

DEFINING DEADLOCK

Gertrude Neuman Levine
Fairleigh Dickinson University
Teaneck, NJ 07666
levine@fdu.edu

ABSTRACT

Deadlock has been widely studied in many fields of computer science, notably in communications, database, and operating systems. Perhaps because (at least one state called) deadlock is tractable, there exists extensive literature on the subject. Deadlock was apparently decisively defined over thirty years ago, with its characteristics and handlers. Yet, much of the literature remains inconsistent in its treatment of this anomaly.

A more precise definition is clearly needed to distinguish between the different states that are termed deadlock. A classification of dead states is required to distinguish the causes and appropriate handlers for each. We introduce a model to structure our research.

Index terms: deadlock, abortion, buffer overflow, collision, preemption, roll back, throttling, time-out, collision

1. INTRODUCTION

The following five classes of dead states have all been termed deadlock in the computer science literature:

- an infinite waiting state containing a nonempty set of processes
- an infinite waiting state containing a nonempty set of processes that make no progress according to the rules of the protocol
- an inactive infinite waiting state containing a nonempty set of processes that make no further progress
- an infinite waiting state containing a circular chain of processes
- an infinite waiting state containing a circular chain of processes, each of which is holding a non-preemptable resource while waiting for another held by the next process in the chain

Of course, the existence of different definitions for “deadlock” is insufficient motivation for a paper. Many texts that characterize deadlock with the properties of the last class, however, contain examples that do not conform to that category. Thus, handlers that are stated to be effective for deadlock control frequently do not apply. Tanenbaum, a leading researcher in Operating Systems, claims that deadlock is well understood and implies that its study is fairly simple[23]. Yet, consider the following example of “deadlock” provided in another Tanenbaum text [24]:

```
producer()
{
    int item;
    while (TRUE) {
        produce_item (&item);      /* If no slot is empty when down(mutex) is executed */
        down (mutex);              /* producer() holds the “soft” resource, mutex, and then */
        down (empty);              /* blocks on empty. producer() waits for consumer to */
                                   /* signal that empty is available. */

        enter_item (item);
        up (mutex);
        up (full);
    }
}

consumer ()
{
    int item;
    while (TRUE) {
        down (full);
        down (mutex);              /* consumer() blocks, since mutex is locked by producer and */
    }
}
```

```

/* cannot execute up(empty), a cooperation mechanism. */
/* consumer() has neither requested nor obtained empty. */
remove_item (&item);
up (mutex);
up (empty);
consume_item (item);
}
}

```

As we shall see, the dead state caused by the indicated execution of the above code does not satisfy all of the conditions specified for deadlock in both of the Tanenbaum texts cited. Deadlock definitions, however, are ambiguous enough to allow such contradictions. How can this anomaly be controlled if it is not understood? We ask our readers to review with us material that they may feel is all too familiar.

2. DEADLOCK AS DEFINED IN OPERATING SYSTEMS' TEXTBOOKS

There are four necessary and sufficient conditions for deadlock [3]:

- Resources must be used in mutual exclusion.
- Resources cannot be preempted.
- Processes hold resources while waiting to acquire others.
- There exists a circular chain of processes, each of which is holding a resource while waiting for a resource held by another process in the chain.¹

Holt [10] characterizes deadlock with a directed (wait-for) graph. In Figures 1 and 2, processes P1 and P2 are in deadlock.

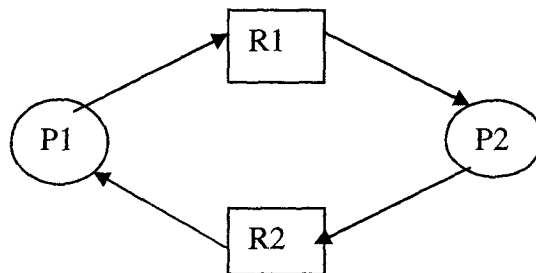


Figure 1. Deadlock with identifiable resource elements.

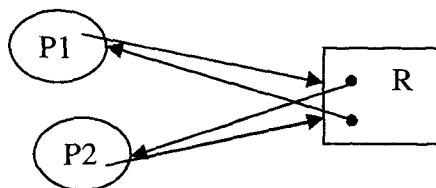


Figure 2. Deadlock with fungible resources.

Figure 2 illustrates a circular wait with two elements of a fungible² resource. (If multiple elements are being requested, the arrow emanating from the process should be assigned a weight.) Fungibility improves resource utilization by enabling scheduling from a single queue.

¹ The system is vulnerable to deadlock because of the first three conditions. With these necessary preconditions, the circular chain is sufficient to cause it [4].

