Deadlock-Avoidance in a Page-Parallel Batch Rasterizer

Yitzhak Birk
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Abstract

A rasterizer converts a document described in some page-description language into a sequence of full-page bitmaps (pagemaps), which can then be printed or displayed. The Page-Parallel rasterizer harnesses multiple processors to work on the same document. Any given page, however, is processed by a single processor, hence the name. For performance reasons, it is desirable to permit out-of-order rasterization. However, this can result in deadlock. This paper shows how to pace the rasterizer so as to avoid deadlock without being overly restrictive. In so doing, we extend previously-proposed deadlock-avoidance schemes to cases which seem to be outside their scope, and our approach may also be useful in other applications.

Introduction

The Page-Parallel Batch Rasterizer

A batch rasterizer receives a description of the contents of a document in some page-description language and converts it into full-page bitmaps, which are then used to directly determine the color of each pixel on the paper. We refer to the input form as *datastream* and to the final result (per page) as a *pagemap*. (An incremental rasterizer may receive an existing bitmap along with a list of required modifications.) Batch rasterizers are used primarily in conjunction with laser printers.

The last several years have brought about a dramatic increase in the sophistication of page-description languages and the applications that generate them. This has resulted in a growing gap between the rate at which pagemaps can be printed, which has been in excess of 200 pages per minute for over 10 years (e.g. IBM 3800) and the rate at which pages can be rasterized. To date, there are no microprocessor-based rasterizers that can match those printing rates for complex pages. Although microprocessors are becoming faster with time, the amount of computation per page will also increase with the introduction of color, multiple intensity levels and higher resolution.

The Page-Parallel rasterizer harnesses multiple microprocessors to achieve high rasterization throughput in a cost effective manner. This rasterizer consists of a set of processor-memory elements as well as shared memory. It employs a "Page-Parallel" approach, wherein each page is rasterized by a single processor, which is referred to as a "Page Imaging Processor" (PIP). The input datastream is scanned sequentially to detect page boundaries and some other information. The task performing this operation is referred to as the *Scanner*. The conversion of each page from the datastream to pagemap is then carried out in two stages: (i) conversion into an intermediate command stream (ICS) and (ii) conversion of the latter into bitmap. The tasks performing the two steps are referred to as *Imager* and *Builder*, respectively. (Any given page is processed by a single instance of Imager and a single instance of Builder.) The Page-Parallel architecture is depicted in Fig. 1.

The rasterization of a page usually requires the use of *resources*, the most prominent of which are fonts. Whenever preparing the resources requires significant processing and they are reused numerous times, the processed versions are cached and are shared among the PIPs. (E.g., raster versions of scalable text characters.) We refer to the processed version as *object* resources, as opposed to *source*. The collaboration among processors in preparing resources and the use of shared memory to store the results introduces coupling among tasks processing different pages.

"Source" resources are always available to the rasterizer when it needs them. This is achieved either by a print server that controls the deletion, by the rasterizer itself doing so, or by permitting the rasterizer to request such resources from the server when it needs them. In either case, "object" resources may (in principle) be deleted at the discretion of the rasterizer since they can be regenerated from the source versions.

TH0326-9/90/0000/0089/$01.00 © 1990 IEEE

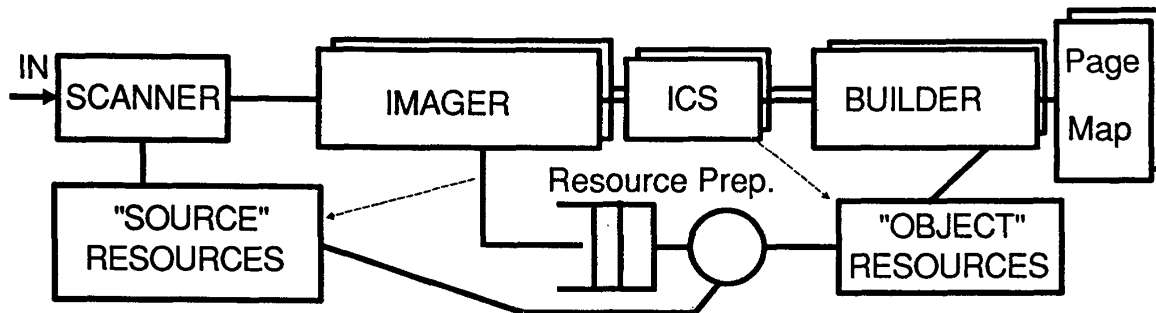

Figure 1: The Page-Parallel architecture. Any given page is processed by a single Imager and a single Builder. Source resources are set aside by Scanner. Imagers request the preparation of object resources and a memory allocation for them. The ICS contains direct pointers to locations of object resources, for use by Builders.

