

29 Oct 72
Sunday

Processes for the CRMS System

Decision 1.

The lowest layer (logically speaking, that is) is a combination of two technological layers: the microprocessors, and a small set of trusted software processes.

The most basic object is the process ready list (one per microprocessor), with the following three routines (residing in each microprocessor):

1. Get ~~next~~ ready process, at highest priority label possible;
2. Add state block to ready list;
3. ~~Remove~~ state block from ready list.

These are called by various higher-level routines, such as Block, Wakeup, Stop, & START.

Need some way to call system. Seem to be 2 kinds of system calls: "light-weight" (i.e. just computing - system never blocks) and "heavy-weight" (i.e. I/O or etc, where system must block)

Can do heavy-weight system calls as traps. This will

- a) Stop trapping process
- b) Store his i.d. in trap-cell of this CPU
- c) Wake up trap-handler ($\equiv$ system).

Problem with doing lightweight calls this way is that caller gets stopped, hence gets recycled to end of (his level of) the ready-list.

29 Oct 72

(2)

High Level ↓

Would prefer for some processes to stay at head of list, as happens with, say, I/O process pre-emption.

Can use a new System-Wakeup op, which

- a) Leaves calling process ready
- b) Stores his id in trap-cell of this CPU
- c) Wakes up trap-handler

(Note: Must identity of trap-handler be built in to operation? Could user screw things up by waking up a friend, and setting trap cell?)

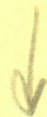
Best idea! : Traps never stop the offending process — this is always the responsibility of the Trap Handler. Then, trap handler ($\equiv$ system) can make informed decision about whether to stop caller.

Note: criterion for stopping APL processes may be ~~quite~~ rather different from those for SIMPLE processes, since $\exists$ APL processor which stands idle until restarted by trap handler.

29 Oct 72

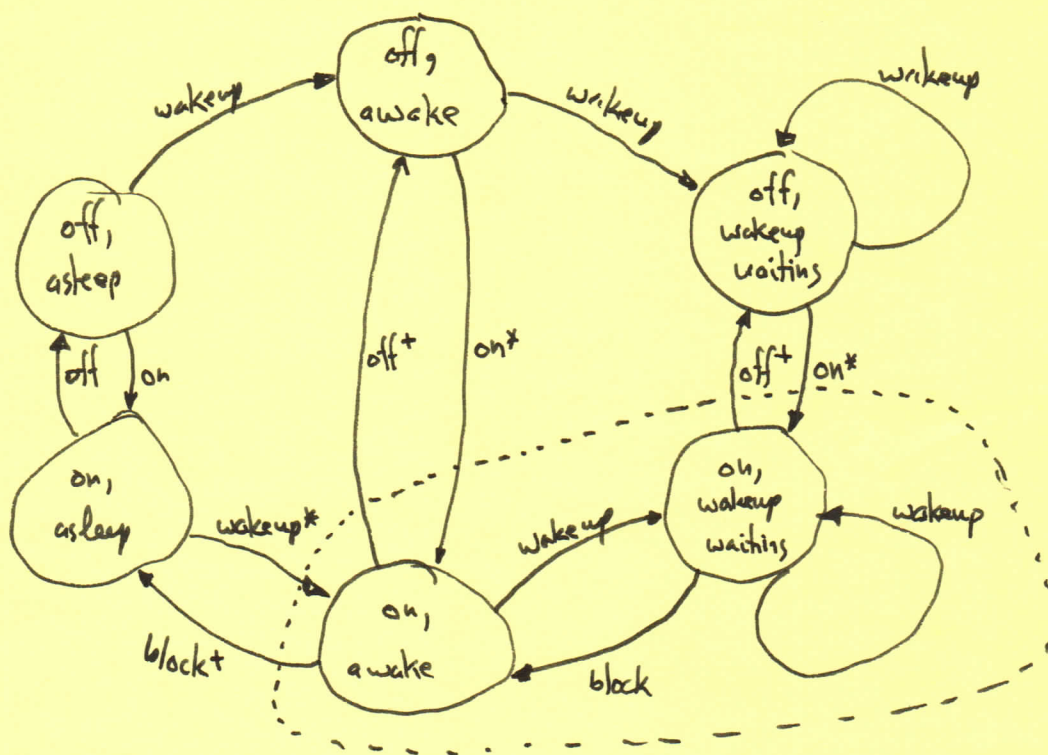
(3)

High Level



States of a process:

1. Turned on or off. This is "the master switch" for a process, and the power to set it is presumably reserved for the one process "in control of" the subject process.
2. Asleep or Awake or (awake with) wakeup waiting. These three states allow cooperating processes to signal each other.



* = must put process into ready list
+ = must remove process from ready list

It is an error to turn a process on (off) if it is already on (off).

29 Oct 72

④

CPU instructions interfacing the process machinery:

1. ~~Wakeup~~ sleep (i.e. block self)
2. Wakeup P
3. attention trap (puts T.ATTN $\rightarrow$ trap cell (CPU))
4. turn off P
5. turn on P

Note that P (process cop) is same for real process or I/O device. CPU-type field in PTE says 'APL' 'SIMPLE' or 'META4'. In the latter case, the format of the rest of the PTE is different, and contains (among other things?) a μ -routine ID (or something) which allows the wakeup operation to activate the right μ -process.

low level $\uparrow$

29 OCT 72

(5)

Processor Ready-List Interlock -

low
level

Use Split-cycle to give Test-and-Set routine in μ -code
TSL(W) W = word addr in core

Then, any code which looks at or modifies

Ready-List must TSL (Readylistlock)

(Note - watch out for lockout due to SIMPLE CPU priority)

Creation & Destruction of Processes (In Software?)

Processes are always created STOPPED.

Processes must be STOPPED before being
destroyed.

Software level

⑥

Software instr on processes

1. Create Process (CPU type ...)

father cap,
who pays?

2. Destroy Process (P); (needs opt. bit)

3. Access (read and/or write) various components of state block
of process P: (needs opt. bit)
closely held segments (procedure, data [caps if CPU type = SIM1])
stack top, etc.

4. Message send/receive

Software level



Error processing (non-attn TRAPS in general) are

sent cruising up the tree. In addition, one ~~error~~

~~trap~~ ^{class} ~~trap~~ is " ^{attention class} ~~trap~~ " which

the APL process uses to call for help from
Shepherd.

Software level



Xmit (D, n, Waitstate)

