

## **Defining Deadlock with Fungible Resources**

Gertrude Neuman Levine  
Fairleigh Dickinson University  
levine@fdu.edu

### **Abstract**

In a previous paper we discussed resource deadlock within a classification of all dead states. We differentiated between the different types of anomalies that are called deadlock in the computer science literature. Our primary aim was to correct inconsistencies that are currently being taught in operating systems' textbooks. We introduced a model that allowed us to precisely define deadlock, but we only provided a partial definition of resource deadlock, which, as we noted, was limited to deadlocks with uniquely identifiable resource units. In response to readers of OSR, we now distinguish between different types of resource deadlock and provide a comprehensive definition. We also suggest that it may be desirable to maintain separate definitions.

### **Introduction**

Resources can be characterized as being uniquely identifiable or as containing multiple interchangeable units. If a resource is uniquely identifiable, it contains exactly one unit. Examples of such "unique" resources include Ethernet MAC addresses and bits on an interrupt vector. We borrowed the term "fungible" from law to denote resource units that are interchangeable, that are of equal value to the users and the resource system. Examples of fungible resources include unclassified buffers, identical blank disks, and data copies in a replicated database using a majority rule voting scheme.

As an example of a simple deadlock with a resource that contains two fungible units, consider the wait-for graph in figure 1: Process A may require either 1 or 2 units of a fungible resource that consists of exactly two units. Let us assume that process A is requesting 1 unit of R1. Deadlock occurs as processes B and C both wait for a unique resource held by process A, assuming that resources are not shareable, nor dynamically created, prematurely released or preempted.

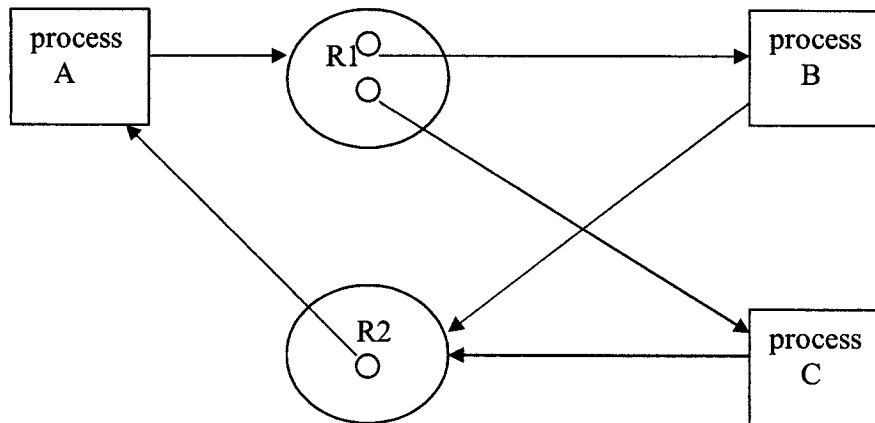


Figure 1: a resource deadlock that includes fungible resources

Defining Deadlock [3] listed the four necessary and sufficient conditions for (resource) deadlock as stated in Coffman, Elphick, and Shoshani [1] and in some textbooks [2, 6].

- 1) Processes claim exclusive control of the resources they require.
- 2) Processes hold resource units while waiting to acquire others.
- 3) Resources can be released only by the processes holding them; they cannot be preempted.
- 4) A circular chain of processes exists, such that each process is waiting for a resource continuously held by the next process in the chain.

The last condition, however, is not sufficient for resource deadlock with fungible resources [5]. The phrase, “next process in the chain,” appears to imply a linear order of processes, but in figure 1, such an ordering is not apparent. We could designate a chain such as:

A waits for R1 blocked by (B or C) wait for R2 blocked by A.

This circular chain is almost sufficient to characterize deadlock since it contains a holding process for each resource unit required. One difficulty with this characterization is that B and/or C may be waiting for resources other than R2, each of which is held by multiple processes that wait, perhaps indirectly, for resources held by A. The chain can become a large threaded tree.