Several decisions made in the design of the Page-Parallel rasterizer affect the handling of cached "object" resources:

- The need for resource preparation is discovered by Imager, at which time the resource is either prepared while Imager waits or else a request is enqueued in a global resource-preparation queue. In either case, Imager requests the memory for the object resource (and waits for it to be allocated) and registers it as its eventual location (in the form of the address of the cached character or an address of a pointer to it.)
- In order to reduce the amount of memory required for ICS, as well as the amount of data copying that takes place, the ICS contains pointers to the resources rather than the resources themselves. The actual resources are stored in shared memory for use by all processors. Moreover, in the interest of performance and simplicity, the ICS usually contains direct pointers to the location of the cached resources.
- A Builder may not request the preparation of a resource. (The ICS points to characters by address, not by name, so A Builder doesn't even know what character to request.)
- An Imager and a Builder should never have to repeat work. In other words, their output may never be discarded before it is used. (Resource-preparation is not included in this restriction.)

One important implication of the foregoing decisions is that although the "object" resources can always be regenerated from the "source" versions, they may not be deleted so long as they may be referenced by existing ICS. Another one is that deadlock cannot in general be handled through recovery.

Irrespective of the handling of "source" resources, rasterizer memory management can never involve them in a deadlock situation. In the remainder of this discussion, we are therefore concerned only with "object" resources, which will simply be referred to as "resources" unless stated otherwise.

Potential Deadlock

For deadlock to occur, the following must all be true [1]:

- There must be shared resources which are held on an exclusive basis.
- Some process must be holding on to some shared resource (exclusively) while waiting for some other shared resource.
- The graph representing the hold/wait dependencies among processes must contain a cycle.
- Preemption must not be permitted.

Violating any of the above suffices to guarantee a deadlock-free system.

The foregoing design decisions give rise to the possibility of deadlock in the event of shortage of resource memory, as illustrated by the following example. Consider a 2-PIP system with a single pagemap per PIP

and 3 pagemaps in shared memory, and a 10 page document in which p.1 is 50 times more difficult than any other page. As long as PIP 1 is busy with page 1, no pages can be printed, so PIP 2 processes the remaining pages. Pages 2,3, and 4 are rasterized and placed in the shared memory pagemaps, page 5 remains in PIP 2's local pagemap, and pages 6 through 10 are stored in shared memory in ICS form. Suppose now that p.1 needs more resource memory in shared memory. Until it gets it, p.1 cannot be printed, so no pagemaps can be freed and the ICS of pages 6 through 10 cannot be consumed. Consequently, no resource memory can be freed, and the system is deadlocked.

A similar deadlock situation can occur involving ICS and pagemap memory, even without shared resources. However, the solution to the resource-pagemap-ICS deadlock subsumes the solution to the ICS-pagemap deadlock, so the discussion will be limited to the resource case.

Organization of the Paper

In the remainder of this paper, we examine ways of having a deadlock-free rasterizer without overly restricting its operation. We begin by looking deeper into the deadlock situation and examining the options, then come up with a set of rules which, if followed, would achieve our goal. Next, we show how those rules can be implemented efficiently. Lastly, we offer some conclusions.

Avoiding Deadlock in the Page-Parallel Rasterizer

There are three ways of treating deadlock: recovery, prevention and avoidance. However, recovery would require that we "roll back" Imager or Builder. Since it was decided that this is not permissible, recovery is not an option. Prevention and avoidance are similar, in that both are preemptive actions. With prevention, one effectively considers the "static" worst case scenario for all contending processes in deciding whether to allocate resources to a given process. In other words, it is always assumed that a process may concurrently need all the resources that it ever uses. Avoidance, on the other hand, looks deeper and takes into account things such as the order in which resources are requested by a process and whether some are freed before others are required. This can results in more permissive policies and consequently in higher utilization of the resources and in higher performance. The performance of a Page-Parallel rasterizer is inherently sensitive to variability in page complexity. It is therefore important to be as permissive as possible. Therefore, we opt for deadlock-avoidance.