² Fungible is a term used in law to characterize interchangeable units. Thus, if a bottle of milk is broken, it is sufficient either to replace it with a unit of equal size or to repay its cost in order to satisfy the obligation.

Deadlock can be prevented or detected, with several algorithms available. Deadlock prevention schemes include negation of one of more of the listed preconditions. For example, to prevent the occurrence of a circular chain, processes can be ordered in their access to resources, with mechanisms such as resource pre-allocation (which also negates the hold and wait precondition) and timestamps. Alternatively, resources may be linearly ordered in terms of processes' access, which, however, negates fungibility. Deadlock can also be prevented by an "avoidance" scheme, which ensures that the system remains in a safe state. Using banker's algorithm, for example, processes pre-state their maximum needs and the system continuously monitors allocation, reserving sufficient resources for processes to complete. Serial process execution and resource pre-allocation are also avoidance algorithms. Heuristics for avoidance are common, such as throttling new processes to ensure 20-30% of free buffers [10] on the assumption that these are sufficient for currently executing processes to complete.

Detection schemes find the circular chain. At least one victim is chosen; its partial service is rolled back and its resources freed, in an attempt to break the deadlock and allow the other processes to complete. The victim(s) can be restarted later, without alteration, but its rolled back service must be repeated.

Holt [10] provides an example of a single process in deadlock: process Revenge is infinitely suspended by a wait for an event that never occurs. Similarly, Nutt [16] states that a kernel process is in deadlock if it is permanently blocked waiting for a resource that will never become available. In accordance with the Halting Problem, however, this type of dead state is impossible, in general, to prevent or even detect with current technology. A deadlock cannot occur with only one process (as erroneously stated in [11], [27]). Deadlock is an anomaly of traffic control (i.e., competition synchronization) and contains at least two competing processes.

Holt [10] also considers deadlock as a circular wait with consumable resources, such as messages, that can be dynamically created and destroyed. Message deadlocks, however, are controllable neither by serial execution nor by most of the previously discussed prevention schemes.

Many Operating Systems textbooks provide the above characteristics and handlers in a Deadlock chapter, but also include incorrect examples. Silberschatz et al. [20] claim that a deadlock exists if two trains on different tracks approach a crossing and wait for the other to proceed. {Yet neither train is holding a resource requested by the other.} Similarly, Stallings [21] gives an example of four cars at different stop signs approaching an intersection, where each gives precedence to the car on its right. {Yet none of the cars (or trains) have been allocated intersection slots, which are the only resources that are mutually requested. Nor is roll back required to break the wait.} Davis and Rajkumar [5], as well as Flynn and McHoes [9], cite a "deadlock" example of two programs issuing a seek command to reposition the access mechanism. Each is interrupted before executing its read command, and discovers that the other has moved the disk arm. Each then reissues the seek command, but is again interrupted by the other. This sequence continually repeats. {This dead state is indeed an anomaly of traffic control and can be controlled by resource pre-allocation. Processes, however, are not blocked from resources, which are in fact preempted. In addition, resources are already requested in a linear order.} Pinkert and Weat [17] include a "deadlock" example similar to the producer/consumer code in our introduction. They state that a typo in the order of cooperation and competition mechanisms results in the consumer locking the mutual exclusion semaphore before it blocks on an empty buffer. {We note that linear orders, resource pre-allocation, and avoidance methods are all ineffective for this dead state.} The above examples are inconsistent with the material presented in their chapters, even though they appear in multiple editions of some of the texts.

Researchers in the field of distributed databases typically limit their applications to requests for data locks, and their treatments are consistent with deadlock material [e.g., 25, 29]. Additional restrictions are required to obtain a global knowledge of a deadlock, however, and fungible resources (basically OR-requests) are generally not included in detection algorithms.

Some additional remarks will be useful. Processes in deadlock may be suspended in an inactive wait or looping endlessly in a busy wait. They may be holding resource buffers, "soft" resources, which are typically fungible. In an attempt to differentiate between different types of dead states, some of the literature has begun to use the term resource deadlock [e.g., 21, 25] for the anomaly discussed above (although not always consistently). In the remainder of this paper, we accede to this terminology, although dead states with resources are obviously not always resource deadlocks.