Of greater concern is that deadlock need not occur if either B or C is waiting for a resource that is not in the chain, or is waiting for a consumable resource, such as a message, that can be created dynamically. A circular chain, therefore, is insufficient to characterize resource deadlock with fungible resources. The chain must be closed, i.e., all branches of the tree that could break the deadlock must cycle back to previously visited processes. If one unit at a time is allocated, both branches are involved. If process

A requests two units of R1 simultaneously (for example, if its hardware instruction requires two memory frames), then one branch cycling back to process A is sufficient to close the chain. (We also limit each process to a single thread of control, so that, once blocked, a process will not be able to create resource units dynamically. This was surely what was meant by “task” in [1].)

Although papers and texts combine the treatment of resource deadlock with fungible and with unique resources, there are some differences in the control mechanisms that are appropriate. Deadlock with unique resources can be prevented by ensuring that requests follow a linear order of resources. This method cannot be used for a fungible resource, since ordering units negates their fungibility. (It is, however, useful for the prevention of circular chains among different types of fungible resources.) On the other hand, the capability of adding backup resource units “on the fly” can prevent deadlocks with some fungible resources, such as storage or transmission capacity, but not with unique resources. Maintaining a safe state is an algorithm applicable for both types of resource deadlock, but, besides resource pre-allocation, is used for neither. Some fungible resource systems, such as spooling systems, frequently rely on heuristics of an avoidance scheme, only allowing new processes to enter the system if perhaps up to 70% of units have been committed.

Detection schemes are clearly more difficult with fungible resources. To identify resource deadlock, we must follow multiple paths to processes holding requested units, and to the resources that they request, until the chain is shown to be open or closed. Deadlock in replicated databases is typically defined for unique resources, with requests characterized as “and-requests” [6]. Yet, if a majority rule voting algorithm attempts to lock replicated data sites, then a deadlock detection algorithm must investigate or-requests and traverse multiple branches to determine if a closed chain exists. Note that three branches may be sufficient if there are five sites in all.

Recovery is also more difficult with fungible resources. It is harder to identify processes that are in the closed chain and killing a process that holds a requested fungible resource may not break the deadlock.

Yet, most of the characteristics of both types of resource deadlock are the same. Resource deadlock is typically caused by the interleaved execution of asynchronous processes requesting resources while cyclically holding other non-preemptable, non-shareable resource units, and is controllable by many of the same prevention schemes, such as serial execution, resource preallocation, and making resources preemptable.

### **Defining Deadlock**

In “Defining Deadlock” [3], we introduced a model of computer systems in order to differentiate between different types of dead states called “deadlock.” When we defined resource deadlock, however, we only treated deadlock with unique resources. We now extend the definition to apply to fungible resources. We treat separately the case in which resource units must be requested and allocated one at a time, and the case in which they

can be requested and waited for in multiple units. We include definitions from our previous paper in italics:

*Follow within is a binary relation between two requests of the same process,  $(m, r, t)$  &  $(m', r', t')$ , such that  $(m, r, t)$  is prevented from moving up to the next layer because of a contingency containing  $(m', r', t')$ , or  $(m, r, t)$  &  $(m'', r'', t'')$  and  $(m'', r'', t'')$  &  $(m', r', t')$ . This relation defines the wait between requests of the same process. It is necessary to express relations in terms of requests. An entire process in resource deadlock must wait for its blocked request, so that it cannot create new units of consumable<sup>1</sup> resources, such as messages [4]. A process was defined [3] as a sequence of (unconditional) requests that must be serviced in order; if a request is blocked, all following requests must wait. (This restriction will be formally expressed by %.)*

*Blocked by is a binary relation between two requests,  $(m, r, t)$  &  $(m', r', t)$ , such that  $(m, r, t)$  and  $(m', r', t)$  conflict over a non-preemptable resource and  $(m', r', t)$  is assigned a priority of high for it.*

According to the definition provided in [3], conflict occurs when there are more enabled output requests (for the same resource at the same time) than there are resource units. A resource unit is allocated to at most one of conflicting requests (the mutual exclusion precondition). Although this definition covers both fungible and unique resources, there is a substantial difference between the blocking relations; unique resources are uniquely held, while units of a fungible resource may be held and released by many different processes. In addition, & does not specify whether multiple units can be requested and waited for simultaneously.