Deadlock Avoidance

Deadlock-avoidance usually works as follows [1]. Given a priori information, for each process, about the amount of resources of each type that may be requested, a deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes. A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid deadlock. More formally, a system is in a safe state if and only if there is a sequence of processes $(p_1, ..., p_n)$ such that if the allocation being considered is made the processes can subsequently be allocated all the required resources in the order specified by the sequence, and such allocation results in all of them completing their jobs. Note that it suffices that the resources required by some process, say p_i, become available only once prior processes in the sequence have completed their work and released their resources.

The first deadlock-avoidance algorithm to follow the foregoing approach was the "Banker's algorithm" due to Dijkstra [2]; it dealt with a single resource type. (The name is due to the fact that bankers must always make sure that they can meet the demands of all customers in some order.) This was later extended to multiple resource types by Habermann [4]. Holt was the first to formalize the notion of deadlocks in terms of a graph theoretical model that could be used to represent our approach [5],[6]. A more recent overview of the deadlock appears in [7]. A comparison among concurrency-control methods using analytical models is presented in [8]. For a characterization of situations that permit the allocation of an available unit of resource to any requesting processor without running the risk of deadlock, see [9].

Application to the Page-Parallel Rasterizer

In attempting to apply the foregoing approach to the Page-Parallel rasterizer, we immediately run into problems:

- The total amount of resource memory required for a page is not known. We can only assume that the amount required to hold a single resource, say a single character, does not exceed the total amount of resource memory in the system. This is not very useful, since we would be able to permit at most one process to hold onto

any resource memory at any given time, thereby ruling out any parallelism.

- Resource memory is used in a way that combines features of shared and exclusive use: in deciding whether to allocate resource memory to a requesting Imager, the identity of the requesting process (or the page it is processing) may be taken into consideration. However, once the memory is allocated, any other Imager may generate pointers to the object resources that reside in it, and this memory cannot be freed so long as there are such pointers to it. It is even possible that an Imager that requested memory for a given resource and was refused generates a pointer to this memory once it has been allocated to a higher-priority Imager.

In view of the problems, it appears that our case is outside the scope of the conventional deadlock-avoidance approaches. However, there is a way around the problem. This hinges on the fact that Imagers and Builders operate in pairs. Once a pagemap is allocated to the Builder of a given page, all the ICS generated by that page's Imager can be consumed (even if the Imager did not complete its work on the page). This, in turn, removes all the pointers of that page to resources. Although we have no control over which page points to which resources and it is perhaps even too expensive to keep track of this, it is still true that if we allocate pagemaps for all pages whose Imaging has started, all pointers to resources can be consumed and resource memory can be freed as required.[1] Therefore, instead of directly controlling the allocation of resource memory to processes, we control the permission for an Imager to start working on a page (and generating pointers to resources on its behalf) and the allocation of a pagemap to the Builder of a page.[2] It is important to observe that we rely on the fact that all ICS for a given page can be consumed once a pagemap has been allocated to that page. Consequently, this approach does not extend to intra-page parallel rasterization with out-of-order generation of ICS within a page.

[1] The actual recycling of freeable memory is handled by the memory management system.

[2] If Builder were permitted to request the preparation of a resource, other approaches could be used, since resources would be deletable even when referenced by existing ICS. However, for this to be strictly true, the reference to each character would have to be by its full long name (font.size.rotation.character), which is very long. Simply adding a level of indirection (i.e., the ICS points to a location in a table which in turn points to the character bitmap) would reduce the amount of memory that cannot be cleared so long as there are references to a character, but would not guarantee deadlock-freedom.

Resource Allocation Rules

Having shown that the peculiarities of the resource memory can be circumvented, we are back within the scope of existing approaches to deadlock avoidance. However, we must still spell out the exact rules for deadlock avoidance and the tests for safety of a state. Because of the printing order constraints, we are guaranteed that if some Imager is blocked due to a request for resource memory that cannot be granted then either the problem resolves itself or else all Imagers will eventually be blocked. For the purpose of deadlock avoidance, it is therefore sufficient to consider the case of all Imagers blocked, and to guarantee that when this happens all ICS can be consumed without need to generate any new ICS (and pointers to resources) in the process. Before proceeding, we introduce some more details of the system which must be taken into account in making allocation decisions. (We are not introducing additional resources.)