In Table 1, we provide a taxonomy of dead states:

	Circular dead states	General dead states (Supersets of corresponding circular dead states)
Caused by cooperation mechanisms	Communication deadlocks	Communication dead states
Caused by competition mechanisms	Scheduling deadlocks	Scheduling dead states
Caused by interleaved execution of competing requests whose service is restricted by cooperation mechanisms.	Interleaved deadlocks (Supersets of resource deadlocks)	Interleaved dead states

Table 1. A Classification of Dead States

3. CIRCULAR DEAD STATES

Deadlock has been defined as any infinite circular waiting state in the literature of communication and database systems [12, 19]. This simplification also occurs in several operating systems textbooks, even when the rest of the section on deadlock clearly is about resource deadlock [7, 20, 21, 24]. For the remainder of this paper we accept this definition, since a circular dead state can be detected by process orders and characterized by a wait-for graph. We illustrate three distinct classes of circular infinite waits, with only resource deadlock, a subset of the third class, having all of the characteristics listed in [3].

3.1 Communication Deadlock

A dead state can occur with a circular chain of incorrectly written processes that wait for a message from another in the chain before each sends the awaited message. For example, two processes will never complete if they are programmed to wait for a (consumable) resource instance before they create the resource instance requested by the other [10]. The producer/consumer example previously cited [17, 24] also contains errors in the cooperation mechanism, preventing the sending of the awaited signal. These communication deadlocks are not caused by interleaved code execution, nor can they be prevented by resource pre-allocation, serial execution, or maintenance of safe states. Typically, after some period of time, a circular wait is detected or assumed and the processes are aborted. Processes in communication deadlock must be reconceived by their users before they can complete service.

3.2 Scheduling Deadlock

A circular chain of four cars infinitely waiting behind stop signs at each of the four corners of an intersection has been called a resource deadlock [21]. A dead state with two trains approaching on different tracks where each gives precedence to the other at a crossroads [20] has also been incorrectly labeled. These cars and trains are not waiting for resources held by others; the scheduler has failed to assign a single process the priority required for accessing the resource. Recovery does not entail roll back and the resultant repetition of service; it is sufficient to dynamically modify the scheduling algorithm at the present point of service, perhaps by waiving one vehicle through. These scheduling deadlocks, caused by incomplete competition mechanisms, do not require any alteration of processes.

3.3 Interleaved Deadlock

Interleaved deadlock is caused by a combination of competition and cooperation mechanisms, such that only the competition mechanisms are incorrect. Requests from at least two processes compete for the same resource elements and the scheduler interleaves their execution, allocating a resource to each. Each process is restricted from releasing the held resource while waiting for an event to occur at another process.

In resource deadlock, the scheduler continually assigns highest priority to the earlier requests (resources cannot be preempted), blocking later (active or suspended) competing requests. Each high priority request waits for the service of its process's blocked request(s). A circular chain exists of processes that are waiting for resources held by other processes in the chain. Resource deadlock can be controlled by mechanisms of either the processes or of the scheduler. For example, processes can agree to access resources in a linear order or the scheduler can maintain a safe state. When resource deadlock is rare, control systems may avoid the overhead of

prevention algorithms. Detection schemes find the circular wait. After roll back, one or more processes are restarted for repeated service, without correction, in the hope that the interleaved execution that caused the dead state does not recur.

4. ACTIVE AND INACTIVE DEAD STATES

In some of the computer science literature, a deadlock is defined as a state containing processes that can make no more forward progress based on the transitions defined in the protocol [22]. Alternatively, some researchers label deadlock as such inactive states and livelock as active ones [15, 28]. There exist three classes of these dead states, corresponding to each of the circular waiting states previously discussed.

4.1 Communication Dead States

Communication dead states are caused by cooperation mechanisms. Such dead states may be active, such as a kernel process that is infinitely looping [16], or inactive, such as infinite suspensions with Wait macros or Enq facilities [10]. An inactive dead state can also occur if a process outputs an incorrect value at a resource; all processes whose service is dependent upon that value will be aborted and never complete service. Or, if a resource is flawed, perhaps containing a defective coprocessor, then a process may be reconceived continuously and still never make progress, assuming no other computer is available. Similarly, if a resource is represented by a flawed server, a client may not receive service. In addition, processes that access forbidden resources should be doomed to failure. We claim that all of these processes enter into a dead state as defined in this section unless correction occurs.

Communication dead states may be impossible to prevent. They are generally the most expensive dead states to correct, requiring that processes be reconceived by their users.

4.2 Scheduling Dead States

Scheduling dead states are caused by incorrect conflict resolution, independent of user assigned cooperation mechanisms. For example, a process may be suspended on a queue from which it is never retrieved. In addition, if a scheduling policy requires pre-allocation of resources, two malevolent or poorly controlled dining philosophers can alternate acquiring chopsticks and keeping the philosopher between them in an inactive dead state [6]. Alternatively, two processes may repeatedly access a resource, collide, and back off, perhaps following a roll back policy to recover from resource deadlock, causing an accordion effect [26]. Circular routing is also an active scheduling dead state.

