponing them since (R4) is such there is always a process non postponed. (Nota : (R4) can be derived from (R3))

3 Fair allocation

3.1 A reliable dynamic allocation can be unfair

Example 1 Consider an isoclaim application, i.e. where all claims are equal. Processes with large requests may be always overtaken by processes with smaller requests if the stock is never large enough.

$$n = 6; m = 18; CLAIM = (4, 4, 4, 4, 4, 4)$$

at t_0 : $GRANT = (3, 3, 3, 3, 3, 2); DIST = (1, 1, 1, 1, 1, 2); stock = 1$;
at t_1 : P_6 requests two more resources; $add(6) = 2$; the resulting state is not safe ; P_6 is postponed;
at t_2 : P_1 requests one more resource; it is served; $stock = 0$;
at t_3 : P_2, P_3, P_4, P_5 request all one more resource ; all are postponed ; $add(2) = add(3) = add(4) = add(5) = 1$;
at t_4 : P_1 releases one resource; $stock = 1$; P_2 is served; $stock = 0$ again. $GRANT = (3, 4, 3, 3, 3, 2); DIST = (1, 0, 1, 1, 1, 2); stock = 0$.
at t_5 : (resp. t_6 and then t_7) : P_1 releases one resource; $stock = 1$; P_3 (resp. P_4 and then P_5) is served; $stock = 0$ again. Note that P_1 has released all its claim and $grant(1) = 0$ at t_7.
at t_8 : (resp. t_{10} and then t_{12}) P_1 requests one resource; it is postponed; $add(1) = 1$.
at t_9 : (resp. t_{11} and then t_{13}) P_2 (resp. P_3 and then P_4) releases one resource and P_1 is served.
at t_{14} : P_5 releases one resource. The allocation state is the same as in t_1. P_6 is still starving.

Example 2 Consider now an heteroclaim application, i.e. where all claims are not equal. Processes with large claims can be prevented from being served when requests of processes with smaller claims are present anew, even if these processes have previously released their total claim in one operation in due time.

$$n = 5;\ m = 6;\ CLAIM = (6, 5, 2, 2, 2)$$

at t_0 $GRANT = (1, 1, 1, 1, 1)$; $DIST = (5, 4, 1, 1, 1)$; $stock = 1$;

at t_1 P_2 requests one resource; the resulting state is not safe; P_2 is postponed.

at t_2 P_3, P_4, P_5 requests one resource; P_3 only is served; $stock = 0$. P_4 and P_5 are postponed.

at t_3 P_3 releases all its resources; $stock = 2$; P_4 and P_5 can be served; $stock = 0$.

at t_4 P_3 starts a new transaction and requests one resource; it is postponed.

at t_5 P_4 releases all its resources; $stock = 2$; P_2 cannot be served; P_3 can be served; $stock = 1$.

at t_6 P_4 starts a new transaction and requests one resource; it is served.

at t_7 P_5 releases all resources. stock = 2; P_2 can still not be served.

at t_8 P_5 requests one resource and is served. The allocation state is the same as in t_1. $GRANT = (1, 1, 1, 1, 1)$; $DIST = (5, 4, 1, 1, 1)$; $stock = 1$; P_2 is still postponed ; P_2 is starving. If P_1 had also made a request, it would also had been in starvation.

3.2 A FIFO allocation ruins the safety of a dynamic prevention policy

Example 3 Consider an heteroclaim application. A FIFO service leads to deadlock while $stock = 1$.

$$n = 3;\ m = 4;\ CLAIM = (4, 2, 1)$$

at t_0 the allocation state is : $GRANT = (2, 1, 1)$; $DIST = (2, 1, 0)$; $stock = 0$.

at t_1 P_1 requests one more resource; P_1 is blocked.

at t_2 P_2 requests also one more resource; P_2 is queued after P_1.

at t_3 P_3 releases all its claim, i.e. one resource. The allocator examines whether P_1 can be served. The state with $GRANT = (3, 1, 0)$ is not safe since it may lead to deadlock if P_1 and P_3 request both one more resource before releasing any; thus the request of P_1 is denied. And as the service is pure FIFO, the request of P_2 is not considered although servicing it would lead to a safe state.

at t_4 P_3 requests one resource; it is queued after P_2. The system is deadlocked with $stock = 1$. Recall that servicing P_2 (in a non FIFO service) would lead to a safe state.