Due to properties of the printing mechanism as well as performance considerations, it is sometimes required to guarantee pagemaps for a number, say N_p, of "priority" pages. These have the lowest page numbers (more precisely, they are the first ones in the printing order.) Pagemaps can only be allocated to later pages once those have been allocated or reserved.[3] We refer to this as the *pagemap priority policy.* Shared memory contains at least $N_p \cdot maxSize$ pagemap memory, where $maxSize$ is the maximum size of a pagemap. (This may be relaxed in special cases, such as the use of large paper only for job separation.) The notation used in stating the rules is as follows:

- M_t (N_t) Total amount of pagemap memory in bytes (pagemaps).
- i Lowest page number that has yet to print.
- N_p (M_p) Number (aggregate pagemap memory size) of contiguous pages, beginning with i, for which pagemaps must be reserved. (Priority pages.)
- N_c The number of contiguous pages, beginning with i, which have been completed by Imager (no additional resource memory will be needed for them). Furthermore, if printing is in singles and in page-number order, these can all be printed and their pagemaps can then be freed.

[3] In the case of a duplex printer in which both impressions are made in one pass, N_p is at least two; it may be larger in mechanisms that incur a significant start-up time whenever printing stops or when desired in order to facilitate recovery from paper jams.

N_{ac} Number of "active" pages. A page becomes active when its Imaging is begun, and ceases to be active once it becomes one of the N_c pages.

P^i_{max} Highest page No. whose Imaging has begun.

P^p_{max} Highest page No. which has been allocated a pagemap.

j Denotes an arbitrary page.

Additional notation will be introduced as required.

We now state necessary and sufficient conditions for allocating a pagemap for a page, e.g. page number j, and for permitting its Imaging to begin. Although the latter always precedes the former, the pagemap allocation is discussed first since the ability to allocate a pagemap influences the decision whether to permit Imaging of j to begin. For each set of conditions, we will show that they are necessary for deadlock avoidance and/or for satisfying the pagemap priority policy. We will also prove that the two sets are jointly sufficient.

Pagemap Allocation. A pagemap may be allocated to a given page, say j, if and only if *all* the following conditions are satisfied:

- there is sufficient free pagemap memory (obvious),
- the allocation leaves enough pagemap memory for those of the N_p priority pages which have not yet received a pagemap, and
- the allocation to j can be shown not to lead to a future violation of the pagemap priority policy.

The last condition differs from the second one whenever page size is variable as well as when certain printing orders are used.

The following examples illustrate the necessity of the different rules.

Example 1. Direct violation of the pagemap priority policy; $N_p = 2$; $Nt = 5$.

Page Number	1	2	3	4	5	6
Imaging completed	+	+	–	–	–	–
Pagemap allocated	+	–	+	+	+	?
Page size	1	1	1	1	1	1

Allocating a pagemap to p.6 would take up the last free pagemap, thereby preventing p.2 from receiving one until p.1 is printed. This would violate the policy since $N_p = 2$, and the violation could lead to deadlock (e.g. if pages 1 and 2 must be printed together on different sides of the same sheet).

Example 2. Eventual violation of the pagemap priority policy; $N_p = 2$; $Mt = 8$.

Page Number	1	2	3	4	5	6
Imaging completed	+	+	+	+	–	–
Pagemap allocated	+	–	–	–	+	?
Page size	1	1	5	1	1	5

Allocating a pagemap to p.6 would not directly violate the policy, since the total amount of allocated pagemap memory would become 7, leaving 1 for p.2. However, once p.1 was printed and p.3 became one of the preferred pages, the amount of free pagemap memory would be 1 (p.2 would still have a pagemap at this time), whereas p.3 would need a pagemap of size 5. The problem here is due to the fact that the amount required by p.3 is larger than that released by p.1. The eventual violation, when it occurs, can also result in deadlock. The foregoing rules are thus clearly necessary to avoid deadlock while adhering to the priority policy.

Permitting Imager to Begin. Imaging of a page, say j, may begin if and only if *all* the following conditions are satisfied:

- if all Imagers were blocked and Building and printing continued as much as possible (pagemaps of printed pages would be freed and allocated for Building of subsequent pages and so on until no more pages could be printed), there would be sufficient pagemap memory for all remaining pages whose Imaging had started.
- all said pagemap allocations would conform to the pagemap-allocation rules.