This class of anomalies is not caused by interleaved execution of process's requests. For correction, waits can be detected and the scheduling policy adjusted without loss of service. Typically, older processes are buffered at their present point of service and their priority increased by their waiting time (a policy called aging). A linear order of processes resolves conflict for processes with the same access time. Sometimes, insertion of a random delay, as in Ethernet, is a probabilistic handler. Other adjustments may be necessary to the scheduling policy, but not to the processes themselves.

4.3 Interleaved Dead States

A combination of competition and cooperation mechanisms can cause dead states if the scheduling mechanism allows interleaved access to preemptable instead of non-preemptable resources. At least two requests from at least two processes compete; service of the competing requests is interleaved.

In the lost update problem [2], two inputs are serviced at the same resource at the same time, each waiting for its output to perform an update. The scheduler places an additional wait restriction on simultaneous input requests from different users so that they cannot complete service as long as any of them have not had their outputs serviced. The first output writes to the resource, beginning its service, and waits for its input. The second output writes to the resource and conflicts with the first output. The later write is assigned a higher priority³ and overwrites the earlier one. The first output and its matching input are cancelled, entering into an inactive dead state. The second output and its input now complete service since there are no more outstanding outputs. Inconsistent retrieval [2] has similar characteristics.

³ This is not true for all outputs, for example, in broadcasting systems using C-Aloha.

Two processes may continuously alternate in moving an access mechanism to seek the location for disk input [5, 9]. The first process's seek performs its update, but must wait until its input is serviced. Its execution is interleaved with a second process. The second process's seek is given higher priority, preempting the first process's seek request, but is interrupted before issuing its read. The first process next detects the overwrite and reissues its seek, canceling the second process's seek, with this cycle continuing and no input completed. Note that resources are not being held; it would be possible for a third process to be scheduled in between these two and complete service, even though the first two processes repeatedly preempt each other's seek.

Resource pre-allocation and serial execution are effective prevention schemes for interleaved dead states. Database systems, for example, either pre-allocate data or else seek serially equivalent schedules. If resources can be preempted, roll back may be monitored, with scheduling adjusted following detection.

5. UNACCEPTABLE WAITS

Some of the literature defines deadlock as a state in which processes wait forever [11, 26], and thus never complete service. We expand our study to include unacceptable waits, states in which some processes do not complete service either by the end of Time or by a shorter time restriction. (It does not concern us if a wait is unbounded, such as in the protocols of stop signs or slotted Aloha, only that the wait is beyond a limit imposed by the user or the resource system.) These anomalies have the same causes and some of the same handlers as their corresponding subsets previously discussed.

5.1 Communication Unacceptable Waits

In communication dead states, users assign cooperation mechanisms for progress that can never occur. In communication unacceptable waits, cooperation restrictions prevent progress within a necessary time interval. For example, a process may have to wait to enter its critical section until an independent process has been serviced in the critical section [6]. If it is possible that the second process will not enter the critical section within a required time period, correction is required. In many cases, cooperation mechanisms that delay service are not incorrect; they ensure proper usage of the resource.

5.2 Scheduling Unacceptable Waits

In scheduling dead states, a scheduler assigns competition mechanisms that prevent the progress of processes. In scheduling unacceptable waits, the scheduler assigns priorities to conflicting processes that delay service of some or all of the processes within a given time constraint. For example, batch processes may be repeatedly given low priorities if interactive requests keep arriving. Device drivers may be delayed by higher priority real-time processes and lose data. For processes with the same urgency, scheduling waits (such as wall times or clock interrupts for inactive waits, or time quanta or hop counters for active waits) may signal the scheduler to employ aging mechanisms. On the other hand, a hard real-time process, which is aborted if it does not complete service within its time constraint, requires process-based priority assignments. In most cases, priority assignments are not incorrect; they are necessary to meet service requirements.

What happens if process arrival rate approaches or exceeds the maximum service rate of a system for a significant interval of time? We learn from queueing theory that waits become unacceptable using any scheduling discipline. Such a wait, called instability [1], is controllable chiefly by throttling mechanisms, an extreme form of aging, which often cause unacceptable delays in the user buffer.