Example 4 Consider an isoclaim application. A FIFO service leads to deadlock while $stock = 2$.

$$n = 3;\ m = 6;\ CLAIM = (4, 4, 4);\ \text{all claims are identical}$$

at t_0 $GRANT = (0, 2, 4)$; $DIST = (4, 2, 0)$; $stock = 0$;
at t_1 P_1 requests one more resource; P_1 is blocked.
at t_2 P_2 requests also one more resource; P_2 is queued after P_1.
at t_3 P_3 releases two resources, $stock = 2$. P_1 cannot be served since state $GRANT = (1, 2, 2)$ is not safe.
at t_4 P_3 requests one resource; it is queued after P_2. The system is deadlocked with $stock = 2$. Recall that servicing P_2 or P_3 (in a non FIFO service) would lead to a safe state.

3.3 Fairness of dynamic prevention

Deadlock prevention forbids a pure FIFO service of postponed processes since it relies on the potential of some transactive processes to finish their transaction and then to give back their resources. The basic idea is to serve FIFO as long as it does not contradicts deadlock prevention. In a safe state, the leading transactive process, the first in the list of transactive processes ordered by increasing distances, is supposed to be never denied of service. This property has to be held on especially when the leading process loses its position after giving back some of its resources or all of them (terminating its transaction), and when the new leading process has to be privileged.

Thus when resources are released, the postponed processes have priority over new incoming requests and they are served in a FIFO order so long as the service leads to a safe state. If not, the FIFO service is stopped. However, if the currently leading process is still postponed, it is served (this is always possible from the banker' algorithm).

However a permanent flow of processes becoming the leading process by turns should not indefinitely overtake the first postponed process. Thus :

- (A1) a process should not keep indefinitely its allotted resources,
- (A2) a process starting a new transaction should not start with a smaller distance that already postponed processes.

Processes must be compelled to respect this assumption (A2) when starting a new transaction, either in delaying the new transaction as long as its initial distance is lower than the distance of processes postponed by the allocator, or in allowing momentary to this new transaction an initial distance that is the maximal distance of the postponed processes (distance inheritance).

Let us examine the effects of this fairness policy to the previous examples.

In example 1, an isoclaim system :

at t_5 : no process is served since P_6 cannot be served and since the leading process P_2 is not postponed; $stock = 1$.
at t_6 : stock = 2 and P_6 is served. Fairness is respected.

In example 2, an heteroclaim system :

at t_3 : P_4 only is served ; P_5 remains postponed since the leading process is now P_4 ; stock = 1.
at t_4 : P_3 is delayed (or inherits of dist(2) = 4) ; obviates recurrent small transactions.
at t_5 : stock = 3 ; P_5 becomes the leading process and is served ; stock = 2.
at t_6 : P_4 is delayed (or inherits of dist(2) = 4) ; obviates recurrent small transactions.
at t_7 : stock = 4 ; P_2 can now be served ; P_3 and P_4 are reactivated (or back to dist(3) = dist(4) = 2) and they are immediately served; $stock = 1$.

Fairness is respected again.

In example 3 : at t_3, the leading process is P_2 which is therefore served. There is no deadlock.

In example 4 : at t_3, the leading process is P_2 which is therefore served. There is no deadlock.

3.4 A fair dynamic prevention algorithm for isoclaim systems

We consider isoclaim configurations which happen often when there is no a priori knowledge available about the distribution of requests. In this case the algorithm of fair dynamic prevention is simplified.

Suppose that all processes have the same claim c ($\forall i, claim(i) = c$).

If for the leading process, named X_l below, (R3) holds then $dist(X_l) = stock$ which is same as $dist(X_l) + grant(X_l) = stock + grant(X_l)$ which is same as $c = stock + grant(X_l)$

As for all processes $dist(i) = c$ then $dist(i) = stock + grant(X_l)$ and (R3) holds for all processes.

In this isoclaim system it is sufficient that the allocator always keeps enough resources to satisfy any request of the leading process.

A resource request is accepted when it is issued by the leading process, or when there remains enough resources left for that leading process once the request has been granted to another process.

Non accepted requests are postponed and they are examined anew when resources are released. This examination of postponed processes is done before accepting new requests.