It should be noted that if Imaging is started in printing order, satisfying the first condition guarantees that the 2nd is also satisfied.

The following examples illustrate the necessity of the conditions and expose some subtleties.

Example 3. j cannot receive a pagemap; $N_p = 2$; $Mt = 3$.

Page Number	1	2	3	4	5
Imaging begun	+	+	+	?	–
Imaging completed	–	+	–	–	–
Pagemap allocated	+	+	+	–	–
Page size	1	1	1	1	1

If Imaging of p.4 were permitted to begin and Imagers were subsequently blocked, there would be no printable pages. Therefore, pages 1,2, and 3 would never

free their pagemaps and p.4 would not be allocated one. As a result, p.4 would still have references to resources, which would thus have to remain in memory, possibly preventing Imaging of p.1 from progressing.

Example 4. permitting j to start may violate the pagemap priority policy; $N_p = 2$; $Mt = 3$.

Page Number	1	2	3	4	5
Imaging begun	+	–	–	+	?
Imaging completed	–	–	–	+	–
Pagemap allocated	+	–	–	+	–
Page size	1	1	1	1	1

If Imaging of p.5 were permitted to start, and Imagers were subsequently blocked, there would be enough pagemap memory for all active pages, including p.5. However, doing so would violate the priority policy, since there would be no pagemap for p.2 when it needed it. This example illustrates the importance of including the pagemap priority policy in the condition.

Example 5. Permitting j to start would cause an indirect problem; $N_p = 2$; $Mt = 3$.

Page Number	1	2	3	4	5
Imaging begun	+	+	?	–	+
Imaging completed	–	–	–	–	–
Pagemap allocated	+	–	–	–	–
Page size	1	1	1	1	1

Initially, it appears that Imaging of p.3 should be allowed to begin, since a pagemap may be allocated to p.3 when desired (pages 1 and 2 are the preferred ones, but there are 3 unallocated pagemaps. Also, p.3 has priority over p.5). However, if all Imagers were blocked, p.5 would never receive a pagemap since p.1 would not be printed. Thus, p.5 would prevent resources from being freed and deadlock could occur. This example shows why the rule must be that "it must be possible to give a pagemap to every active page" rather than merely to j. It can also clearly be observed that this problem would never occur if Imagers were begun in page-number order.

The Rules are Necessary and Sufficient. Given that deadlock must be avoided without violating the pagemap priority policy, the 5 foregoing examples prove that the conditions are necessary. It thus remains to prove that they are sufficient.

Sufficiency follows directly from the statement of the conditions and the mechanism outlined earlier for deadlock avoidance. If we adhere to the rules for permission to begin Imaging, we are guaranteed that all ICS is consumable; this, in turn, breaks the apparent hold/wait cycle, thereby violating one of the necessary conditions for deadlock. The rules for allocating a pagemap, which form part of the test for beginning Imaging, further guarantee that the pagemap priority policy is not violated.

Implementing the Rules

In the last section, we characterized a safe state. In this section we present efficient algorithms for testing the safety of the state following the allocation of a pagemap or the permission to an Imager to commence working on a page. Throughout the discussion, we use the term "page number", implicitly assuming that pages are printed in page-number order. Nevertheless, other printing orders can be accommodated by simply interpreting "page number" as the printing order, provided that the pagemap priority policy is also stated in terms of the printing order. The algorithms presented shortly constitute a precise implementation of the rules. The examples that served to prove the necessity of the different rules and exposed their subtleties also help understand the algorithms.

Pagemap Allocation. The willingness to consider out-of-sequence pagemap allocations is important. For example, a good operating policy may be to request a pagemap for a page only when more than 250KB of ICS have been generated for it or its time to print is approaching. As another example, consider a case wherein the difficulty of pages is known in advance. (This would happen if the document had been rasterized in the past and is archived in datastream form along with the relative rasterization time of each page.) It would be useful to begin work on a difficult page as early as possible, but this may require a pagemap.

Pages (and therefore pagemaps) may vary in size, and this introduces a possibility of deadlock whenever several small pages are followed by large pages. If, for example, after having allocated a pagemap to page 1, which happened to be smaller than page 2, the remaining pagemap memory were allocated to pages 3 and above, deadlock would result even if $N_p = 1$ since the freeing of page 1's pagemap would not suffice for page 2.