5.3 Interleaved Unacceptable Waits

In interleaved dead states, each process that is holding resources cannot complete service based on a wait for another's service. If resources are locked, the scheduler always assigns newly arriving requests a lower priority, guaranteeing that processes currently being serviced will not be preempted. Then interleaved execution can cause resource deadlock. If resources are not locked, interleaved execution may cause the repeated preemption of processes currently being serviced and an interleaved dead state. In interleaved unacceptable waits, processes also execute requests in an interleaved order, with some requests unable to complete service because of their process's delayed requests. For example, in fragmentation-based congestion collapse in networks, a packet is broken into fragments and then must be reassembled before it is forwarded. Buffering or servicing of some fragments causes other fragments from each packet to be discarded, perhaps because of time-outs or overflowing buffers [8]. In reassembly deadlock, the buffers of some nodes are filled with partial

messages that are waiting for packets that cannot be accepted [12], causing congestion in the surrounding nodes. Congestion impairs the network, so that most packets time-out, are preempted, and/or overflow their buffers, and generally experience unacceptable delays. To control interleaved waits with fungible resources, heuristics are commonly used, such as maintaining a safe state by limiting resource commitment to 70%⁴ of line capacity, buffer space, or real memory frames. Alternatively, the roll back of processes may signal the deployment of such throttling mechanisms, for example, following increased activity to a paging disk in UNIX or the discarding of a packet in TCP.

6. A MODEL FOR COMPUTER SYSTEMS

We introduce a model to obtain an unambiguous definition of resource deadlock and the processes it contains, as well as of other critical terms in this paper. A description of the restrictions imposed upon processes and their requests, controlling their movements, allows us to differentiate between resource deadlock and other types of infinite waits.

6.1 The Process

P is a finite, nonempty set of processes in a computer system. A process, $p \in P$, is a set of requests that are bound together with restrictions in order to complete service as a unit. For example, a sequential process must have its requests serviced in order (ignoring RISC architecture); a concurrent process is restricted at coordination points as well. Each process has a unique identifier that specifies its user and itself. Processes of the same user share buffer elements. We recognize that processes may not require the service of all of their requests, such as statements within a loop that is never entered. For simplification, only unconditional requests are considered here.

6.2 The Resource

R is a finite, nonempty set of resources in a computer system. Each resource, $r \in R$, has a unique identifier and a buffer. Fungible elements share their resource's buffer and identifier. Each resource element has a finite number of linearly ordered units corresponding to instances of Time.

6.3 The Layers

All requests attempt to traverse a set, **M**, of ordered layers of a computer system (Figure 3):

- 1) A Process Conception layer, **PC**, is a set of requests that are continually reconceived until they can be submitted to higher layers. If requests move down to this layer, they are requesting to be reconceived, i.e., corrected.
- 2) A Process Buffer layer, **PB**, is a set of requests that have been conceived but have been postponed (buffered) before access to the resource service layer.
- 3) A Delivery Buffer layer, **DB**, is a set of requests that are being transferred between the process layers and the resource layers.
- 4) A Resource Buffer layer, **RB**, is a set of requests that are stored waiting for access to the resource.
- 5) A Resource Service layer, **RS**, is a set of requests that are being serviced and await completion of service.
- 6) A Completion layer, **C**, is a set of requests that have completed service.

We define an ordering relation, $>$, on **M**, such that $C > RS > RB > DB > PB > PC$.

For **M** and **M'** $\in M$, where $M > M'$, if $m \in M$ and $m' \in M'$ we say that $m > m'$ and m is in a higher layer. If m and m' are elements of the same layer, we say that $m == m'$.

The requirement of six layers is a simplification; for example, in memoryless broadcasting channels there is no **RB**. We model absence of layers by the movement of requests through them without delay. Alternatively, each layer may be subdivided further; for example, in the waterfall model of software engineering, **PC** is itself layered. In addition, the model can be considered to be circular, with **PC** equivalent to **C**, so that requests that complete service are reconceived for reuse [14].

⁴ The Seventy Percent Rule states that the cost of using a resource rises rapidly when more than 70% of resources are allocated. [18]

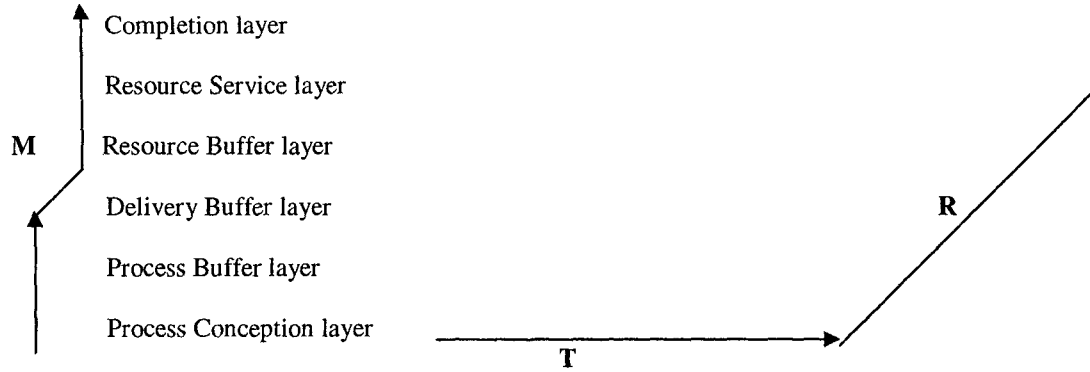


Figure 3. The Layers of the Computer System

6.4 The Request

Each request is a triple (m, r, t) , where:

$m \in M$ identifies the current position of the request in the system layers in its request for completed service. For fungible resources or buffers, it also identifies the specific fungible element to be assigned by the scheduler.