*Dependent Upon is a binary relation between two requests,  $(m, r, t)$  %  $(m', r', t')$ , such that  $(m, r, t)$  &  $(m'', r'', t'')$  and  $(m'', r'', t'')$  &  $(m', r', t')$  or  $(m, r, t)$  %  $(m'', r'', t'')$  and  $(m'', r'', t'')$  %  $(m', r', t')$ . This relation extends the wait to all blocking requests in a chain of process dependencies (% is transitive). It also requires that the process's following requests are blocked, and therefore that the process is blocked.*

We next define relations in terms of processes. We assume that requests in process  $p$  are Dependent upon requests in process  $p_i$ , all  $i$ , that no requests in these processes have time or contingency constraints forcing them to release held resources, and that  $p$  and  $p_i$  are elements of  $P$ , our process set.

We define a blocking relation,  $\&^{r/N}$ ,

$p \&^{r/N} \{p_{1,n_1}, p_{2,n_2}, \dots, p_{i,n_i}, \dots\}$ ,  $\sum n_i = N - r + 1$ , where  $r, i, n_i > 0$ , such that process  $p$  is requesting  $r$  units of a resource consisting of  $N$  units, and is blocked by processes  $p_i$ , each of which is holding  $n_i$  units of the resource.

- a) If a resource has only one unit (i.e., is unique), then at most one unit can be requested and blocked ( $r, N, i$ , and  $n_i$  equal 1).  $\{p_{1,1}\}$  is the unique blocking set.

---

<sup>1</sup> A request for a message specifies the process that must create the message (a mailbox would specify a wild card), similar to the specification of a printer that outputs a paper or a disk drive that reads a sector.

- b) If a process requests or is allocated one unit at a time, then  $r = 1$  and the resource must consist of exactly  $\sum n_i$  units ( $N = \sum n_i$ ). Each of the blocking processes is crucial to delaying  $p$ 's service, creating a unique blocking set.
- c) If a process waits for  $r$  units at a time,  $r > 1$ , then, of the  $N$  resource units, there may be up to  $r - 1$  units irrelevant to the blocking condition. We therefore have restricted the relation to a minimal blocking set of  $N - r + 1$  units. For example, in figure 1, if process A is requesting 2 units of R1, and one unit continues to be held by process B, process A will be blocked even if process C is forced to release its unit. Minimal blocking sets are therefore  $\{B\}$  and  $\{C\}$ .  $\{B, C\}$  is a blocking set, but not minimal.
- d) For systems that use mechanisms that allow simultaneous waits for multiple types of resources, such as resource preallocation, the blocking relation for processes must be extended.

Dependent Upon is a binary relation between two processes, such that  $p \% p'$  iff  $p \in P'$  ( $P'$  a minimal blocking set of  $p$ ) and  $p' \in P'$ , or if  $p \% p''$  and  $p'' \% p'$ . The relation,  $\%$ , is transitive, but not commutative. The arrows begin with a single process,  $p$ , and branch out to its dependencies. The transitive closure of the Dependent Upon relation for processes in minimal blocking sets is called a chain.

- a) If resources are unique, then  $\%$  defines a linear list of dependencies between processes.
- b) If resources are fungible and are requested one unit at a time,  $\%$  defines a tree of dependencies, such that  $p$  is dependent upon each process in the chain of blocking sets. (Tree traversals are appropriate for detection of resource deadlocks with all fungible resources.)
- c) If resources are fungible and processes can wait for multiple units at one time, then the graph is more complex.  $p'$  could be a member of several different minimal blocking sets, but  $p \% p'$  states that  $p'$  is an element of a specific minimal blocking set,  $P'$ . Then  $p$  is Dependent upon all processes in  $P'$ , and on all processes in a minimal blocking set for each, if such exists, etc. Note that some resource units may be free, or freed by processes not in the minimal blocking set. Then, as our chain branches out from the initial minimal blocking set, some branches (arrows, edges) may circle, unfettered, through free resource units.
- d) It is possible to have mixtures of different kinds of resources and requests in a chain of dependencies.