A conservative solution would be to reserve $M_p = N_p \cdot maxSize$ pagemap memory for the priority pages. However, assuming that the Scanner discovers the sizes of pages it scans, that pages are scanned in page-number order, and that a page is never allocated a pagemap before the Scanner discovers its size, we can do better. The general idea is to first determine how much of the free pagemap memory must be set aside in order to avoid violations of the priority policy, be

they immediate or eventual, and to subtract that from the amount of free pagemap memory. If the result is greater than the amount requested by j, a pagemap is allocated. Otherwise it is not.

The test is carried out by considering a sliding window containing N_p consecutive page numbers. In its initial position, the window contains pages i through $i+N_p-1$, and sliding stops when it contains pages $j-N_p$ through $j-1$. For each position of the window, we sum up the sizes of the pages contained in it and refer to the result as the total memory size of that window. This is denoted by $M_{wt}(k)$, where k is the lowest page number in the window. We then compute the maximum of $M_{wt}(k)$ over k in the range i through $(j-N_p)$ and use that in the test for allocating a pagemap to j.

Although the above approach would prevent deadlock, it is overly restrictive in two ways:

- Although $M_{wt}(k)$ indeed represents the required amount of pagemap memory for the k-window, some of the pagemaps for the window may have already been allocated. No free memory need be reserved for those pages.
- Before $(k, k+1, ..., k+N_p-1)$ becomes the priority-page window, all pagemaps for pages numbers smaller than k will have been freed. The pagemap memory already allocated to those can be relied upon for pages in the k-window.

We therefore use a more sophisticated test.

The test. At any given time, the real amount of memory that still needs to be reserved for the k-window, denoted $M_{wr}(k)$, is given by

$$\begin{aligned} M_{wr}(k) &= M_{wt}(k) \\ &\quad -(\text{pagemap mem. already allocated} \\ &\qquad \text{to pages } k, ..., k+N_p-1) \\ &\quad -(\text{pagemap mem. already allocated} \\ &\qquad \text{to pages } i, i+1, ..., k-1) \\ &= M_{wt}(k) \\ &\quad -(\text{pagemap mem. already allocated} \\ &\qquad \text{to pages } i, i+1, ..., k+N_p-1) \end{aligned}$$

We now slide the window and obtain the amount of unallocated pagemap memory that must be reserved for earlier pages than j as

$$M_r(j) = \max\{M_{wr}(k)\}, \quad k = i, i+1, ..., j-N_p.$$

For any given k, $M_{wr}(k)$ can be negative (more memory will be freed than is needed). However, we clearly cannot allocate memory that is not free (the first condition). Therefore, j can be allocated a pagemap if and only if

$$M_{\text{free}} - \max\{M_r(j),\ 0\} > \text{size}(j)$$

Note that when printing is in page-number order, $M_{wr}(i)$ is never negative, since there are no earlier pages than i in the system, so the max is redundant. We keep it for clarity and for the case of strange printing orders.

Clearly, $M_r(j)$ changes with i as well as with any pagemap allocation or freeing of a pagemap, so it should be reevaluated as necessary. However, there are ways to hold the computation to a minimum. For example, we could store $M_{wr}(k)$ for all k. For every event involving a page, say j, (freeing a pagemap, allocating a pagemap, etc.), $M_{wr}(k)$ needs to be updated only for $j-N_p < k < j+N_p$. Savings in computation of $M_r(j)$ could be attained in similar ways by trading storage for computation. (This is similar to the idea of dynamic programming.)

We next sketch the proofs of correctness and tightness of the foregoing pagemap-allocation algorithm. (We need only show that it precisely implements the rules, as those were proven to be necessary and sufficient for deadlock-avoidance subject to the design decisions and the pagemap priority policy.) We use P^p_{max} to denote the maximum page number that has been allocated a pagemap.