$r \in R$ identifies a named resource.

$t \in T$ identifies a discrete interval of Time, an initial subset of the natural numbers, bounded by the lifetime of the computer system.

All requests are either Output or Match [14] operations, identified by a predicate on the request that also contains a key value to be output or matched, as well as the process's identification. A Match request that has a wild card for a key value is called an Input.

6.5 The Computer System

A Computer System is a quintuple $\{M, P, R, T, F\}$, where M is a set of ordered layers, P is a set of processes, R is a set of resources, T is slotted time, and F is a function (called a scheduler in this paper) that controls the movement of requests through the single and composite layers of M . For each P , F assigns a mapping [13], $g_P: P \rightarrow P'$, such that:

$g_P(m, r, t) = (m', r, t)$, $m' > m$, if restrictions permit this mapping, (called upward mapping) else
 $g_P(m, r, t) = g_P(m', r', t')$, $m == m'$, $t' > t$, if restrictions permit this mapping, g_P recursively defined, else
 $g_P(m, r, t) = g_P(m', r', t)$, $m > m'$ (called downward mapping).

A request is mapped from the process conception layer, in order, to the completion layer as restrictions allow. If restrictions prevent mapping to a higher layer at some specific time, the request is rescheduled to the same layer for a later time unit, if possible; else it is mapped down to a lower layer where it awaits upward movement.

6.6 The Restrictions

At each layer, a request is assigned three types of restrictions:

1) A contingency predicate specifies a set of requests and a movement for each request. Each request's movement is contingent upon the actions of the requests in its contingency predicate. A request is removed from a contingency predicate only following the specified action. For example, a sequential request must wait for the previous request to be serviced before it can move to the resource service layer. In addition, all of a process's requests are contingent upon each other so that none can complete service unless they all are serviced. The abortion of a request, such as following a memory access violation, causes the abortion of its entire process.

2) Time predicates specify the minimum or maximum number of times that a request is rescheduled from a particular element of M . A restriction of a minimum number of rescheduling times will delay the request from moving to a higher layer. For example, waiting times for citizenship, cooking times, and the Ada delay statement specify a minimum rescheduling wait. A restriction of a maximum number of rescheduling times

limits the delay at the present layer. For example, if a time quanta in operating systems or a time-to-live field in IPv4 is exceeded, a process is mapped to a lower layer.

3) Priority predicates determine request mappings when conflict occurs. A request is enabled for upward mapping if its corresponding contingency set is empty and its minimum waiting time is 0. A request is enabled for rescheduling to the same layer if its corresponding contingency set is empty and its maximum waiting time is not 0. Conflict occurs if an output request and at least one other request with a different key value are enabled for mapping to the same resource or buffer unit. (For fungible resources, conflict occurs if there are more enabled output requests with different key values than there are fungible units.) The requested service or buffering will be provided at that time unit to at most one conflicting request that is assigned a priority value of high; the other conflicting requests must be assigned a priority of low at that time unit.

7. ANOMALIES OF A COMPUTER SYSTEM

The key terms in this paper are next defined in terms of our model:

An *anomaly* is a state that contains at least one process whose requests are mapped to a lower layer. A process that moves to a lower layer must repeat previous upward movements before it can complete service. We assume that a maximum rescheduling time is assigned to all requests based on the lifetime of the system, so that buffer and resource service requests that have not completed service move down to the lowest layer at the end of Time.

Preemption is an anomaly containing a process with a request that has begun service (mappings in **RS**) and is mapped down from **RS** because of a low priority assignment following conflict.

Abortion is an anomaly containing a process that is mapped down to **PC**.

Roll back is an anomaly containing a process that must repeat all previous mappings to **RS** as well as in **RS**.

Buffer Overflow is an anomaly containing a process that is mapped down from a buffer layer because a conflicting request has been assigned a priority of high.

A *time-out* is an anomaly caused by the expiration of a maximum rescheduling restriction.

A *collision* is an anomaly in which two or more conflicting requests are mapped to **RS**.

Throttling is a state in which a process's buffer requests are assigned a low priority for upward mapping.