Waitstate = state in which to wait if receiver not waiting

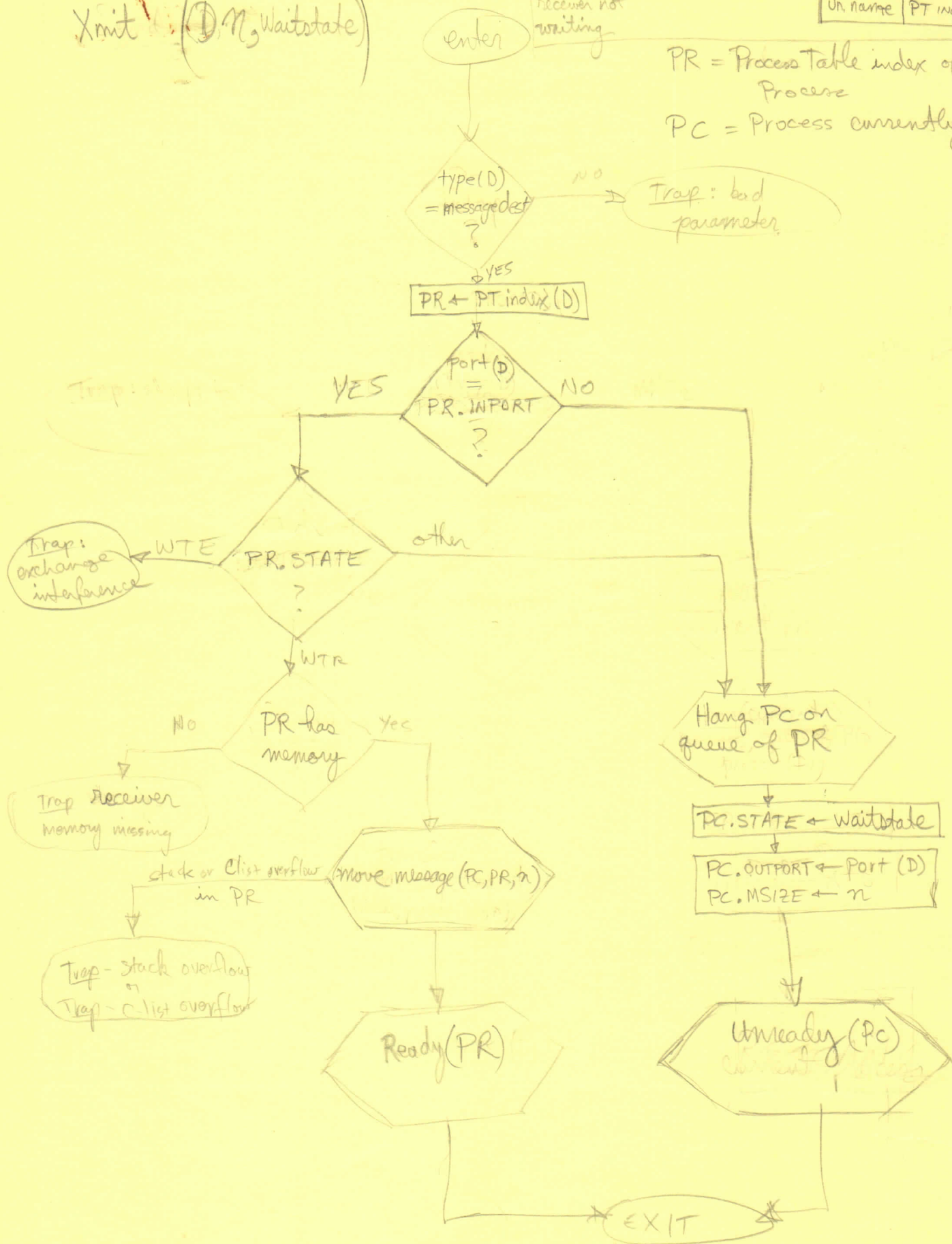
n = message size

D = cap

type	port
Un. name	PT index

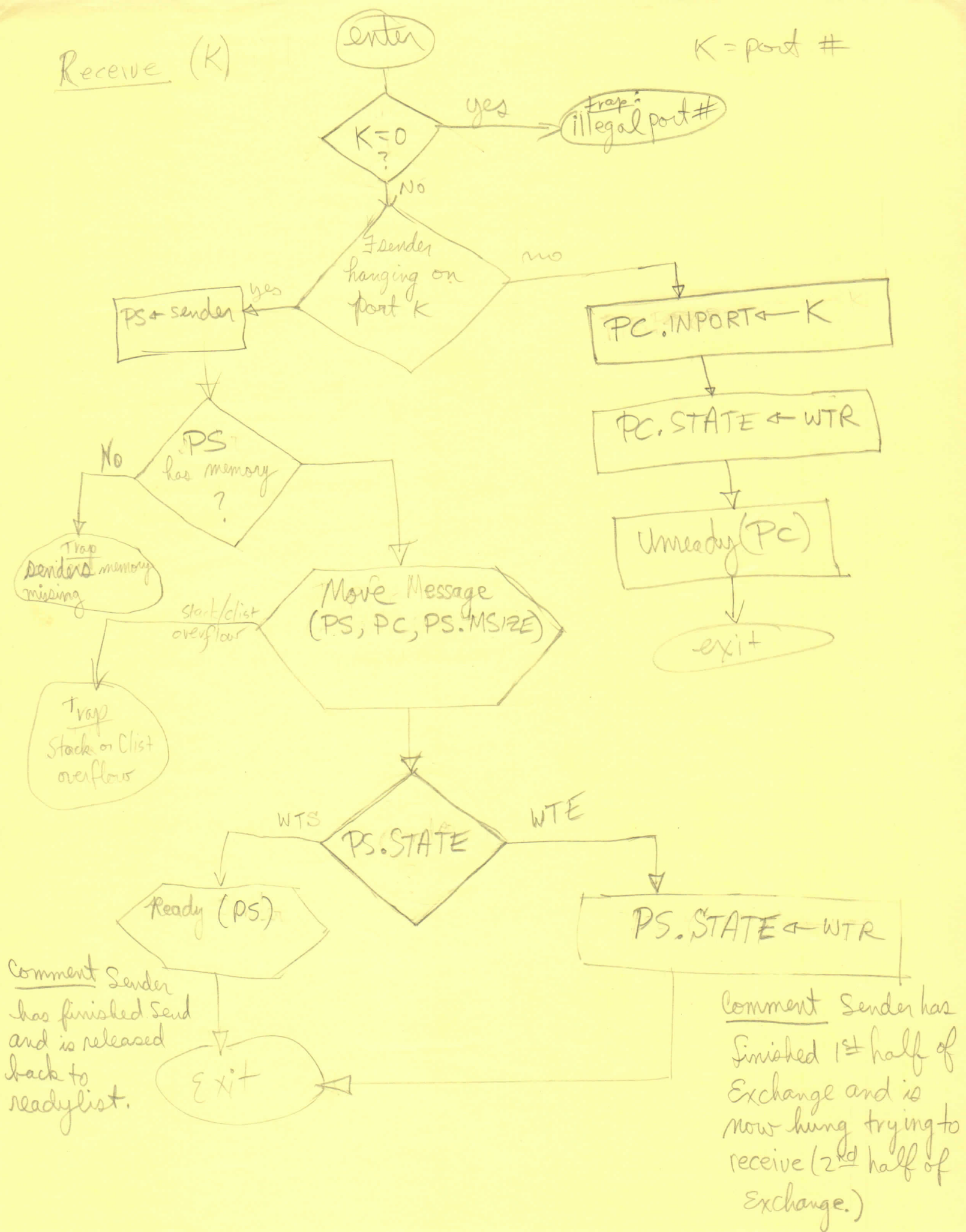
PR = Process Table index of receiving Process

PC = Process currently running

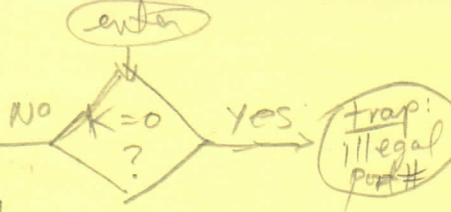


Receive (K)

$K = \text{port \#}$

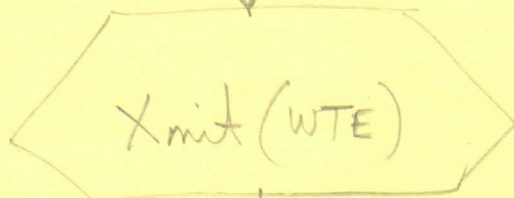


Exchange (D, n, K)

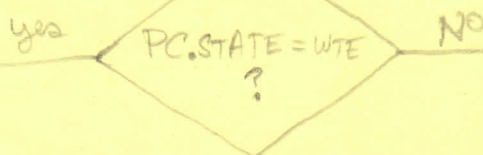


D = dest cap
 n = message size
 K = port # (for reply)

note that senders must be locked out in this region.



PC.INPUT ← K

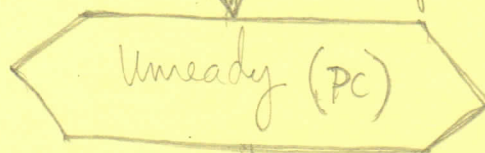


Comment

Process is hung trying to send (1st half of Exchange)

comment

Process has completed send and is hung trying to Receive (2nd half of Exchange)



