CGTM 63
April 1969
Bob Russell

INTERRUPT SYSTEMS

There have been several articles written that describe on-line data acquisition and control systems (1,2,3,4,5), and in all of these systems the computer depends heavily on interrupt capability for interacting with its real-time environment. Interrupts are important in time-sharing systems, and are now commonly a part of the I/O structure on nearly all 3rd generation computing systems. The purpose of this memo is to take a closer look at interrupts. We will delimit several different uses of this technique, examine its applications in existing computers, and suggest some modifications we feel should be included in the next generation. The ultimate aim is to show how interrupt handling should be integrated with the design of the total operating system in order to achieve consistent scheduling and resource allocation.

Existing Hardware

We begin by giving imprecise definitions of the concepts: interrupt, trigger, interrupt class, interrupt mask, priority level. These will be described by referring to their implementation on existing computers.

An interrupt is a method of forcing an instruction processor to break its current execution sequence and to begin a new one. This "sequence break" or "processor switch" is not to be confused with the "instruction-fetch sequence breaks" caused by jump and subroutine-call instructions.

Jump instructions are inserted by the programmer to redefine the "instruction successor function"; that is, to direct the processor to the memory location which contains the next instruction of the execution sequence. The important point is that they are part of the execution sequence and are necessary to map the linear succession of instruction execution by the processor into the non-linear arrangement of these instructions in storage. They are not switching the processor from one sequence to another, but are guiding it through memory in a manner planned by the writer of the sequence.

Interrupts break the instruction execution sequence planned by the programmer, and may occur anywhere in the sequence (beyond the control of the writer of the sequence). They are not merely a way of guiding a sequence through memory, but constitute a complete change of context. In the usual case of a single instruction processor, the processor switch is initiated by a signal to the processor called an interrupt trigger, which is generated in a myriad of possible ways, both internal and external to the processor operations. Typically an interrupt trigger causes the processor to store in a fixed memory location a pointer to the next instruction in the sequence it was processing and begin decoding a different instruction sequence indicated by the fixed location. Depending on the machine, various parts of the processor information will be automatically saved, ranging from just the program location counter on the IBM 1800 to all program registers and status register on the CDC 7600. Machines also vary in whether they actually transfer control to the specified location, force an execution of the instruction there (IBM 7094), or use

the contents of the location as an address to which control is transferred (IBM 1800).

Which fixed memory locations are used depends on the cause of the interrupt. Each source is hardwired to a fixed memory cell or cells. Often many individual sources are treated as a single interrupt class and are all hardwired to the same location. Many computers like the SDS 9300 or IBM 7094 are wired so that all devices on the same data channel are treated as a single interrupt class and therefore all use the same fixed memory location. Different channels however, are assigned different fixed memory cells.

There are usually mechanisms by which the processor can decide which interrupt sources will be allowed to interrupt and which will not. This is usually accomplished by an "interrupt mask register" which may or may not be visible to the programmer. Each bit in the register corresponds to an interrupt source (or class, depending on the machine) such that when the source sends a trigger signal, the interrupt actually occurs only if the bit is on. In this case the interrupt class is said to be unmasked, enabled, or armed. If the bit were off, the interrupt is masked, disabled, or disarmed, so that the processor would not be aware of any trigger and would not break the sequence of instruction processing. Whether or not a masked trigger is remembered until the processor decides to unmask that source depends on the machine. Some, such as the SDS 9300, allow the processor to specify both "mask and forget" (disarm) and "mask and remember" (disable). Blocking and unblocking can occur on the basis of individual conditions, individual devices, interrupt classes, priority groups, and all together, depending on the machine. Many machines, such as the 7094 and 9300 allow a permanent level and a temporary level of interrupt control. On the permanent level each device is individually enabled or disabled, while the temporary level indicates either block everything or return to the permanent level specifications.

In most machines, once an interrupt has occurred, (i.e., has caused a processor switch), all other interrupts in that interrupt class are automatically blocked. If they were not, a second trigger, before processing of the first one was complete, would cause the information from the first interrupt in the fixed memory location to be overwritten and lost. This is a special case of the more general re-entrant subroutine problem. To allow re-entrant programs requires a stack mechanism for moving and restoring linkage information without loss. Since most computers do not have automatic stacking, the only way to prevent a second interrupt to the same fixed location is to block all other triggers in the same class until the first interrupt is complete. (The B5500 is a notable exception since it has a stack and automatically pushes all relevant processor information into the stack when an interrupt occurs.)