A *dead state* is an anomaly containing at least one process that does not complete service.

An *unacceptable wait* is an anomaly caused by a time-out.

Follow within is a binary relation between two requests of the same process, $(m, r, t) \text{ } \$ (m', r', t')$, such that (m, r, t) is prevented from moving up to the next layer because of a contingency containing (m', r', t') , or $(m, r, t) \text{ } \$ (m'', r'', t'')$ and $(m'', r'', t'') \text{ } \$ (m', r', t')$.

Blocked by is a binary relation between two requests, $(m, r, t) \text{ } \& (m', r', t')$, such that (m, r, t) and (m', r', t) conflict over a non-preemptable resource⁵ and (m', r', t) is assigned a priority of high for it. (Since the resource is non-preemptable, $(m', r', t' + n)$ will be assigned a priority of high until the resource is voluntarily released.)

Dependent Upon is a binary relation between two requests, $(m, r, t) \% (m', r', t')$, such that $(m, r, t) \text{ } \$ (m'', r'', t'')$ and $(m'', r'', t'') \text{ } \& (m', r', t')$ or $(m, r, t) \% (m'', r'', t'')$ and $(m'', r'', t'') \% (m', r', t')$.

Resource Deadlock (RD) is a dead state containing at least two processes, whose requests have maximum rescheduling values set to the end of Time, such that there exists requests (m, r, t) in one process and (m', r', t') in another process, where $(m, r, t) \% (m', r', t')$ and $(m', r', t') \% (m, r, t)$.⁶

8. THE MODEL AND RESOURCE DEADLOCK

We have identified two waiting relationships in resource deadlock: Follow within, a cooperation mechanism, and Blocked by, a mutual exclusion mechanism. In the wait-for graph of Figure 4, two processes in resource deadlock compete for two resource units. The arrow within each process circle signifies a Follow within relationship and the composite arrows between two processes signify a Blocked by relationship.

⁵ Although we have repeated the requirement in [3] that resource deadlock requires non-preemptable resources, interleaved deadlock can also occur if a process not contained in the circular wait is allowed to preempt the blocking request and complete service, as long as the resource is then reallocated to the preempted request(s).

⁶ Resource deadlock with fungible resources requires this relationship between all processes with a request that holds a unit of a requested resource.

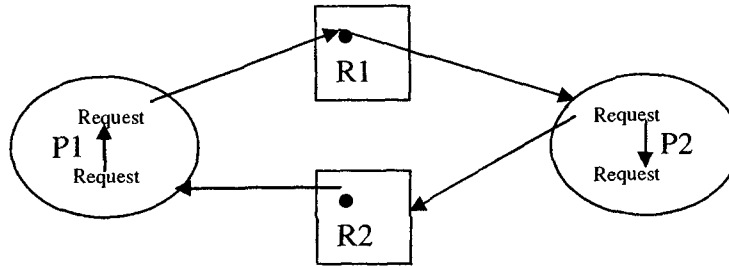


Figure 4. Modified Wait-for Graph for Resource Deadlock

The producer-consumer problem of the introduction contains two processes, each with a Follow within wait, one with a Blocked by wait, and the other with a Contingency containing a request of the other process. This anomaly is not a resource deadlock. Confusion occurs because the same construct has been used to implement both competition and cooperation synchronization.

We next discuss a dead state that appears to be ambiguous. In priority deadlock [12], a system pre-allocates resource buffers for message reassembly. All buffers at a receiving router have been reserved for low-priority messages. High-priority packets are scheduled to all of the bandwidth, but are discarded by the receiving router because of the lack of buffers. The sending router keeps timing out and resending the high-priority packets, so that low-priority packets cannot reach their pre-allocated buffers and complete their reassembly. A circular dead state occurs due to interleaved service of packets that are assigned resources needed by others in the circular chain. We see that the prevention schemes of (total) resource pre-allocation as well as resource preemption are applicable. Yet, are there inconsistencies? Can the scheduler break the dead state without roll back by simply raising the priority of low-priority packets. In addition, it appears that bandwidth is preempted, violating one of the preconditions [3]. On further examination, we see that bandwidth is repeatedly reassigned to the high-priority packets. In addition, high-priority packets are indeed rolled back each time they are refused buffer space at the receiving node. The deadlock can be broken only following roll back. This resource deadlock indeed satisfies our definition. Low-priority requests are Blocked by high-priority requests for bandwidth, while high-priority requests are Blocked by low-priority requests for buffer space. Messages cannot complete service since each of the blocking packets has been assigned a Contingency containing a blocked packet of its message.