EXIT

Inter u-processor signals

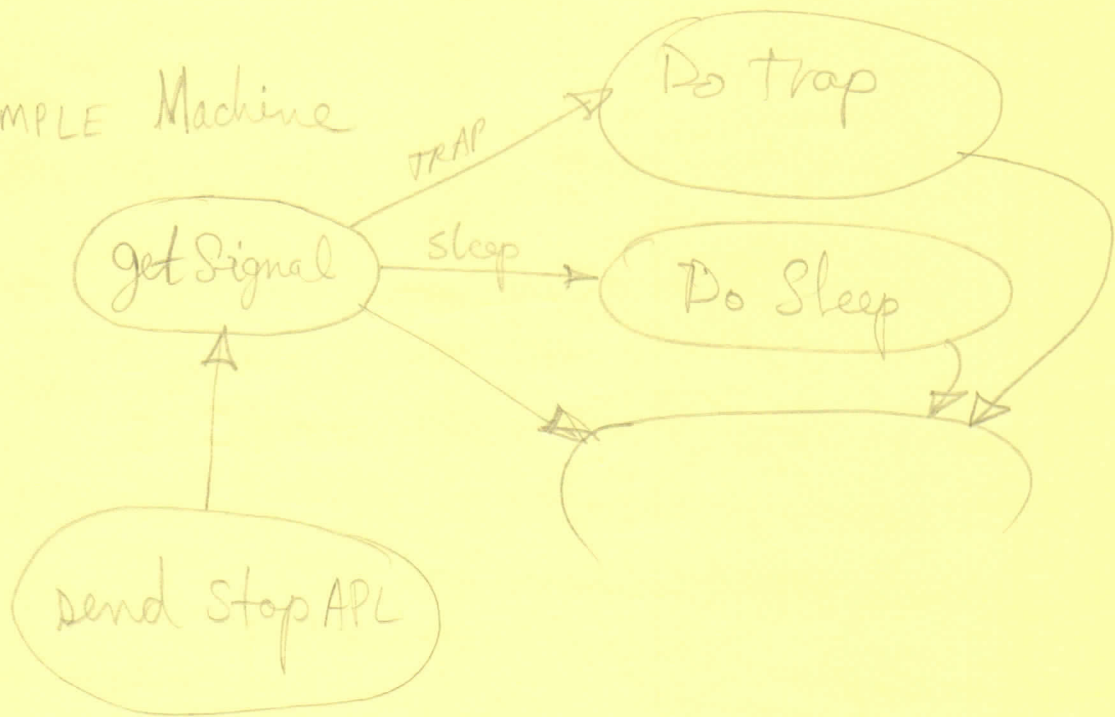
A. SIMPLE - TO - APL

1. Stop (dump current state)
2. Start (Process #N)

B. APL - TO - SIMPLE

1. Generate APL Trap. (trap datum already stored)
2. put process to sleep.
3. I am stopped.

SIMPLE Machine



HARDWARE INTERPROCESS COMMUNICATION

6/9/72

Summary

The hardware interprocess communication mechanism proposed is designed to fulfill two needs:

- a) To replace interrupts as the interface to I/O devices
- b) To function as a general monitor-call operation, thus providing a complete line of software services, including a more powerful and general (e.g. message channel) mechanism for interprocess communication.

At the hardware level, a process communicates with the rest of the world by sending and receiving signals to/from other processes (including hardware processes representing I/O devices.) A signal is a small packet of data and/or capabilities. Attached to each process is a signal buffer, through which all incoming or outgoing signals must pass. The signal buffer consists of a small fixed number of slots, each of which can hold a capability or a datum.

To output a signal, a process must do two things:

- a) load its signal buffer: (several CPU operations)
the data/capabilities comprising the signal are moved into the appropriate slots into the signal buffer, one by one.
- b) send the signal from its signal buffer: (one CPU operation)
the contents of the signal buffer are copied into the signal buffer of the receiving process.

Similarly, to input a signal, a process must:

- a) receive the signal in its signal buffer (one CPU operation)
- b) unload its signal buffer (several CPU operations)

Furthermore, each signal is of a certain class, and the sender/receiver must specify the class of the signal to be sent/received. In the case of the receiver, this is simply an arbitrary integer. In the case of the sender, the class is specified in the signal capability which must be presented as authorization to send the signal in question. A process can fabricate capabilities allowing other processes to send it signals.

Sending and Receiving

The primitive actions send and receive function in pairs. Whenever a send-receive pair is matched, the signal is transferred from the sender's buffer to the receiver's buffer, and both processes continue execution. When a send (receive) is done before its matching receive (send), the process is blocked until the matching action is performed.

Additional primitive actions provided are conditional-send and exchange. Conditional-send allows a process to send a signal to an untrusted process without risk of hanging up if the signal is never received. Conditional-send will send the signal if the matching receive has already been done. Otherwise, it will fail.

A complementary action to conditional-send is exchange. Exchange is simply an atomic send-receive sequence; this allows a signal to be sent and a reply signal to be received, even if the reply was sent with a conditional-send. A normal non-atomic send-receive sequence could fail in this case if the following occurred:

- (1) Process A sends "Request" signal to process B.
- (2) Process B attempts to return a "Response" signal with a conditional-send, which fails.
- (3) Process A attempts to receive the "Response" which has already been lost.

The exchange/conditional-send mechanism is intended for use as a "monitor-call" sequence. Note that the two signals involved provide a unified "input parameter" and "output parameter" mechanism.

Loading/Unloading the Signal Buffer

Each slot in the signal buffer must be individually loaded/unloaded from the memory or C-list of the process. This is preferred over an atomic load/unload-entire-buffer operation for several reasons:

- (a) Some simplification in the micro-code.
- (b) Elimination of any kind of "specification-list" of how the buffer should be loaded. (Such a list could sometimes be compiled, but would often need to be constructed at execution time.)
- (c) More flexibility at the receiving end:
 - (1) In placement of capabilities received.
 - (2) In signal formats (e.g., value of parameter 1 determines types of remaining parameters.).

Each slot in the signal buffer can hold a capability or an equivalent amount of data (presumably 2 words). A flag word in the buffer indicates which type of item appears in each slot. The primitive actions are

- (a) Load capability into signal buffer -
loads the indicated capability into the indicated slot and sets the flag for that slot to "capability."
- (b) Load data into signal buffer -
loads two words of data into the indicated slot and sets the flag for that slot to "data."
- (c) Unload capability from signal buffer -
stores the capability found in the indicated slot at the indicated position in the C-list, or generates an error if slot contained data.
- (d) Unload data from signal buffer -
stores the data found in the indicated slot at the indicated position in memory, or generates an error if slot contained a capability.

no checking is done by the hardware to validate the format of the signals (e.g., 1 capability, 2 data, 1 more capability). Such checking (analogous to argument type checking in procedure calls) is left to the software of the processes involved. Perhaps some kind of standard library routine should be provided to perform type-checked signalling.

Traps

Some mechanism will be provided to allow a process to deal with its own errors. Whenever a process is unable or unwilling to handle an error it has made, a trap results, which causes the CPU to scan up a chain of "father-pointer" capabilities. Each process state-block contains such a pointer; hence, a tree-structured hierarchy is formed. When the CPU scan locates a responsible process, the offending process is forced to do an exchange with the responsible process. Note that while this allows restarting of a hardware operation which caused an error, it precludes restarting a monitor call which caused an error, without reloading the signal-buffer, since the trap-exchange uses the signal buffer.

"Return link" capability -

As software convention, "caller" should send a destination capability (normally, a port of its own) where response is to be returned. The port# is also specified as a parameter to the RECEIVE half of the SEND. Agreement between these two is crucial, and will be the responsibility of the user prog and/or the compiler.

One place where all this is rather awkward is in external-trap messages. The return link cap is clearly needed here, which means that the μ -code must fabricate it

So Send (C, n)

Receive (p)

Exchange (C, n, p)

Traps

① What port is specified in the implicit exchange when a trap is handled externally? #0

② What would happen if trap did a send-stop instead of a send-Receive (exchange)?

The main differences would be:

- a) Trap handler would have harder time returning a result to the trapped process. (But, the kind of result he would return would only be of value if process wanted a message.* Trap during an explicit receive is only case of this. Stop would work here too)
- b) No port on the trapped process would need to be used. Only trap handler. (This is a plus.) How is this port specified?

* i.e. format(array) would be wrong otherwise.

Simple language: constructs for messages

1) Send - pretty straightforward

"Send (C, x, y, z)"

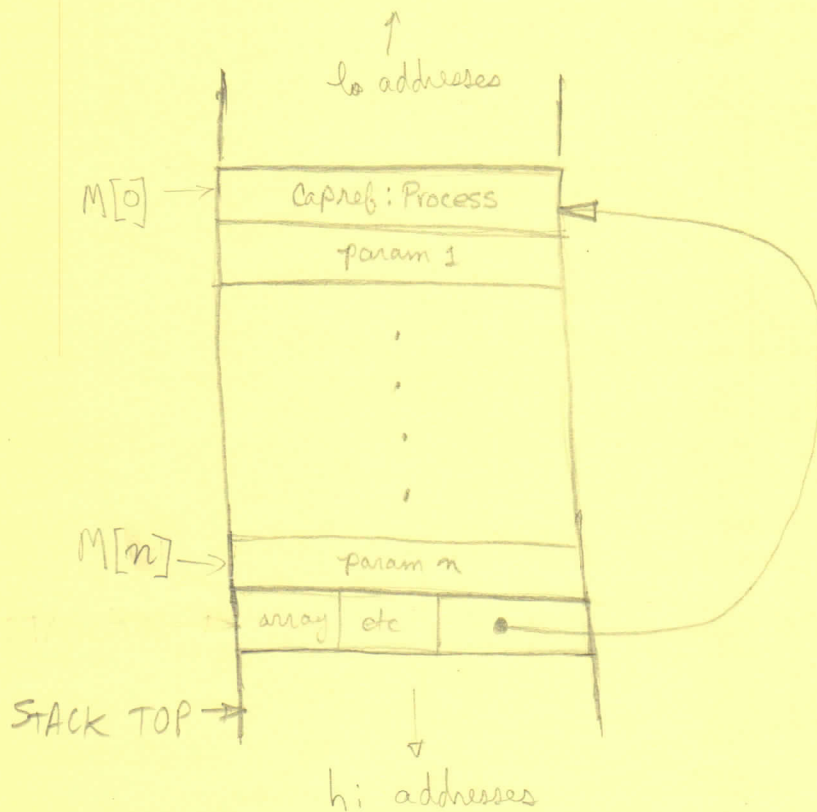
compiles: push z
push y
push x
send C, 3