Note that, for the sake of deadlock prevention, a postponed request shall not block all later requests, since one of these later request may be issued by the currently leading process which therefore must not be blocked. This is the case when the leading client is running. This is also the case for postponed processes since, indeed, when the leading process releases some resources and therefore no longer holds the max allotment, the new leading process may be one of the postponed processes. This forbids a pure FIFO service of requests

Fairness is obtained by applying a FIFO service when not in contradiction with the deadlock prevention policy. Deadlock is impossible if the leading process is never blocked and is allowed to overtake the other process which are FIFO served. If we observe that this leading process will in turn end its transaction and release its resources, thus this release will settle another leading process which must also be allowed to overtake the FIFO service. This recurrent settlement of successive leading processes ends when the oldest postponed process becomes itself the leading process. There is no opportunity for a starting transaction to overcome a postponed one since all initial distances are equal.

The fair algorithm and its proof are derived from these observations. A requesting process is always served when it is a leading process ; if not, it is yet served, partially or fully, if it leaves enough resources in the pool, once served, for allocating the leading process up to its maximum claim. If the requesting process is only partially served, it is postponed. Each time resources are released, these resources are allocated in priority to postponed processes (fortunately this is the concurrency model for Ada protected objects). The currently leading task, if postponed, is served and if all resources are not saved for future requests of the leading task, the remaining resources are allocated to the first postponed process until its request is fully served.

Safety and fairness have been formally proven with a colored Petri net model [Kaiser 97] using techniques detailled in [Barkaoui 98].

4 Large systems. Taking advantage of scale for partionning the resources and for structuring a composite allocator

4.1 Cost of safety for large systems with a unique allocator

Let a pool of m identical resources to be dynamically allocated to n processes and consider the effects of scale. Let a unique allocator of resources be installed.

To prevent deadlock, it uses dynamic prevention or, if there is enough resources, beforehand precaution. However respecting safety has a cost which may be too high in some situations.

For example, suppose an overload situation which lead to a burst of requests. Suppose a current state where the stock is zero and where one process, which claim was one, is running using its unique resource while all other $n-1$ processes are postponed because they request more resources. Naturally the state is safe and some of the postponed processes have a distance of one. In this state, deadlock prevention costs a maximum in resources being locked up since $m-1$ resources are stuck on blocked processes and costs also a maximum in reactivity since the number of concurrently running processes is one instead of n.

The larger m and n, the heavier the cost. Since processes waiting for resources retain also resources and contribute to reduce the stock, the probability of blocking new requests is certainly higher when the stock is small, i.e. when the postponed queue is large ; and therefore this high probability of blocking

results in making the chain larger. Note that in such an overloading situation it is preferable to react quickly in order to resorb the overload. If postponed processes are to be served according to a priority order, this order cannot always be respected since it could bring back deadlock situations as shown for fairness, and here also it might be necessary to serve first successive currently leading processes if they are postponed. The response of urgent transactions could be delayed a lot. A first objective, when seeking after efficiency, is to limit this concurrency reduction as well as the related resource retention and therefore to break this lengthy chain of mutually obstructing processes.

Respecting fairness with a pseudo FIFO service, as shown before, may also introduce an additional cost. Processes with large claims may wait for a long time before being safely served and if a fairness policy is implemented, they may refrain small claim processes to get resources if their requests are posterior in the queue. The response time of small transactions could be augmented too much for the application. Another objective, when seeking after efficiency, is to introduce some kind of weaker fairness which allows small transactions to finish before large ones but which prevents them to overtake the large ones too many times.

4.2 Structuring composite scheme for efficiency of allocation

In the case of large numbers of processes and resources, it is necessary to improve the efficiency of allocation since avoiding deadlock and providing fairness may introduce high costs especially when the system is overloaded.

Several approaches may be considered for reducing the response time of bursting processes and therefore the cost of safety and fairness. All install a composite allocation scheme which partition the process set and/or the resource set. Process priority, process response time or urgency (deadline), rate of unutilized resources, rate of potentially concurrent processes as well as efficiency of allocation for a given distribution of request arrivals and removals, may help choosing one of the composition.

We are investigating an approach based on the examination of the application specificity and requirements, on the measurement of its resource usage and on comparisons by simulation of possible composition scheme and various tuning of a composition.