What about triggers in other interrupt classes? Most machines arrange the interrupt classes into priority levels such that triggers in higher priority levels can interrupt the processor when it is processing an interrupt of lower priority, but triggers of lower priority classes must wait until the processor has finished processing all higher priority interrupts. In some computers such as the 7094, several different classes are assigned to the same priority level (an interrupt on any data channel automatically blocks all triggers from all other channels). Some computers permit the automatic blocking (of other triggers on an active level) to be overridden by the program before the interrupt processing has actually been completed. The "restore channel traps" instruction on the 7094 is an example of this.

Interrupt types

The sources of interrupt triggers vary widely. Almost anything can be designed to generate a trigger signal that causes an interrupt in the processor. We can however classify most sources according to the type of interrupt, by which we mean the type of information being conveyed by the trigger signal and its relationship to the execution sequences of the processor.

Type 1) I/O Interrupts

These are the most common interrupt sources and on some machines are the only ones. In most I/O operations the processor will activate a device (tape drive, card reader, etc.) and initiate a data transmission between that device and storage. Hardware in the device then assumes control of the transmission while the processor is free to continue executing instructions unrelated to the transmission. Since the I/O device is now operating asynchronously to the processor, it must have some method of communicating to the processor when it is done, when it needs more data, or when it has detected an unusual condition (parity error, end of file, etc.). The simple solution is to send an interrupt trigger to the processor, thereby diverting its attention from an unrelated instruction sequence back to instructions relevant to operation of the device.

Type 2) Explicit demand for operating system functions

Often called "supervisor calls" or SVC's. It is a mistake to call these true interrupts since they do not really cause a break in the current instruction sequence, but are in fact just another instruction in the sequence. What is desired is the ability to switch from "user" mode to "system" mode, which on some machines is mistakenly associated with switching from uninterrupted to interrupted mode respectively (the 360 for example). We will discuss this point in some detail later.

Type 3) Implicit demand for operating system intervention

Includes paging faults, elapsed time notification, and any other conditions which notify the operating system that its scheduling and resource allocation functions must be invoked. The instruction sequence being processed at the time of these interrupts should not know of their existence, since they are intended solely for scheduling convenience by the operating system. Therefore once the indicated allocation has been performed, the interrupted task should be allowed to continue.

Type 4) Defective hardware detection

Includes parity errors, unit failures, power loss detection, and any other conditions which indicate that the hardware is not functioning as designed. No processing should be allowed to continue until corrective action has been taken by the machine operator (i.e., the system cannot recover by itself).

Type 5) Defective instruction sequences

Includes op-code checks, memory access violations, privileged instruction execution, bounds' violation, and any other conditions which indicate that the instruction sequence being executed by the processor is faulty and should not be allowed to continue. The operating system should therefore not allocate the processor to that instruction sequence any more (the sequence cannot recover by itself, nor can the operating system correct the error in any rational way).

Type 6) Defective data detection

Includes overflow, underflow, divide by zero, illegal characters in conversion strings, and any other circumstance where the machine does not have the capability to handle the data as instructed. This information is valuable to the task being run, since in many cases the task is prepared to deal with the situation when and if it occurs. The interrupt is simply a convenient mechanism for detecting the unusual piece of data without repeatedly testing for it, and for forcing a break in the processing from one point in the instruction sequence to another in the same task. The operating system really does not have to know when these occur. However in most machines the operating system is used to handle the details of the sequence switching within the same task. (for example the "ON" statements of PL/I)

Type 7) Debugging aid interrupts

Break-point address detection, instruction tracing, transfer detection, and any other of the usual methods by which the progress of the processor through an instruction sequence is monitored. Here the interrupt mechanism has been adapted for "spying" or "snooping" on the execution of a program. This is superimposed upon the "normal" unmonitored control flow by the user of the task in execution, usually to determine more about the events leading to occurrence of interrupts of type 5 or 6. Again the operating system need not know about these interrupts but is usually employed to switch the processor between monitored and monitoring programs. The monitoring routines in fact are often made part of the system so that they have access to type 5 interrupts as well as type 6 and type 7. Also the instructions which turn on and turn off the tracing mode are often privileged instructions which must be controlled by the operating system.