2) Receive - more subtle problems

a) What to do with sender's i.d:
could make Receive function return it
as result:

"Sender ← Receive(N)" ^{port number}

b) Rest of message is harder - want
to munch it apart dynamically,
hence can't use arg list of Receive function.
How about creating an array on
receiver's stack and returning
descriptor for it as the value.
This is graceful way of sending n
to the receiver



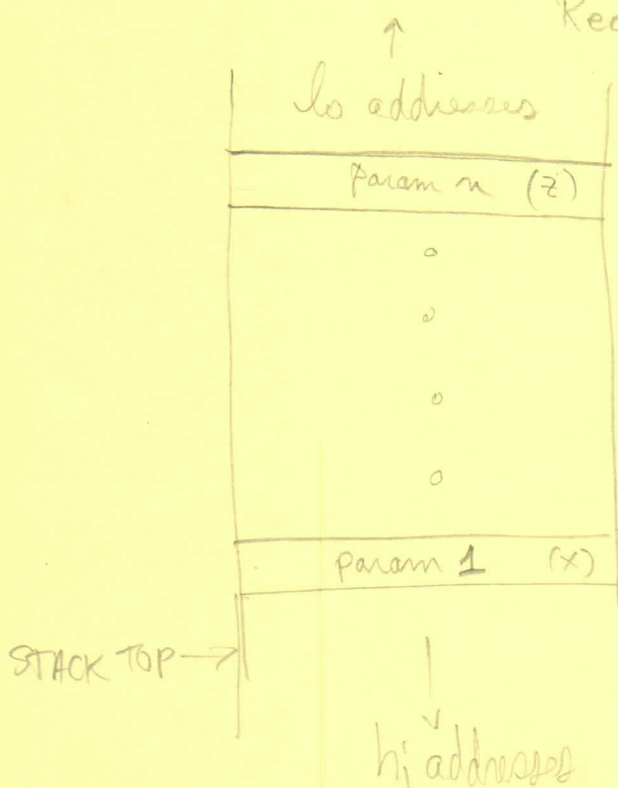
Source:

message ← Receive (Port)

code:

Receive Port
Pop
Store Message

Message on Receiver's stack after execution of
Receive operator.



Source:

SEND (Process, x, y, z)

code:

push z
push y
push x
send Process, 3

Extendible IPC

1. Send-Receive function in pairs. 1st half of each pair waits for second.
2. Send-Receive transfer directly between stacks.
3. Micro-code works only if both address-spaces are swapped-in and receiver has space in his stack.

Otherwise, trap to software.

4. To do buffered messages, $\neq$ buffered-Send operation which also traps if receiver not waiting. Message manager can then create little buffer process to wait in the queue, and original process ^{can} proceed. (Must be careful about core requirements of buffer processes. Either share data segments or swap them out. Former is unpleasant, but latter is probably too slow.) Alternative would be to put a pseudo-process in target processes queue, which would cause trap to message manager when a receive bumped into it. Ah ha! Pseudo process is just a regular process whose address space does not exist - hence attempt to receive from it causes trap to Message Manager as desired.

Arguments for wired-down buffer

- 1) Simpler to find (stupid argument)
- 2) Gives place to put message if sender or receiver has gotten swapped out.

(Not important - if either one is swapped out, the real & compute time to get him in justifies doing the message passing in software)

- 3) Allows sender to go on if receiver is waiting but is swapped out. (again - do it in software - use a buffer process)

Want IPC to function as a general "system-call"

Desirable Properties for IPC:

1. High speed $\hat{=}$ low overhead
2. Ability to pass general parameters
 - a) any number
 - b) any mixture of types
 - c) both input & output params
3. Parameter checking
 - a) number
 - b) type
4. Flexible activation properties
 - a) both serial and parallel cases included
 - b) recursive activation & backtracking (suggest stack)
5. Capability protection on activation rights, classes
6. Ability to represent I/O device signals.

2nd level message system generalizes synchronization properties. Also improves messages

- a) delivered direct to core/c-list
- b) any length OK

Minimum required properties of low-level IPC:

- a) must allow activation of message handler process.
- b) must represent "trap" or "error" processing (i.e. what to do to guy found by scan up tree)

7 June 72

①

Mixing ^{capabilities} ~~events~~ & data in events

process can ^{read/write} ~~write~~ caps and data from memory / c-list to / from buffer

Question: do caps or their indices go into buffer? Caps are awfully big.

1. Load buffer

- a) from C-list - sets "capability" bit
- b) from memory - sets "data" bit

2. Store buffer

- a) into - C-list - requires "capability" bit
- b) into memory - ignores type bit (i.e. can display a cap parameter)
(should display it)

buffer is really a parameter list! (All hardware ops with params could go thru it.) In particular, send event can take param #0 as cap for receiver (process + port). Rest of parameters are passed to receiver's parameter buffer. Doesn't seem to be any good way to have system do parameter checking - this would require gates. Can't have gates as separate objects in core. Even gates attached to processes would be a pain. Swapping them out would be difficult or impossible. Could provide library routine to check parameters against a specification list in a standard format (kind of cruddy: ~~no~~ no protection against other user code. Prob. not too important.) In any case, param specs should be inside the receiver's address space

must integrate I/O controller signals into event mechanism. Simplest idea is just a 1-parameter (or even no-parameter) event and all other signals via shared memory. ~~Other mechanisms~~ ^{Signals to controller} should look just like events to a process - funny cap type can signal hardware to run the controller micro-code. Question arises: what if input signal ~~arrives~~ from controller and responsible process is not waiting for it. Should probably cause error. Certainly doesn't make much sense for controller to wait? (like stacked interrupt? doesn't really handle same case [i.e. higher priority int. routine in control]).

Another worry: this "system call" mechanism provides no stack. What were functions of subprocess call stack. How bad is it to do same stuff without stack

Possible Rationale for dual (2-level) IPC facility.

When two ^{protected programs} (i.e. in our system, processes) interact, these interactions tend to fall into one of two classes.

1) Machine-extension - ~~process~~ process A is serving to extend the virtual machine on which A ~~itself~~ runs, to present a "higher-level" virtual machine to process B. In general, the extending functions of process A fall into two categories

a) resource controls

b) service actions

The former operate independently of ~~process B~~ and not necessarily to B's direct benefit. The latter are invoked on B's request ^(explicit call or implicit trap) as operations of B's extended machine. In terms of protection, A is ~~certainly~~ certainly protected from B, while B ~~is~~ is often not protected from A, hence A is generally termed B's "superior"

2. Co-operating programs - process A and process B are cooperating (in a mutually protected way), and neither ~~sees~~ sees the other as a "machine-extension". Neither A nor B is necessarily superior to the other; ~~most~~ generally, their relationship can be termed "mutual suspicion".

Might be able to claim that micro-coded lo-level IPC ~~does~~ does #1 and hi-level software IPC (message channel ~~manager~~) does #2

Given want to send capabilities and data thru the signal buffer:

Question: Should capabilities be copied into the buffer

Or should only ~~the~~ pointers to capabilities appear there?

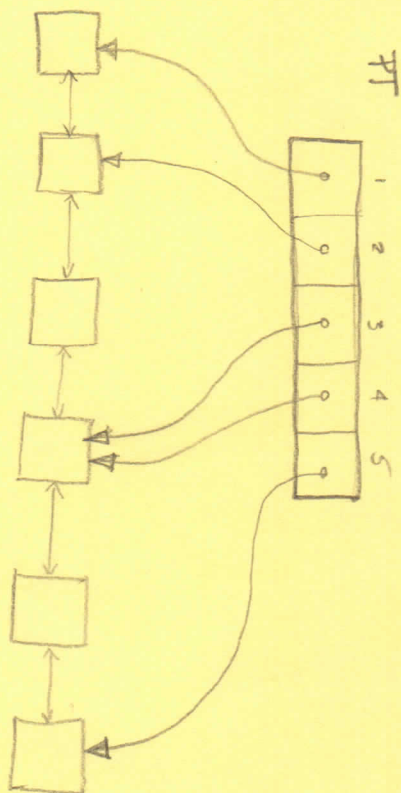
Two corresponding schemes:

- ✓ {
1. Copy capabilities
 - a. Sender loads each cap/datum into buffer
 - b. Sender does send } system copies buffer verbatim
 - c. Receiver does receive }
 - d. Receiver unloads buffer into core/C-list
 2. Copy pointers to caps ~~xxxxxxxxxx~~
 - a. Sender loads each datum/cap pointer into buffer
 - b. Sender does send } system copies data across verbatim
 - c. Receiver ~~xxxxxx~~ does receive } copies cap from designated slots into bottom of receiver's C-list (and loads pointer to each cap into approp. slot in buffer?)
 - d. Receiver unloads data into core. (may or may not need cap pointers)

Advantages of 1 over 2:

- 1) uniform treatment of caps & data. No fooling around w/pointers
- 2) At time of transfer, all data is right in buffer. No problems with complicated addressing into

Assume 1 pointer per priority class (to head of sublist)

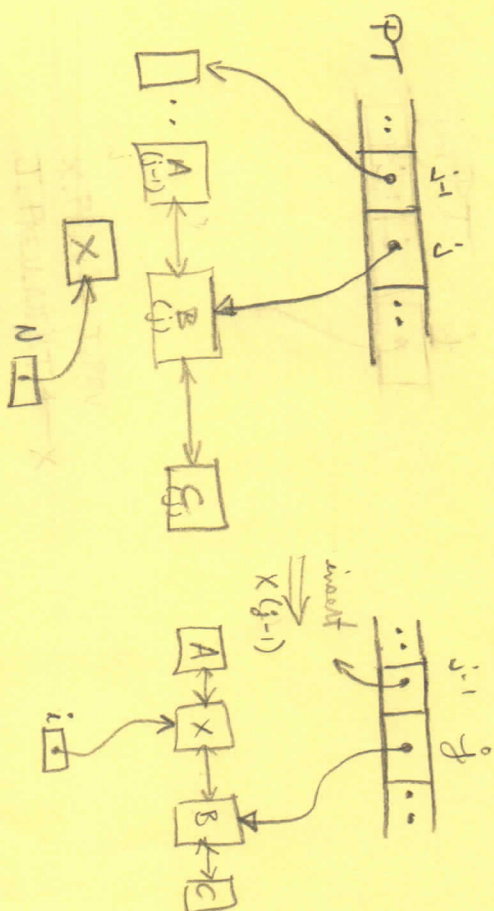


for given pointer (priority class), can have

- a) no processes
- b) 1 process
- c) several processes

Possible operations are

- a) insert in queue before pointer j (tail of $j-1$)
- b) insert in queue after pointer j (head of j)
- c) remove



insert at tail of Q_{j-1}

$N.NEXT \leftarrow PT[j]$
 $N.PREV \leftarrow PT[j].PREV$
 $N.PREV.NEXT \leftarrow N.NEXT.PREV \leftarrow N$
 $PT[j-1] \leftarrow N$ IF $PT[j-1] = PT[j]$

insert at head of Q_j

$PT[j-1] \leftarrow N$
 $PT[j] \leftarrow N$
 $N.NEXT \leftarrow PT[j].NEXT$
 $N.PREV \leftarrow PT[j].PREV$
 $N.NEXT.PREV \leftarrow N$
 $N.PREV.NEXT \leftarrow N$

Send signal

Parameter buffer (before)

P0 = Cap: Receiver

$$\begin{matrix} P1 \\ \vdots \\ P_m \end{matrix} = \begin{matrix} \{ \text{Cap} \} \\ \{ \text{Data} \} \end{matrix} : \begin{matrix} \text{Signal} \\ \text{Data} \end{matrix}$$
Send signal or fail

same before

after
 P0 = ^{Data:} result (0 $\Rightarrow$ sent, 1 $\Rightarrow$ failed)
Receive signalbefore

P0 = Data: signal class

$$\begin{matrix} P1 \\ \vdots \\ P_m \end{matrix} = \text{undefined}$$
P_mafter

P0 = Data:

Capabilities: 2 words

segment
cap { unique name (32 bits)
type (6-10 bits)
options (~20 bits)

24	8
options	type
Unique name	

Other types allow 2nd word to be set w/ authorization
 Will 32 bits always be enough?

Hardware IPC proposal:

Attached to each process is an event buffer which can hold one event (including associated control info). Algorithms below concern interaction between two processes' event buffers (How a process looks at his own event buffer is ignored - assume he can read/write it).

The Basic actions ~~are~~ send and receive ~~which~~ function in pairs. Whenever a send and receive pair is matched, the ^{signal} ~~message~~ is transferred from then sender's ~~event~~ buffer to the receiver's ~~event~~ buffer and both processes continue ^{execution} ~~work~~. Whenever a send ~~or~~ (receive) is done before its matching receive (send), the process is blocked until the matching action occurs.

Each ^{signal} ~~message~~ is of a certain class, and a sender/receiver must specify the class of the ^{signal} ~~message~~ to be sent/received. In the case of the receiver, this is simply a parameter (i.e. arbitrary integer). In the case of the sender, the class is contained in the ^{signal} ~~message~~ capability which ~~must~~ ^{be} presented as authorization to send the ^{signal} ~~message~~ in question. ~~Each process must have a capability to send each class of signal~~ A ^{com} process fabricates ^{signal} capabilities allowing other processes to send it ^{signals} ~~and each process must have a capability to receive each class of signal~~

conditional send An additional action provided is Exchange, which is

cont. an atomic send-receive sequence performed by a single CPU operation. The receiver of an event ~~message~~ can tell whether the event ~~message~~ was transmitted by a send or by an exchange. In the latter case, the receiver knows that the sender is already waiting for a reply, hence there is no danger in sending (of same event class).

that reply (i.e. hanging up if reply is never accepted)

This mechanism is rather crude, but should be adequate for use as a "system call" facility, allowing a more sophisticated ("message channel" or etc) IPC system to be implemented in software.

Interaction ~~between~~ this mechanism and self-caught traps (e.g. process own <sup>[overflows
user "quits"
etc]</sup>) is not clear.

Probably, each trap should check a mask (catch-own-traps_i). If bit is on, then an implicit function call occurs. If ~~this~~ bit is off, then scan up father tree is done to find an interested process and a forced-exchange is done by the trapped process. A further possibility would be trap-suppression in the offending process, which might want to completely ignore the trap, or might test for it explicitly ("condition code" or etc.). (This latter is only useful for system calls and, possibly, for ^{some} function calls ^(like procedures) - certainly not for infix operators). Possible snarl if trap occurs during a system call (i.e. p-counterpoints at the exch rather than after it). Then, the process is doing 2 exchanges at once! This

Receive (rev-class)

```

begin got-event  $\leftarrow$  false;
  for sender  $\leftarrow$  self.sender-queue, sender.senderqueue while sender  $\neq$  null do
    if sender.event.class = rev-class then
      begin
        dequeue (sender);
        self.exchange  $\leftarrow$  sender.exchange
        self.buffer  $\leftarrow$  sender.buffer;
        wakeup (sender);
        got-event  $\leftarrow$  true
      end;
    if  $\neg$  got-event then block
  end;

```

~~Send (send-cap)~~

```


  begin self.exchange  $\leftarrow$  false;
    emit (send-cap);
  end;


```

Exchange (send-cap) ~~atomic~~

```

begin (atomic)
  self.exchange  $\leftarrow$  false;
  send emit (send-cap);
  Receive (send-cap.event-class)
end;

```


Send

~~Wait~~ (send_cap)begin

R ← send_cap.receiver;

C ← send_cap.event_class;

if ~~R~~ ^{receiving} R. ~~ready~~ ∧ C = R.event_class thenbegin

R.event_buffer ← self.event_buffer;

~~R.event_buffer ← self.event_buffer;~~

wakeup(R)

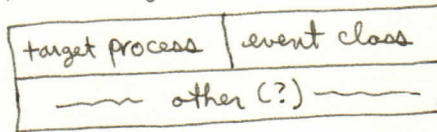
endelsebegin

enqueue (self, R.sender_queue);

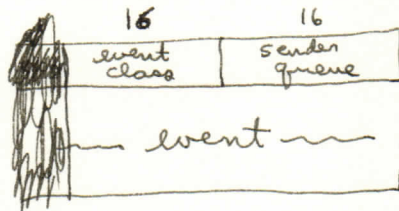
block:

endend;

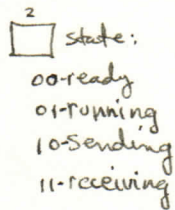
send capability:



event buffer:



plus, in state block:



Note - Want both

- 1) Conditional send (only if receiver waiting)
i.e. for monitor responses
- 2) Exchange (for user $\rightarrow$ system calls)
to guarantee user gets hung in receive soon enough.

IPC Questions

(1) given both exchange and send-or-fail
can process now pass out arbitrary caps
to send its events?