4.3 Evaluation criteria

Several factors can be recorded each time the allocator is called (serving or postponing a request, releasing a request and possibly serving postponed requests). Corresponding mean values are computed at the end of the experimentation.

Concurrency factor CF

This factor records the current number of processes that run concurrently.

$CF = 1$ when the processes run sequentially and $CF = n$ when all n processes are concurrent.

Resource retention ratio RF

This ratio records the resources not in use since their recipient process is postponed or since the allocator has denied their allocation for safety.

$RF = 0$ if no requestor processes are postponed (also $CF = n$)

$RF = [\sum_{i|add(i)>0} grant(i) + Stock]/m$, if there exists i such that $add(i) > 0$.

4.4 Case studies

Example 5 Let us examine the case of an isoclaim system. Consider a set of $n = 100$ processes which share a pool of $m = 200$ resources. All processes have an identical claim of $c = 4$. This configuration may correspond to telecommunication systems. Suppose a unique allocator of 200 resources has been installed. It uses fair dynamic prevention since there is not enough resources for beforehand precaution.

However respecting safety and fairness may have a high cost as in the following scenario.

Suppose for example that during normal conditions 53 processes have been granted of 3 resources each, which leaves a stock of 41. Let a burst of activities occur and create an overload situation. Thus the 47 remaining processes all start their transactions with a request of 1 resource each. Responding to the overload involves also that the 53 normal processes request one more resource each for a while before releasing it and proceeding again with their loan of 3. Once served, each bursting process makes some primary computation, then requests 2 more resources for its final computation.

At bursting time, only 40 bursting processes are served, this leaves a stock of 1 and a queue of 7 bursting processes (each having a distance of 4, a grant of 0 and a request of 1) ; the queue is soon augmented by 52 normal processes (each having a distance of 1, a grant of 3 and a request of 1) and the stock becomes null. From now on, only 41 processes run concurrently.

When the 40 bursting processes have concurrently finished their primary computation, they become postponed (each having a distance of 4, a grant of 1 and a request of 2). Thus only 1 process, a normal one, is running. The FIFO queue of postponed processes contains 7 bursting processes preceeding 52 normal processes proceeding 40 bursting processes.

After this peak of inactivity, the number of running processes starts growing very slowly as the processes finish their transactions.

When this scenario is simulated, it is caracterized by a mean resource retention ratio is of 61% and a mean concurrency factor of 18% during a total execution time of 1213 time units.

Several approaches may be considered for reducing this cost of fairness. All install a composite allocation and expect taking advantage of the small value of claims and of the short duration of bursting transactions for maintaining a large concurrency factor and for reducing the length of the postponed queue. Let us examine some approaches.

A first approach consists in dividing the resource service for the nt transactive processes in two parts. An allocator of 200 resources with fair dynamic

prevention or fair beforehand precaution serves a subset of processes, called the fortunate processes and a waiting-room holds the others, called the expecting processes. Only a maximum of nf transactive processes can be fortunate at a given moment. Any process starting a new transaction when the number of fortunate processes is less than nf will be accepted immediately as a fortunate one; if not, it becomes expecting and it has to wait until one of the current fortunate transaction finishes. The number of expecting processes is $ne = max(0, nt - nf)$. This policy aims to limits the number of resources retained by postponed processes and to avoid a long chain of processes waiting for resources. An urgent transaction, if any, can be placed ahead of the process queue in the waiting-room and then waits only until the end of the first finishing fortunate transaction.

The scenario has been simulated for this first approach with fair dynamic prevention with $nf = 82$ and with fair beforehand precaution with $nf = 66$ and $nf = 50$ (For beforehand precaution, nf is at most 66 since $66 * (c - 1) < 200$ and when $nf = 50$ the fortunate process are never postponed). The results show the importance of preserving a high concurrency factor.

nf	allocation policy	waiting-room	resource retention ratio (mean)	concurrency factor	total execution time
100	fair dynamic prevention	no	61%	18	1213
82	fair dynamic prevention	yes	26%	37	608
66	fair beforehand precaution	yes	13%	51	407
50	fair beforehand precaution	yes	8%	37	610

Fig. 1. Results of the simulation

Example 6 Let us examine now the case of an heteroclaim system. The implementations of distributed object oriented systems such as Guide [Hagimont 94] have shown the coexistence of several size of memory chunks allocated to objects and a distribution of objects with a peak of small objects and another peak of large objects. Buffers for data communication may contain small texts, large files or very large set of image pixels.