Type 8) Attention interrupts

Signals originating from activities unrelated to and uncontrolled by the computer, but requiring interaction with the computer. The sources are active devices (data acquisition equipment, human operators, other computers, etc.) that control independent, asynchronous functions of their own and require the aid of the computer as part of this function. The occurrence of these triggers is usually totally unpredictable to the computer operating system and totally outside its control. Attention triggers are seldom direct consequences of any previous action by the computer, as are all other types, but are instead signals to initiate new action.

Type 9) Interprocess interrupts

The trigger for this type of interrupt is generated by instructions in one sequence in order to initiate or inhibit the progress of another independent instruction sequence. Some machines such as the IBM 1800 implement the triggers as "programmed interrupt" instructions, while others like the SDS 9300 require an improvised arrangement in which specially designed external hardware accepts an output control signal from a program and turns it into an "attention" interrupt trigger. Task A can cause suspension or termination of task B by either blocking all interrupt triggers intended for task B, or by causing the operating system to interrupt task B and not allow it to be rescheduled (see Fabry (8)).

Type 2 interrupts (SVC's) are an example of a special use of this type of interrupt restricted in the following manner: (a) the sequence being initiated by the SVC is always a program in the operating system; (b) progress in the current sequence is suspended while the new sequence is being executed. In the more general case neither of these conditions is required.

Type 9 interrupts represent a special use of the standard interrupt mechanism in order to accomplish interactions between processes similar to those discussed by Dijkstra (6) and Lampson (7). In their schemes, one process causes the activation or deactivation of other processes by setting or resetting a switch called a "semaphore" by Dijkstra and a "wakeup waiting switch" by Lampson. Interprocess interrupts are an implementation of these interactions restricted to conform with current interrupt hardware.

Traps

A distinction is often drawn (Fabry (8), Lampson (7)) between "traps" (or "exceptions") and "interrupts", which is based on the notion of a computational process, a concept that is frequently defined intuitively (see Dahm (9), Dijkstra (7), Fabry (8), Lampson (7)). "Interrupts" arise from sources external to the process for which they are intended. The source may be I/O units, the operating system, external "attention" sources, or other processes. "Traps" are forced switches of the processor from one instruction sequence to another in the same process, due to a condition arising from the execution (or attempted execution) of an instruction in the first sequence. Both the cause and the effect are internal to a single process.

Traps are usually generated by exceptional conditions arising within a process either due to a faulty instruction sequence (type 5) or faulty data (type 6). "An exception can be viewed as a kind of implicit error return associated with certain instructions" (Fabry (8)). The trap signal causes the processor to switch to another instruction sequence within the same process in order to correct the error in the data and/or instructions of the main sequence. The occurrence of a trap is highly synchronized with the actions of the processor on an instruction sequence. Thus type 7 interrupts are also called traps, since the debugging routines are considered part of the process in execution and must have access to the processor stateword of the instruction sequence being executed, since this stateword will contain information needed to debug the instruction sequence.

True interrupts, on the other hand, are asynchronous to the execution of a process' instruction sequence and are directed at a particular process regardless of the process currently in execution. The new instruction sequence is independent of the sequence in execution at the time of the interrupt, and the processor stateword should therefore not be accessible to the new instruction sequence.

The processor switching that occurs in a trap may or may not involve the operating system, depending on the implementation. Data spaces, addressing maps, and allocated resources will not change as the result of a trap, since these items are associated with processes, and the process does not change with the switch from one instruction sequence to another. Interrupt servicing, however, may require a great deal more than just processor switching; the new process may require allocation of new resources, pre-emption of resources from other processes, and inhibiting of still other processes. Consequently interrupts invariably require the intervention of the operating system to perform the necessary bookkeeping functions.