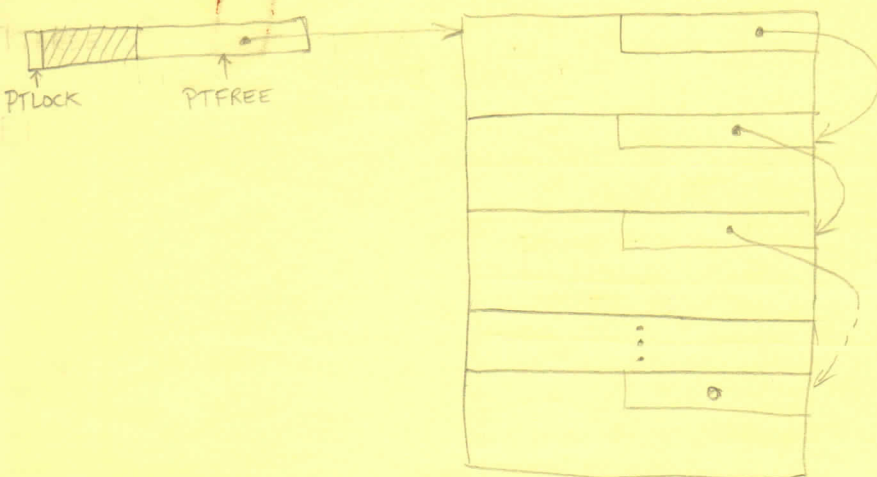
(2) Should exchange demand both events
be of same class? This seems
restrictive. What are alternatives

(pass return ~~class~~ as part of event?

store in separate word in receiver's state?)

Both class and process could be implicit, as
in Hansen's paper.

Process Table



Process Table Entry (SIMPLE Processor)

UNIQUE NAME	1 wd
FATHER POINTER	1 wd <i>for software only</i>
CPU TYPE	1 wd
PRIORITY	
QUANTUM	< 1 wd
STATE (READY, WAIT-TO-SEND, etc)	
REGISTERS	4 wds
TOP-OF-STACK PTR	1 wd
MEMORY: Seg caps (wd 2 only) for proc data esp segments	3 wds
LINK WORDS	
READY-LIST	1 wd
SEND QUEUE	1 wd
MESSAGE SIZE	1 wd
OUTPORT	
INPORT	
~ 16 words	

① down, across pointers?

Interlocks

1. Process Table (incl both readylists)

to avoid cases like 2 CPU's try
to add a process to same readylist
at same time.

2. Need lock to prevent both CPU's from sending a message to the same process (i.e both generate traps to mem manager at same time) Is process-table lock too heavy to use here?

Microcode Routines

Ready (P) Locks Process Table

Inserts process P at end of appropriate sublist of the ready-list. ^{Unlocks PT} Compares priority of P with current priority and calls Rescan if P is higher

Unready (P)

Removes process P from readylist. Checks whether P was running on its CPU and calls rescan of that CPU if so.

Claim: only connection between CPUs is a pre-empt wire, used by one CPU when it makes a hi priority process ready on the other CPU. This must go in both directions. (e.g. SIMPLE $\rightarrow$ APL due to scheduler decisions)
APL $\rightarrow$ SIMPLE due to traps)

Wakeup (P)

1. Error if P ready
2. Check "trap-on-wakeup" flag in P's state
if on, goto 5
3. Insert P in readylist at appropriate spot.
4. exit
5. Generate "wakeup" trap: signal = C: process P
D: trap.wakeup

Sleep (P) called by Block,

1. Error if P not ready (or running)
2. Remove P from ready-list
3. if P not running then exit
4. Initiate rescan by CPU (P) (i.e. 3 cases:
 - 1) SIMPLE process blocks self
 - 2) APL process blocks self
 - 3) SIMPLE process stops APL process

Rescan()

1. Store current CPU state in stateblock of $\hat{P}$
2. Scan ready-list for 1st non-empty queue
3. Load CPU state from 1st stateblock in that queue

Generate Trap(i)

1. Push n items of trap information onto the stack

2. if Trap-mask[i] = 0 goto 5

process catches
own trap

{ 3. push descriptor in Trap-function[i] onto stack

4. goto Function Call

5. $P \leftarrow \hat{P}$

6. $P \leftarrow \text{father}(P)$

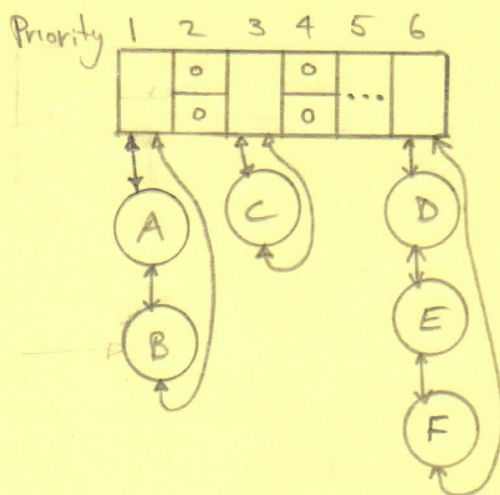
7. if Trap-mask[i] = 0, goto 6

8. goto Send(P, n)

Move Message (Sender, Receiver, Size)

Structure of Ready-List

Lampson proposes a table of list heads of 2-way circular lists, with heads of empty lists set to $\boxed{00}$. For Example:



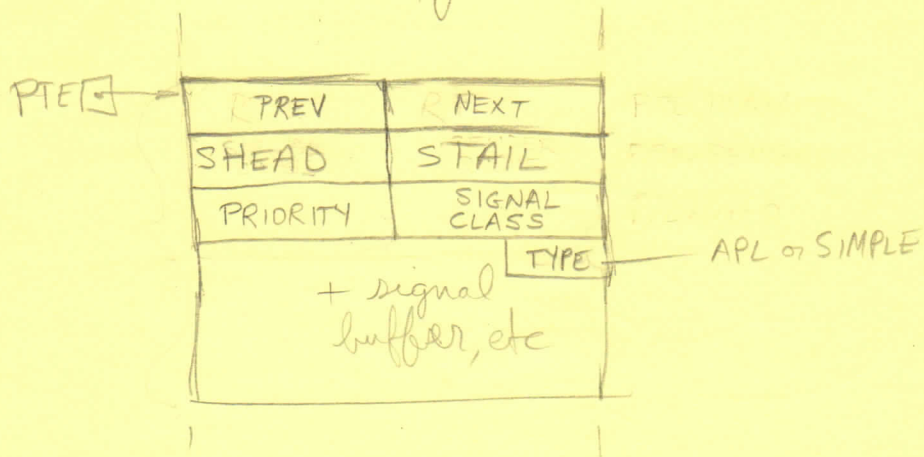
Note: better to set empty to

X: $\boxed{X \ X}$

To find 1st ready process with this scheme sometimes entails scanning down the table to find the 1st non-empty queue. On the other hand, insertion & deletion of processes is very easy.

A modified scheme might avoid scanning the table by putting all processes into a single ready list, so that 1st item in list was always pointed to by the head. To avoid scanning the list during insertions (i.e. wakeups), extra pointers into the list could mark the natural insertion points.

Process Table entry



CPU state block



w/ multiple CPUs, running scheduler, need

- Interlocks (test & set in memory)
- Notifications of pre-emption (process added to ready list of other CPU)
goes thru I/O attention-bit mechanism
- Quantum overflow on APL machine - what if APL machine has runned part way down ready-list when timer strikes on simple machine?

12 June 72

FUNCTION READY (PROCX);

```
PTE ← @ PROCESSTABLE [PROCX];  
CPU ← PTE.TYPE;  
LOCK (READYLIST [CPU]);  
HEAD ← @READYLIST [CPU, PTE.PRIORITY];  
PTE.NEXT ← HEAD;  
PTE.PREV ← HEAD.PREV;  
PTE.NEXT.PREV ← PTE.PREV.NEXT ← PTE;  
UNLOCK (READYLIST [CPU]);  
RETURN;  
ENDFUNC
```

also

check status ≠ ready

FUNCTION STOP (PROCX);