9. CONCLUSION

A model of a computer system and a characterization of request movements enable us to distinguish between resource deadlock and other types of infinite waits. Our study has identified numerous inconsistencies and misunderstandings in this area, which this paper was designed to correct and clarify. Our model has led to a uniform treatment of dead states, including their causes and methods of control.

REFERENCES

- [1] M. Andrews, B. Awerbuch, A. Fernandez, T. Leighton, Z. Liu, and J. Kleinberg, "Universal-stability Results and Performance Bounds for Greedy Contention-resolution Protocols," *Journal of the ACM*, vol. 48, no.1, pp. 39-69, Jan. 2001.
- [2] P. Bernstein, and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computer Surveys*, vol.13, no. 2, pp. 185-211, June 1981.
- [3] E.G. Coffman, E.G., M. J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67-78, June 1971.
- [4] E. G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice Hall, Englewood Cliffs, NJ, 1973.
- [5] W.S. Davis and T. M. Rajkumar, *Operating Systems, A Systematic View*, 5th ed., Addison-Wesley, Reading, Mass., p. 123, 2001.
- [6] E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, Academic Press, London, 1965.
- [7] L. Dowdy and C. Lowery, *P.S. to Operating Systems*, Prentice Hall, Englewood Cliffs, NJ.1993.

- [8] S. Floyd and K. Fall, "Promoting the Use of End-to-end Congestion Control in the Internet," *IEEE/ACM Transactions on Networks*, vol. 7, no. 4, pp. 458-472, Aug.1999.
- [9] I. M. Flynn and A. M. McHoes, *Understanding Operating Systems*, Brooks/Cole, Australia, pp. 109-110, 2001.
- [10] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179-196, Sept. 1972.
- [11] D. Horner, *Operating Systems, Concept and Applications*, Scott, Foresman and Co. Glenville, Ill., pp.160, 105, 182, 1989.
- [12] W. S. Lai, "Protocol Traps in Computer Networks- a Catalog. *IEEE Transactions on Communications*," Com-30, no. 6, pp. 1434 -1448, June 1982.
- [13] G. N. Levine, "The Control of Starvation," *International Journal of General Systems*, vol.15, pp. 113-127, 1989.
- [14] G. N. Levine, "A Model for Software Reuse," Proceedings of the 5th Workshop of Specification of Behavior Semantics, OOPSLA '96, pp. 71-87, Oct. 1996.
- [15] J.C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *ACM Trans. on Computer Systems*, vol.15, no.3, pp. 217-252, Aug. 1997.
- [16] G. Nutt, *Operating Systems, a Modern Perspective*, 2nd edition, Addison-Wesley, Reading, Mass., pp.150, 279, 2000.
- [17] J.R. Pinkert and L. L.Weat, *Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, p.48, 1989.
- [18] P. J. Plauger, "On Being Fast Enough," *Embedded Systems Prog.*, vol.4, no.1, pp. 81- 92, Jan. 1991.
- [19] D. J. Rosenkrantz, R. E., Stearns, and P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. on Database Systems*, vol.3, no.2, pp.178-198, June 1978.
- [20] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts*, 6th edition, Addison-Wesley, Reading, Mass., pp. 204, 243, 244, 266, 2002.
- [21] W. Stallings, *Operating Systems, Internals and Design Principles*, 3rd edition, Prentice Hall, Englewood Hills, NJ, pp. 254, 1998.
- [21] A. Tanenbaum, *Computer Networks*, 4th edition, Prentice Hall, Upper Saddler River, NJ. 2003.
- [23] A. Tanenbaum, *Modern Operating Systems*, 2nd edition, Prentice Hall, Upper Saddle River, NJ, p. 185, 2001.
- [24] A. Tanenbaum, *Operating Systems, Design and Implementation*, 2nd edition, Prentice Hall, Upper Saddle River, NJ, pp. 67-69, 1997.
- [25] Y. C. Tay and W. T. Loke, "On Deadlocks of Exclusive AND-requests for Resources," *Distributed Computing*, Springer-Verlag , #9, pp. 77-94, 1995.
- [26] D. Tsichritzis, and F. Lochovsky, *Data Base Management Systems*, Academic Press, London, p. 260, 1977.
- [27] R. Turner, *Operating Systems Design and Implementations*, Macmillan Pub. Co., London, p. 140, 1986.
- [28] R. J. Van Glabbeek, "Notes on the Methodology of CCS and CSP," *Theoretical Computer Science*, pp. 329 - 349, 1997.
- [29] H. Wu, W. Chin, and J. Jaffar, "An Efficient Distributed Deadlock Avoidance Algorithm for the AND Model," *IEEE Trans. on Software Engineering*, vol.28, no.1, pp. 18-29, Jan. 2002.