Interrupt functions

What's wrong with current interrupt systems? About the only critical review in the literature appears in Lampson's article in CACM (7). He lists 4 areas in which current systems are poorly designed. The first 3 points can be summarized as follows: there are usually two schedulers - one the hardware priority interrupt "scheduler", the other the software background scheduler - with no convenient inter-communication between them. The two are often quite dissimilar, with software scheduling usually very slow, cumbersome, but somewhat flexible; the hardware scheduler fast, efficient, and inflexible. Further, the interrupt scheduler invariably takes precedence over the normal scheduler because interrupts are still considered exceptional cases which must be handled immediately. It turns out that this is not always true. The scheduler dichotomy results in a task dichotomy - basic differences in the type and capabilities of interrupt tasks and "normal" tasks. Interrupt tasks are often forced to be part of the system, which means they run in privileged or supervisor mode, often without adequate protection safeguards, but have limited access to the system functions available to "normal" processes, such as formatted I/O, etc. These tasks are often required to be permanently resident in core storage, whereas "normal" processes can be swapped between core storage and secondary disk or drum storage. This dichotomy is often too clear in many computers, where normal tasks can be written in Fortran or other high-level languages, but interrupt routines must be hand-coded in assembly language. There are seldom adequate mechanisms for re-entrant subroutines, which compounds the difficulties of writing interrupt routines.

We strongly concur with these objections, and will try to explain how these difficulties arose and how they can be resolved. We begin by investigating in detail the functions which interrupts currently perform. There are at least 5.

1) Detection: the essential purpose of all interrupts is to detect the occurrence of some condition, be it external to the computer (types 1,8),

or generated by actions of the computer itself (types 4-7). The detection mechanism operates independently of the programmable processor functions, and is invariably, though often incorrectly, implemented as hardware that simply detects the presence or absence of a trigger. There is no way in which the program can affect the detection criterion by specifying, for example, a boolean combination of triggers, or a real-time sequence of various trigger signals, as the conditions which must be satisfied before the interrupt is initiated. Quite simply the trigger detector is non-programmable. Therefore detection is currently synonymous with initiating the other functions of the interrupt hardware, whereas it might be desirable to impose a level of decision-making between the detection and initiation functions.

2) Scheduling: this is the hardware scheduler to which Lampson referred. There are several components of this scheduler. a) if 2 triggers arrive simultaneously the hardware decides which to satisfy and which to remember until later; b) it determines the priority of each trigger, and compares this with the "priority level of operation" on which the processor is busy. The decision is then made whether to switch the processor in response to the new trigger (priority of the trigger greater than current level of operation) or to remember the trigger until the current operation releases the processor (current level greater or equal to priority of the trigger). The priority assignments are invariably hardwired, thus eliminating dynamic assignment or reassignment by programming. c) the hardware maintains a mask register which says which triggers should be permitted to initiate a processor-switch, which should be held until later (blocked but not lost), and which should be totally ignored. Changing this mask register is the only mechanism by which the software scheduler and "normal" tasks can affect the hardware interrupt scheduler. (In effect, this block-unblock operation crudely alters the hardwired priority assignments.) d) the hardware scheduler maintains the fixed "interrupt class" groupings of the various trigger sources. The assignment of these classes to priority levels is also immutably fixed.

The interrupt hardware is really scheduling only a single resource -- the instruction processor, which is currently (and we feel incorrectly) considered a special type of resource. For this resource the hardware maintains a queue of pending interrupt requests, which of course has its own enqueuing and dequeuing rules that are totally separated from the software queuing disciplines for the same processor resource.

3) Allocation: as mentioned under detection, the purpose of triggers is invariably to activate processor-switching hardware. This hardware is just a resource allocation mechanism for the processor resource, since it handles all necessary saving and restoring involved in processor state switching. Of course, what components of the processor resource are saved is predetermined in the hardware. Where they are saved and where new ones are obtained is also fixed, depending on the interrupt class. Unfortunately this mechanism is totally different from and toally independent of the "normal" software processor allocation mechanism. Hence the processor becomes a resource with 2 unique and often conflicting allocators, a decidedly unhealthy state of affairs. Lampson's fourth criticism of interrupt systems is that they do not switch enough of the processor state. This we feel is due to the poor definition of the processor allocation function in general, and the fact that items he would like switched (internal timer, elapsed time clock) are items that are in the domain of the "normal" software scheduler and therefore not easily accessible to interrupt hardware due to the previously mentioned dichotomy.

4) Capability acquisition: if we generalize Dennis and van Horn's (10) concept of capabilities (as has been done by Fabry (8)) to mean a specification of any and all types of control for any and all types of resources, then we see that a great deal of the mechanism for transferring capabilities are included in the interrupt hardware. Most often this is implemented by having the interrupt automatically switch the processor from problem state to supervisor state, thereby allowing unregulated availability of privileged instructions to: a) execute I/O instructions (capability over all I/O resources); b) change interrupt masks (crude capability over the hardware scheduling algorithm); c) change storage protection status (capability over the memory resource). Often the processor allocation mechanism in the interrupt hardware also changes the storage protection keys (IBM 360) or the limits registers (CDC 7600) which is effectively switching capabilities for the core storage resource. Once again these procedures are totally unrelated to the "normal" software procedures which deal with exactly the same resources and their capabilities.