When an overload occurs, if there is a safe but unfair dynamic allocation, processes with large claims may wait for a long time before being served since they may be overtaken by a flow of processes with small claims. Furthermore, if a fair policy is implemented, the large claim processes may refrain the small claim processes to get their resources and to execute within a small response time. This shows that it is difficult in both cases to provide an equal quality of

service to all processes.

A composite allocation may be an approach for avoiding a too large difference in the service.

Consider a concurrent application where 60 processes have a claim of 11 and realize short transactions for which the response time is important (they are considered as small clients) and where 8 processes have a claim of 101 and a long transaction (these are considered as large clients).

In normal working conditions, there is an observed mean of 10 small clients and 6 large ones. This leads to provide a stock of 700 resources and to allocate them by dynamic prevention (i.e. without fairness).

Let now consider at time $t = 0$ a load of 8 large clients starting a transaction with an initial request of 50 each. Then each client has a cyclic behavior in which it requests 40 resources for a while before releasing them.

At time $t = 1$ a flow of 60 small clients starts, 15 by 15 with a lag of 1 between them. These clients request 10 resources for 2 units of time and then quit.

This scenario corresponding to an overload has been simulated with four different allocation policies. We give below the total execution time and for the 600 first time units, the concurrency factor, the mean working time and the mean waiting time for large and small clients.

allocation policy	total execution time	at t=600				
		CF	Large clients		small clients	
			working time mean on 8 clients	waiting time mean on 8 clients	working time mean on 60 clients	waiting time mean on 60 clients
dynamic prevention	1233	29	5	594	295	125
fair dynamic prevention	1076	15	504	95	39	533
fair beforehand precaution	1124	10	6 at 561 2 at 38	6 at 39 2 at 561	107	419
fair combined prevention	1343	16	255	344	133	375

Fig. 2. Results of the simulation

With the dynamic prevention policy, large clients are not served (only 5 units of working time per client) while small clients monopolize the allocator (295 units of working time per client for a maximum working time of 300).

If we transform the dynamic prevention policy in a fair one (clients starting a new transaction are placed in a waiting-room if there is some client postponed by the allocator), large clients can easily obtain the requested resources (504 units of working time for a maximum working time of 561) while small clients

are almost always postponed (only 39 units of working time).

With the fair beforehand precaution policy (clients starting a new transaction are placed in a waiting-room if condition (R4) does not hold), 6 large clients monopolize the resources for 561 units.

The fair combined prevention policy (clients starting a new transaction are placed in a waiting-room if they are neither accepted by the fair dynamic policy managing 560 resources nor by the fair beforehand policy managing 140 resources), makes progress equally the small and the large clients.

5 EXPERIMENTATIONS

5.1 Programming in Ada

Simulation programs have been implemented in Ada with tasks and protected objects. Programs are available on ftp://bacchus.cnam.fr/pub/articles/

Several allocation policies have been programmed such as unfair dynamic prevention, unfair beforehand precaution, fair dynamic prevention, fair beforehand precaution.

We suppose that a processor is allocated to every process, thus all active transactive processes (not sleeping) which are not postponed execute concurrently.

5.2 Simulation scenario

The simulation scenario is described by a script, in which
Claim i, Enter i, Get i, Release i concern i resources,
Sleep j, Work j concern j time units and
Quit is associated with the releasing of all granted resources.
The 2 classes of processes in example 5 are expressed as following:

- Ex5_normal : Claim = 4, Behaviour =
  (Sleep 1, Enter 3, Work 1, Get 1, Work 100, Release 1, Work 200, Quit)
- Ex5_bursting : Claim = 4, Behaviour =
  (Sleep 2, Enter 1, Work 80, Get 2, Work 1, Quit)

The 6 classes of processes in example 6 are expressed as following:

- Ex6_large : Claim = 101, Behaviour =
  Sleep 1, Enter 50, 10*(Work 4, Get 40, Work 50, Release 40), Quit
- Ex6_small_i : Claim = 11, Behaviour =
  100 * (Sleep i, Enter 10, Work 2, Quit); where $i = 1..5$.