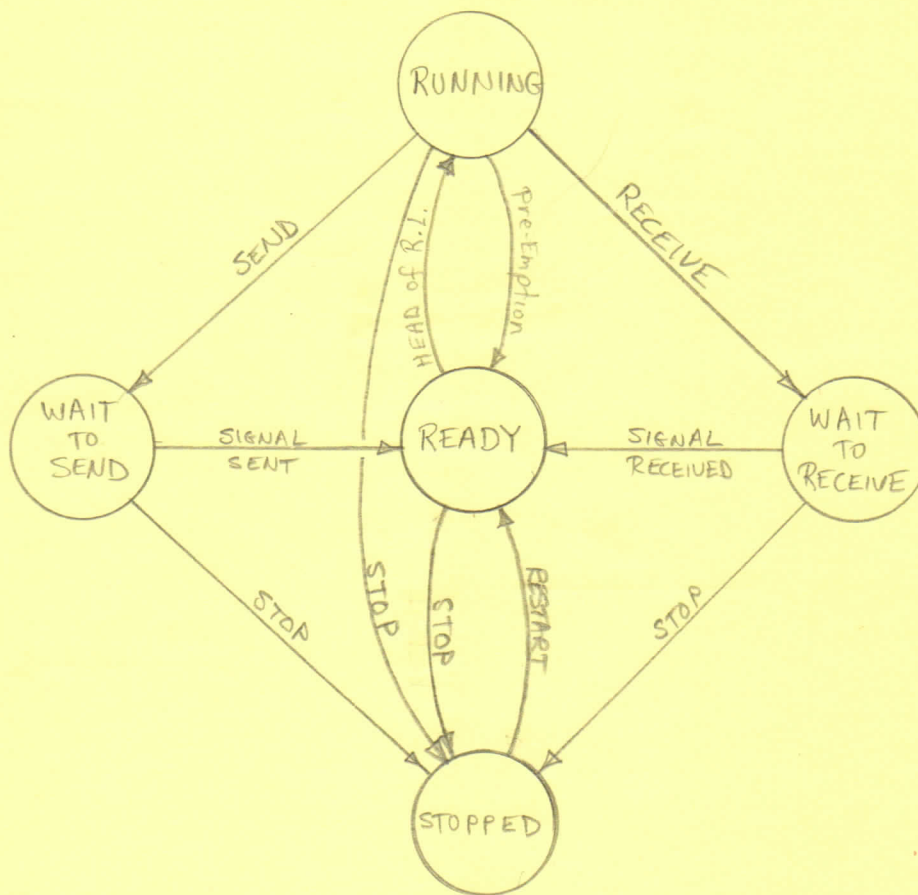
```
PTE ← @ PROCESSTABLE [PROCX];  
CPU ← PTE.TYPE;  
HEAD ← @READYLIST [PTE.PRIORITY];  
PTE.NEXT.PREV ← PTE.PREV;  
PTE.PREV.NEXT ← PTE.NEXT;  
RETURN;  
ENDFUNC;
```

also

check status = ready
lock readylist

FUNCTION BLOCK ();

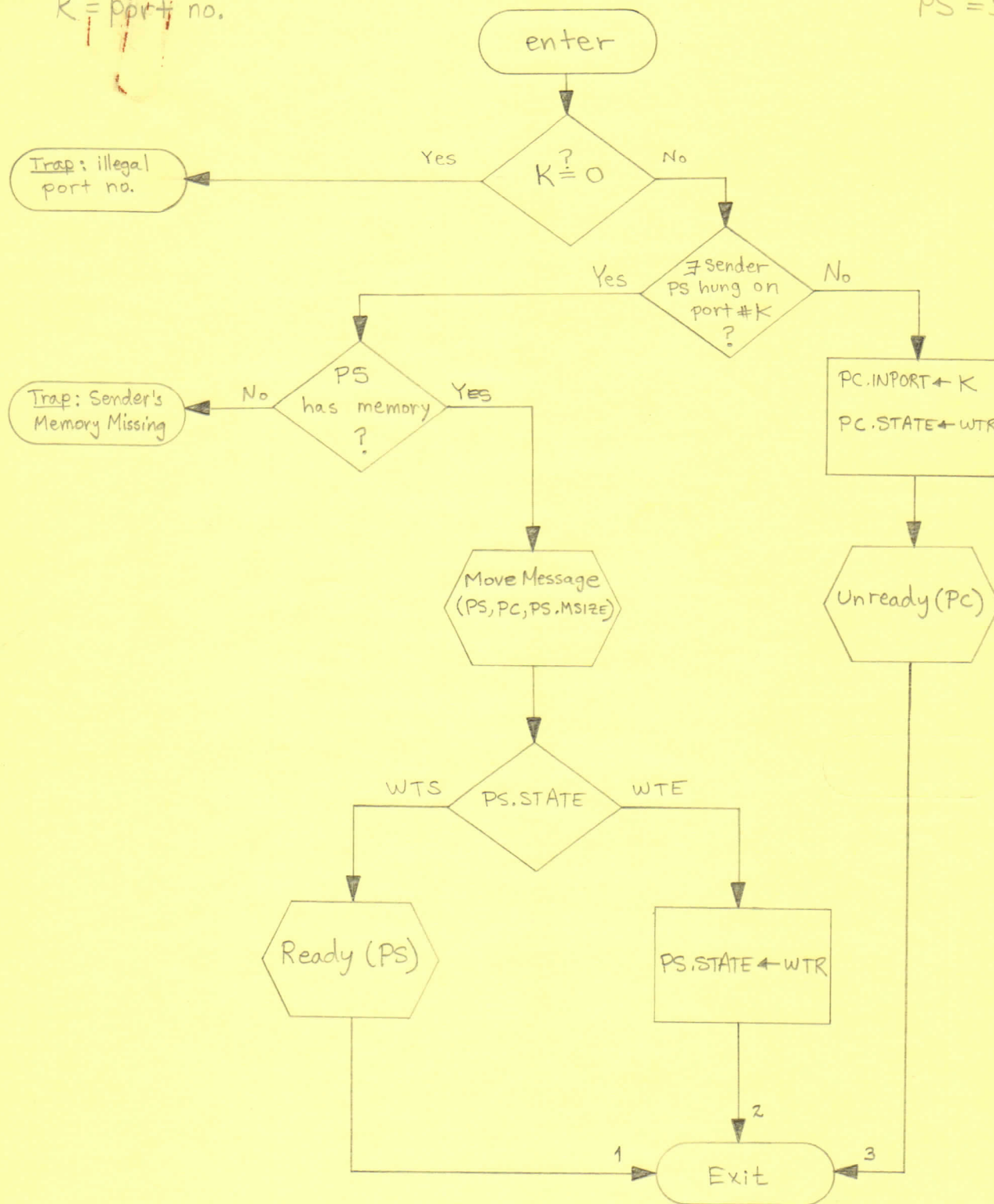
27 June 72



State Transitions of a Process

Receiver (K)
K = port no.

PC = current process
PS = sending process



Note Conditions on exit:

- 1) Current process and sender both ready
- 2) Current process waiting; sender hung in part 2 of Exchange
- 3) Current process hung waiting for message.

Exchange (D, n, K)

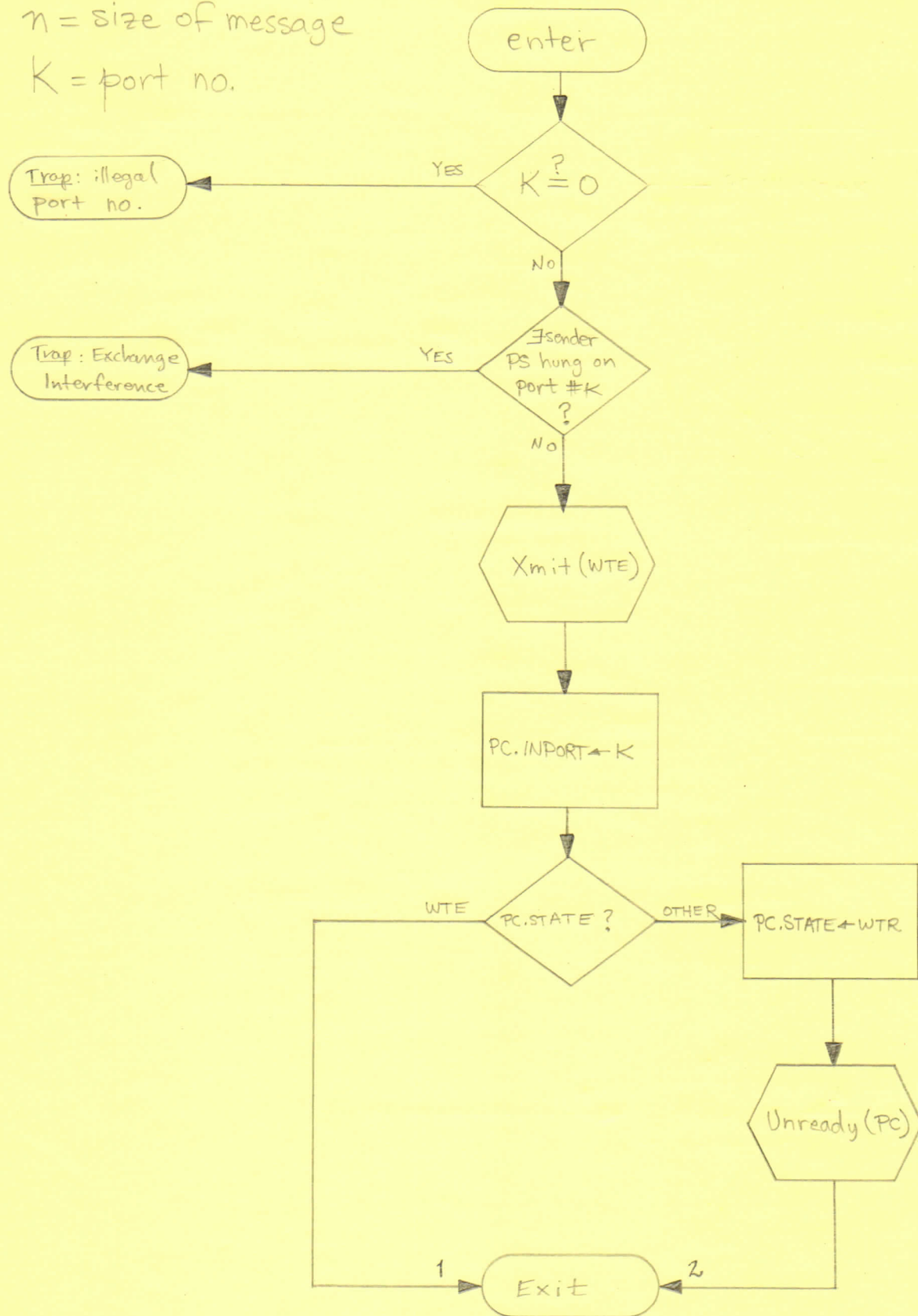
D = destination (cap)