5) Task Interaction: several uses of the interrupt mechanism are really just task interactions of a special type. Interprocess interrupts (type 9) enable one task to activate another. Setting interrupt mask bits allows one task to prevent activation of another. Supervisor calls (type 2) are interactions between user and system tasks, with the interrupt mechanism used primarily for the purpose of transferring capabilities as previously mentioned. Interrupts caused by unusual conditions (type 5 and 6) and by debugging or monitoring conditions (type 7) can also be classified as types of interactions between cooperating tasks. In these cases capabilities are transferred between the tasks, but an additional factor is introduced - that of mutual exclusion between the tasks. For example, when the overflow condition is detected, not only must the recovery routine be activated and given capabilities over resources belonging to the routine causing the error, but the routine causing the error must be blocked (i.e., prevented from continuing) until the recovery routine takes corrective action and permits it to resume. Another very common example is found in I/O interrupts, where the task issuing the I/O request is often blocked until the "operation complete" interrupt from the device unblocks it. This type of block/unblock

interaction is found in many applications and has been discussed at length by Dijkstra (6). Current systems sometimes employ interrupts, sometimes other techniques for this same purpose, again largely due to the hardware-software scheduling dichotomy.

Suggestions

In view of the previous discussion we can make some observations on what needs to be done next. Namely, we need to design a total system that deals explicitly with the functional components as separate, independent, though interrelated entities. Some points to consider in this new design follow.

1) We need a method for allocating all resources that includes the instruction processor as just another resource, although an important one. This may involve specifying the allocation, deallocation, and interruption procedures as part of the resource itself, as has been suggested by Dahm (9), so that the designer of the operating system can specify exactly how each individual resource is allocated, deallocated, and what is saved, switched, restored, etc. on pre-emption. In this regard we suggest a definitional capability that enables the designer to specify exactly what the resource is and what is is not. Very important is the separation of the resource allocation function from the other functions, such as capability acquisition, which are now completely intertwined (type 2 interrupts, for example). Also very important is the separation of the resource, which is an autonomous mechanism, from status information about the current operation of the resource, which is really part of the task using the resource, not the resource itself. As Lampson points out, this implies that the state of the processor when a processor-switch occurs is saved not with the processor resource nor in a fixed memory area but rather as part of the task that was being processed at the time of the switch.

2) We need a model of operating systems that clearly distinguishes tasks, resources, and capabilities as the basic elements to be scheduled and allocated. This model needs precise definitional statements about what constitutes each element (and what does not) , and what are the permissible interactions between the basic elements. Once we have such a model, meaningless distinctions between interrupt servicing tasks and "normal" tasks, problem state and supervisor state, etc. will disappear. Dahm made a first attempt at delimiting the basic elements and assigning some structure to them, but he does not specify interactions to any extent. A machine (8) based on the capability list notion of Dennis and van Horn (10) should prove interesting insights on the relationship between tasks, resources, and the capabilities that interconnect them.

3) As part of a general operating system model we need a precise specification of a scheduling mechanism. This would involve specifying the structure and operations needed by a scheduler, perhaps by defining a "scheduler language" that would permit writing and implementing of scheduling algorithms in a coherent manner. Essentially we need to formulate a meta-system in which schedulers can be designed, studied, and implemented. As part of future computing systems there should be a unique functional unit, distinct from and operationally independent of all other resources being scheduled, called "the scheduler". This would be designed in the meta-system and would interface with the other components of the system in the manner prescribed in our general operating system

model. A great deal of thought needs to be given to such topics as: exactly what are "scheduling structures"; how do you specify them; what operations are needed on them; how do you write scheduling algorithms; what should be included and what excluded; how does all this fit into the total computing system; what type of interaction and feedbacks are desirable; how does one measure scheduling performance, efficiency, and cost?