- **Correctness.** (no deadlock)
  - The proof is by induction on events. The events are:
    1. printing of page i and freeing of its pagemap
    2. allocating a pagemap to page j, $i \leq j < i+N_p$
    3. allocating a pagemap to page j, $j \geq i+N_p$
  - **Initial conditions.** Assuming that there is at least $N_p \cdot maxSize$ pagemap memory, and that it is initially free, there is clearly no problem at the outset.
  - **Induction step.**
    1. The new "critical" window consists of pages $(i+1, i+2, ..., i+N_p)$. If $P^p_{\text{max}} > i+N_p$, the requirements of this window

were already accommodated whenever the allocation of a pagemap to $j > i+N_p$ was being considered. The eventual freeing of i's pagemap and its availability for later pages was possibly relied upon when allocating pagemaps to later pages. However, having actually freed it is clearly a change for the better. If $P^p_{\max} \leq i + N_p$, no pagemaps have been allocated to pages beyond the "critical" ones, so the assumption of sufficient pagemap memory for those suffices.

2. This reduces the amount of free memory, but also reduces the requirements for any window that includes j. Later windows are unaffected, since j's pagemap will be freed before they need it. Earlier j-less windows do not exist since $j < i + N_p$.
3. We verify sufficient memory for early windows before allocating, so no new problem can be created.

- **Tightness.** A formal proof is omitted. The idea is to relax each of the constraints and show one example of deadlock in each case. Examples 1 and 2 can serve for this purpose.

It readily follows that any available pagemap memory may always be allocated to a page once all lower-numbered pages have been allocated a pagemap.

Pagemap allocation decisions are sometimes complicated by two dynamic changes:

- Pagemaps are compressed once completed.
- The total amount of memory that may be allocated for pagemaps is allowed to change dynamically.

We extend our algorithm as follows:

1) Compression of Pagemaps in Shared Memory

Unlike the page size, the compression ratio is only determined once a pagemap is completed and compressed, and thus is not known to the Scanner. Nevertheless, the algorithm used for variable size pagemaps is sufficiently powerful for compression, and only the following minor modifications are required:

- initially, use the uncompressed size of the pagemap; once it is compressed, use its compressed size in the calculations.
- if there are pending requests that cannot be satisfied and a pagemap is compressed, reevaluate them.

The correctness of this simple way of accommodating compression hinges on the assumption that a compressed pagemap is no larger than a noncompressed one. Thus, if things were OK before compression, they remain OK after it is carried out. It is therefore important to either pick compression algorithms that never expand, or to use the noncompressed version if compression fails. Otherwise, one would have to use the product of the noncompressed size and an upper bound on the expansion factor as the initial estimate of a pagemap's size.

2) Variable Amount of Pagemap Memory

Increase in pagemap memory.
Change the total pagemap memory size and the amount of free memory. Next, see if any pending requests can be granted, and proceed as usual.

Reduction in pagemap memory size.
This must be done carefully, since some of the free memory may have been implicitly committed by letting the Imaging of some page begin. A careless reduction of the free memory could thus result in deadlock. The algorithm is as follows:

- Create a dummy page
  - *pageSize* =desired reduction in pagemap memory
  - $pageNumber = P^i_{\max} + 0.5$
- Whenever a page is printed, attempt to allocate a pagemap to the dummy page using the usual algorithm, with the following differences:
  - Give the dummy page priority in the sense that its request is evaluated before those of other pages (the granting is still subject to the regular policy for a page number $P^i_{\max} + 0.5$).
  - The pagemap may be allocated in pieces. (Whenever some memory is allocated to the dummy page, its size is reduced by the amount allocated to it; the dummy page thus never owns any memory.)
- Whenever the dummy page receives memory, reduce M_t by that amount.
- Once the dummy page receives the entire requested amount, delete it.

- In deciding whether to permit the Imaging of a page, say j, to begin, distinguish between two cases:
  - if $j < P^i_{max}$, there is no change.
  - if $j > P^i_{max}$, subtract the current size of the dummy page from M_{free} (possibly causing it to become negative). Other than this, ignore the dummy page in the calculations.

We next prove that this algorithm is correct and tight.

Correctness. The assignment of a page number higher than P^i_{max} to the dummy page, along with the use of this number in the pagemap-allocation decisions and with the fact that the dummy page does not affect the decision to start Imaging of lower-numbered pages, guarantees that allocating pagemap memory to this page cannot cause deadlock involving pages numbered P^i_{max} and below. In deciding whether to permit Imaging of later pages to begin, the adjustment of M_{free} to correspond to the situation after the completion of the memory reduction guarantees that Imaging of such pages can begin only if deadlock can be avoided without relying on memory that is to be removed. Finally, no later pages can receive pagemaps before the dummy page is removed, so pagemap allocation to later pages is not a source of concern.