5.3 Simulation results

Each simulation run provides :

- the mean values of the resource stock, of the resource retention ratio, of the concurrency factor,
- the total execution time and the ratio of time spent by the processes in each of their states,
- for each process the total time spent in the different process states : Sleeping, Entering, Claiming, Working, Releasing, Quitting,
- for each significant simulation date, the number of processes in each state.

Bibliography

[Barkaoui 98] Barkaoui K., Pradat-Peyre J.F., Verification in Concurrent Programming with Petri Nets Structural Techniques, 3th IEEE conf. on High Assurance Systems Engeenering (HASE'98), pp.124-133, November 1998.

[Coffman 71] Coffmann E., Elphick M.,Shosani A., Systems Deadlocks, Computing Surveys 3.2, pp.67-78, June 1971.

[Dijkstra 67] Dijkstra E.W., Cooperating Sequential Processes, in Programming Languages, F.Genuys, ed., Academic Press 1967.

[Habermann 69] Habermann A.N., Prevention of System Deadlocks, Communications of the ACM 12.7, pp. 373-377, August 1969.

[Hagimont 94] Hagimont D., Chevalier P.Y., Freyssinet A., Krakowiak S., Lacourte S., Mossiere J., Rousset de Pina X., Persistant Object Support in the Guide System : Evaluation and Related Work. 9th Conf. on Object-Orientd Programming Systems, Languages and Applications (OOPSLA) Portland oct 1994

[Holt 71] Holt R., Comments on Prevention of Systems Deadlocks, Communications of the ACM 14.1, pp 36-38, January 1971;

[Holt 72] Holt R., Some Deadlock Properties of Computer Systems, Computing surveys 4.3, pp 179-196, September 1972

[Kaiser 97] Kaiser C., Pradat-Peyre J.F., Reliable Resource Allocation with Ada95 Preference Control, Cedric R.R. 97-06, CNAM 1997

Annexe

Banker's algorithm with transactive processes ordering

Let the set of transactive processes be ordered totally by increasing values of $dist(i)$; if two processes have same $dist(i)$, no matter their relative order.

Suppose the current allocation state is safe, i.e. (R3) holds.

a) request of P_k
Let a candidate process P_k with an additional request $add(k) > 0$. If P_k was not transactive, than insert it in the transactive set. The banker's algorithm is called when an allocation seems possible since $stock \geq add(k)$ and its role is to check whether the new state, after allocation, is still safe.

```
procedure bankers_decision(k :in processid; request :in positive; safe : out boolean) is
   --The old safe state Sold has to be saved
   stock_prime : positive := stock;
   g_prime : positive := grant(k);
   d_prime : positive := dist(k);
   r_prime : positive := rank(k);
begin
   -- The candidate state Scand is settled down
   stock := stock - request ;
   dist(k) := dist(k) - request ; grant(k) := grant(k) + request ;
   -- starting the banker'algorithm
   total := stock; safe := false;
   if dist(k) <= total then           -- sufficient condition always checked first
      safe := true;                   -- Scand is safe;
                                      -- in this case also dist(k) <= stock_prime
   else
      i := first (ordered processes); -- rank(i) =1
      while dist(i) < dist(k) loop    -- if dist(i) > dist(k) then Scand is not safe
         if dist(i) <= total then
            total := total + grant(i);
            i := succ(i);
            if dist(k) <= total then  -- sufficient condition always checked first
               safe := true; exit;    -- Scand is safe
            end if;
         else
            exit;                     -- Scand is not safe
         end if;
      end loop;
      -- if dist(i) = dist(k) and dist(k) > total then
      --    safe = false since Scand is not safe;
      -- end if;
      if safe then
         -- Pk will be removed from ordered set
         -- and be reinserted in a new rank rank(k) = r_prime
      else
         -- The old safe state Sold is restored
         stock := stock_prime;
         grant(k) := g_prime;
         dist(k) := d_prime;
      end if;
end bankers_decision;
```

b) release by P_k

After a release the new allocation state is safe

- P_k will be removed from ordered set and be reinserted in a new rank $rank(k) = r\_prime(k)$;
- if P_k has finished its transaction, it is no longer in the set of transactive processes.

For fairness, postponed candidates are checked for possible allocation. They

are served in FIFO order as long as possible and thereafter, if the leading process is still in the postponed queue, it is also served (as a property of the banker'algorithm, this is always possible).

This article was processed using the LaTeX macro package with LLNCS style