4) The essential purpose of interrupts is detection of some happening, either within the computing system itself or in the environment. We mentioned that this detection is by means of a "trigger" signal generated in some mysterious fashion, the meaning of which is understood by some predefined convention. However "happenings" or "conditions being satisfied" are a convenient method of specifying interactions between distinct tasks, as has been discussed by Dijkstra (6) in his Sleeping Barber Algorithm, and Lampson(7) with his wake-up waiting switch. Hence both interrupt triggers and semaphore action (Dijkstra) are just instances of the more general need for specifying conditional interactions between elements in a computing system. Dahm (9) has attempted to fulfill this need by supplying the mechanism known as an "event". We feel however that his "event" is too restrictive because it does not provide a conditional detection mechanism to cause events. In other words, we need the ability to specify conditions and mechanisms for testing these conditions such that, when the test is satisfied, an associated interaction will occur. In this framework we would implement interrupts as follows: we have a functional unit called the detector which will accept "condition structures" from the rest of the system. These structures might specify single trigger sources, logical combinations of sources, time-dependent and/or order dependent sequences of sources such that if and when the condition is satisfied, the detector will "cause the event to happen" in Dahm's terminology. This would result in the detector submitting to the scheduler a request to activate all the tasks which have been waiting for the event to happen (i.e. the tasks which are on the event's queue in the Dahm model). More general would be the ability to specify, along with the event definition, the system action that is to be taken when the conditions are satisfied (i.e., activate all tasks on the event's queue, activate only the first task, etc.) Finally we would generalize the power of the detector so that it would include the ability to monitor other components of the system itself, both hardware and software. Thus the detector could cause an event whenever a specified task executes a specified instruction, accesses a specified variable, modifies a specified register, etc. This would include the functions of "address stopping" found in the hardware of some machines (IBM 1401, CDC 7600), monitoring of parameters and procedures found in higher level languages (B5500 Algol, Lisp), and to some extent hardware diagnostic ability of snooper computers (CDC 7600 machine check unit).

These ideas depend heavily on the existence of a good model for operating systems, something which is not now available but which we hope to present in the near future. However interrupt systems are such an important part of computer systems, that it is essential to have a clear understanding of their components and interactions with other elements in the system. We feel that a lack of such understanding has caused much of the confusion and poor design found on existing computers. We would greatly appreciate any feedback on the notions presented here, as that is one of the primary purposes for writing this paper.

References

(1) Brown, R.M., Fisherkeller, M.A., Gromme, A.W. and Levy, J.V., "The SLAC High-Energy Spectrometer Data Acquisition and Analysis System" Proc. IEEE, Vol 54 # 12 (Dec. 1966), pp. 1730-1734.

(2) Friedl, P.J., Sederholm, C.H., Lusebrink, T.R., and Jenny, C.J., "A Computer System for Automation of a Laboratory" F.J.C.C. 1968, Vol. 33, part II, pp 1051-1060

(3) Gelernter, H.L., Birnbaum. J., Mikelsons, M., et al.: "An Advanced Computer-based Nuclear Physics Data Acquisition System", Nuclear Instruments and Methods Vol. 54 (1967) pp 77-90

(4) Sachs, M.W., Bromley, D.A., Mikelsons, M., Summers, P.D., and Birnbaum, J.,: "A Multiprogrammed Operating System for Nuclear Physics Data Acquisition" IEEE Transactions on Nuclear Science Vol. NS-16 #1 (Feb 1969) pp 145-152

(5) Russell, R.D. "MIDAS: A Multilevel Interactive Data Acquisition System" IEEE Transactions on Nuclear Science, Vol. NS-16 #1 (Feb. 1969) pp. 122-126

(6) Dijkstra, E.W., Cooperating Sequential Processes (Sept. 1965) Technological University Eindhoven.

(7) Lampson, Butler W., "A Scheduling Philosophy for Multiprocessing Systems" C.A.C.M, Vol. 11 #5 (May 1968) pp. 347-360.

(8) Fabry, R.S., "Preliminary Description of a Supervisor for a Machine-Oriented around Capabilities", ICR Quarterly Report (University of Chicago) #18 (Aug. 1, 1968) pp. I B-1 - I B-97.

(9) Dahm, D.M., Gerbstadt, F.H., and Pacelli, M.M., "A System Organization for Resource Allocation", C.A.C.M. Vol. 10 #12 (Dec. 1967) pp. 772-779.

(10) Dennis, Jack B., and van Horn, Earl C., "Programming Semantics for Multiprogrammed Computations", C.A.C.M. Vol. 9 #3 (March 1966) pp. 143-155.