Tightness. The priority given to the dummy page in consideration for receiving pagemap memory (the granting is per the rules, to avoid deadlock), along with the permission to allocate memory to it incrementally, guarantee that the requested memory reduction is achieved at the earliest possible time. (An earlier allocation would violate the pagemap-allocation rules, which have already been shown to be tight, and would thus result in potential deadlock.)

Permission to Start Imager. With a general printing order, it is impractical to implement the rules in closed form. Instead, one can state them as a program that *simulates* the blocking and printing process as well as the allocation and freeing of pagemaps. If the program succeeds in allocating a pagemap to all active pages, including j, then j's Imaging may begin. Else it may not. Clearly, the regular allocation rules must be obeyed. Following is a skeleton of such a simulation program.

```
Permission to begin Imaging (j)
   Stop all Imagers; /*this is the worst case*/
   REPEAT
      Print all Printable pages and free their
         pagemaps;/*in specified order*/
      Allocate free pagemaps to next in line;
      Build all pages whose Imaging has been
         completed and which have a pagemap;
   UNTIL (no new pages Built);
   IF (possible to allocate a pagemap to all
      active pages including j)
                   RETURN (OK); /*to start j*/
   ELSE RETURN (not OK);
```

Comments:

- The loop is equivalent to "print all eventually-printable pages and free their pagemaps."
- The allocation test is conducted as discussed earlier, including all the generalization, so those need not be addressed again.
- It is possible to insert the test for allocation to all active pages into the loop and exit if OK, potentially saving iterations.
- Whenever pages are printed in bunches of b (e.g., in some duplex printers $b = 2$), we slide the window in steps of b pages, visiting only the feasible positions.
- In specific implementations with a known printing order and page size properties, a simpler equivalent of this program will usually be easy to construct.
- As in the case of the pagemap-allocation test, additional memory can be used to store intermediate results and reduce the processing requirement.

Conclusions

The desire to share resource preparation and storage, along with the decision to permit the intermediate command stream to point at cached raster characters by location, i.e., not by their full name, give rise to the possibility of deadlock in the Page-Parallel rasterizer. We studied this deadlock problem and provided efficient algorithms for flexibly allocating resources while avoiding deadlock.

Although the peculiar way in which resource memory can be shared among processors precludes the direct application of established deadlock-avoidance algorithms, we were able to circumvent the problem by taking advantage of the fact that by allocating a pagemap for a given page we are able to delete all the references to resources that were made on its behalf; if this is done for all active pages then all resource memory can be freed. This kind of approach seems quite general, and may also apply to other situations.

References

[1] J.L. Peterson and A. Silberschatz, *Operating System Concepts*, 2nd Edition, Addison-Wesley Publishing Company, (1986).

[2] E.W. Dijkstra, "Cooperating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, (1965); reprinted in Gen68, pp.43-112.

[3] F. Genuys (Ed), *Programming Languages*, Academic Press, London (1968).

[4] A.N. Habermann, "Prevention of System Deadlocks", *CACM*, Vol. 12, No. 7 (July, 1969) pp. 373-377, 385.

CACM

[5] Richard C. Holt, "On Deadlock in Computer Systems", PhD Dissertation, Cornell University, (Jan 1971). (Also CSRG Tech Rep 6, CSD, U. Toronto)

[6] R.C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4 No.3 (Sep. 1972) pp.179-196.

[7] S.S. Isloor and T.A. Marsland, "The Deadlock Problem: An Overview", *IEEE Computer*, Vol. 13, No. 9, (Sept.1980), pp.58-78.

[8] K.C. Sevcik, "Comparison of concurrency control methods using analytical models", *Info. Processing* 83:847-858, (1983), R.E.A. Mason (ed)

[9] M.C. Chen and M. Rem, "Deadlock-Freedom in Resource Contentions", *Acta Informatica*, Vol. 21, pp. 585–598, Springer-Verlag, (1985).

Acknowledgments and Credits

Steve Scott observed that once a pagemap is allocated to a page its ICS can be consumed. Steve, Wil Plouffe, Arlen Strietzel, Doug Ehlers, Jeff Lotspiech, and Sig Nin participated in various discussions on this topic. I am particularly grateful to Sig Nin and to Jim Stamos for critically reviewing early versions of this manuscript.