*Resource Deadlock (RD) [with unique resources] 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  $(m', r', t') \% (m, r, t)$ ...[but] Resource deadlock with fungible resources requires this relationship between all processes [in a minimal blocking set] with a request that holds a unit of a requested resource.*

A **Resource Deadlock** with fungible resources (RDF) is a dead state consisting of a process  $p$ , where  $p \in P$ , and a closed chain of blocking sets containing processes  $p_i$ , where  $p \in P$  and  $p_i \in P$  for all  $p_i$  in the chain.

A definition for deadlock with fungible resources was not included in our previous paper partly because the paper was already too complex, but also because there were outstanding issues. One issue, which is still not resolved, is that of a unary resource deadlock.

Our definition does not require the existence of two processes in RDF. Let us consider a specific unary dead state with a fungible resource: process A is executing on a personal computer and holding all of the memory available to user processes, yet cannot complete service since it needs more memory. You may say, with indignation, that of course this is not a resource deadlock. This process is not involved in a circular wait with other processes. According to "Defining Deadlock," resource deadlock is an anomaly of traffic control caused by interleaved execution of at least two processes' requests. And, indeed, deadlock prevention and detection algorithms do not apply. Note, however, that this dead state does not mandate abortion, in which the process is sent back to the user for correction, as we stated was required for "communication dead states." Instead, the process can be sent to the user buffer layer and submitted to another resource that contains more memory units. The process is incorrect only in its choice of resources. In addition, in a uniprocessing virtual memory environment, a process that requires considerably more memory than is available will enter a period of thrashing. But thrashing is a form of congestion or unacceptable interleaved wait, which is a superset of interleaved deadlock [3]. Thus, we are unwilling to exclude unary deadlocks from RDF by requiring that it contain at least two processes.

Then why not use the second definition for resource deadlock with unique resources? Consider the unary dead state in which a process makes two successive locking calls to a binary semaphore:  $P(S); P(S)$ . Such a state appears to satisfy the definition for RDF but is not a resource deadlock. We therefore specifically exclude unary dead states from the definition of RD.

Another open issue is which processes are contained in deadlock. Does Resource Deadlock include only the processes in the closed chain, or does it also include other processes whose service is prevented due to priority or contingency predicates involving, perhaps indirectly, some process in the closed chain? Our definition has specifically allowed for the latter.

## **Conclusion**

The purpose of "Defining Deadlock" [3] was to distinguish between resource deadlock and other types of dead states. Our current paper deals with two different types of resource deadlock, differentiated by resources that are fungible or unique, and between deadlock with fungible resources that are blocked singularly or in multiple units. We have provided a definition that could encompass all types, but doing so would blur

important distinctions. Two definitions of resource deadlock are therefore included in this paper.

### **Acknowledgments and Corrections**

My thanks to Dr. David Warme and Irene Frawley for their encouragement and to Dr. Gary Nutt for his correction of “Defining Deadlocks.” Footnote #1, in which the four necessary conditions for resource deadlock were deemed sufficient is inconsistent with footnote #6; the preconditions are sufficient only if resources are unique and are not dynamically created.

### **References**

- [1] E.G. Coffman, M. J. Elphick, and A. Shoshani, “System Deadlocks,” *ACM Computing Surveys*, 3 (2), pp. 67-78, June 1971.
- [2] I. M. Flynn and A. M. McHoes, *Understanding Operating Systems*, Brooks/Cole, 2001, p. 112
- [3] G. N. Levine, “Defining Deadlocks,” *Operating Systems Review*, ACM Press, 37(1), pp. 54-64, January 2003. [http://alpha.fdu.edu/~levine/course\\_offerings/september.doc](http://alpha.fdu.edu/~levine/course_offerings/september.doc)
- [4] G. Nutt, *Operating Systems, a Modern Perspective*, 2<sup>nd</sup> edition, Addison-Wesley, 2000.
- [5] J. R. Pinkert and L. L. Wear, *Operating Systems*, Prentice Hall, 1989.
- [6] W. Stallings, *Operating Systems*, 4<sup>th</sup> edition, Prentice-Hall, 2001, p. 273.
- [7] Y. C. Tay and W. T. Loke, “On Deadlocks of Exclusive AND-requests for Resources,” *Distributed Computing*, Springer-Verlag , 9, pp. 77-94, 1995.