n = size of message

K = port no.

PC = current process

PS = sending process



Note Conditions on exit:

- 1) Current process hung in part 1 of Exchange (Send)
- 2) Current process hung in part 2 of Exchange (Receive)

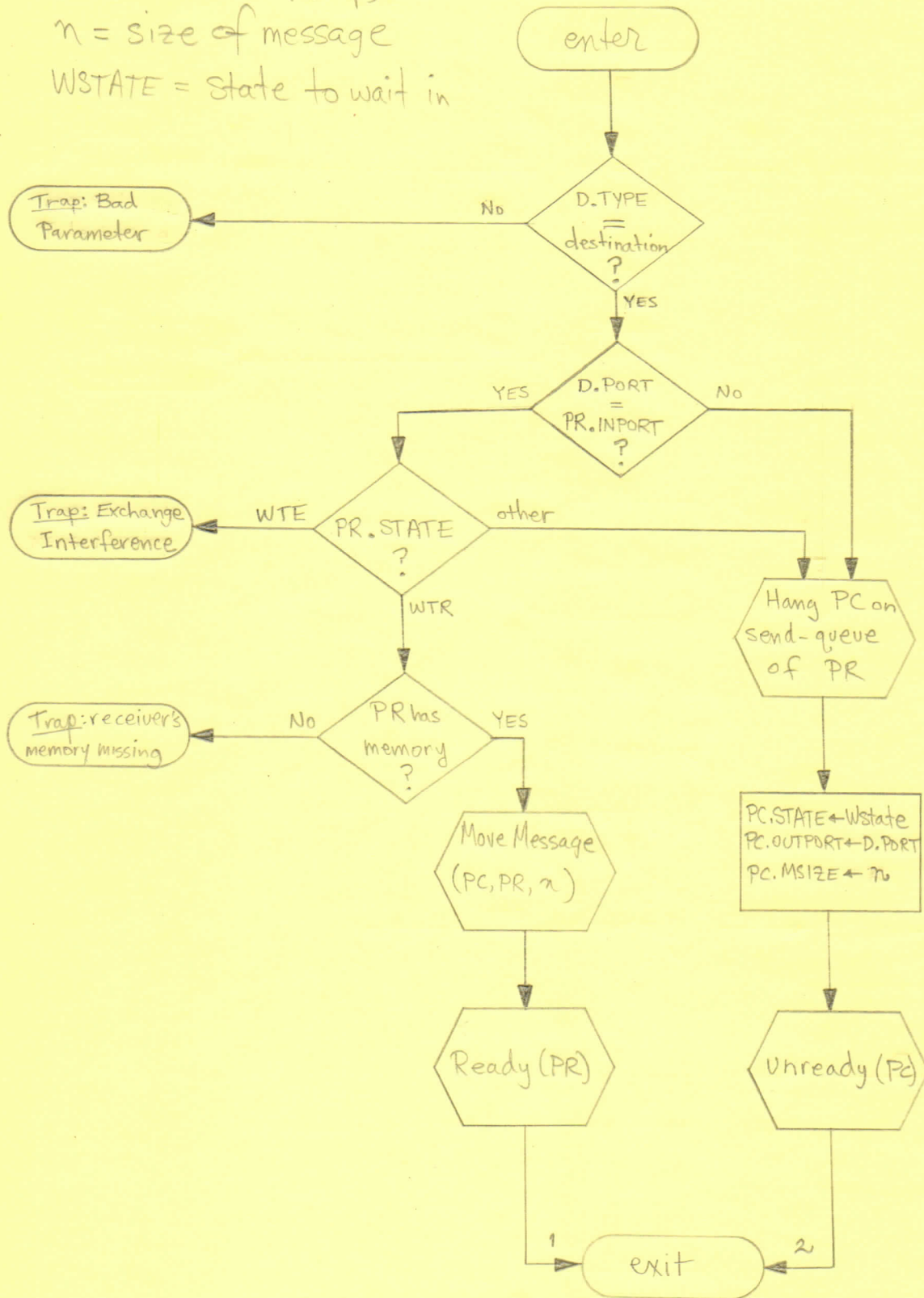
$Xmit(D, n, WSTATE)$

D = destination (cap)

n = size of message

$WSTATE$ = state to wait in

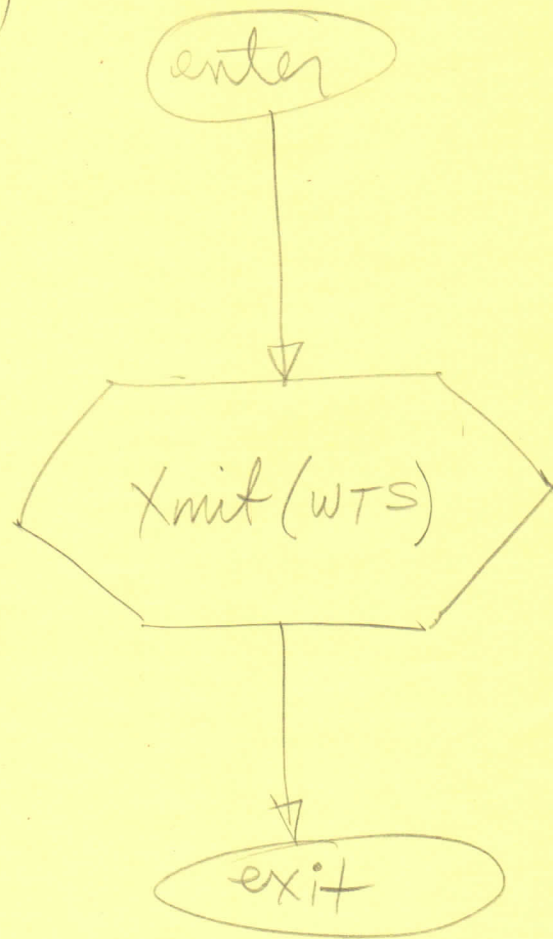
PC = current process
 PR = receiving process
 (from D)



Note Conditions on exit

- 1) Message transferred and both processes left ready
- 2) Current process hung in 'Wstate' waiting to send.

Send (D,n)



Stop / Restart

- 1) Stopping and then Restarting should always be transparent to the process (i.e no matter what it was doing at the time)
- 2) While a process is Stopped, nothing but a Restart can make it run. For example, if it was waiting for a signal when it was Stopped, the signal cannot be accepted until a Restart is done (i.e Restart will redo the Receive)
- 3) It must always be possible to look at a Stopped process and tell exactly what was going on when it was Stopped. Most crucial case is a non-accepted trap, which left process hung in a receive (forced exchange) even though there is no indication in the code what was going on.

To send a signal, process S must:

a) push a sequence of descriptors onto its stack

1) These descriptors may be:

a. capability references

b. immediate data descriptors

c. general descriptors of any other type

2) maximum of, say, 8 descriptors (limited by size of signal buffer.)

b) perform a Send(C, n) operation, which pops n descriptors off the stack, and sends signal to process R (pointed to by send-cap C which also gives port # k)

The Send operation does the following

a) S's slot #0 $\leftarrow$ cap to send to S on port k

b) For each descriptor (D_1 thru D_n)

1) if D_i is a capability reference, the capability is copied to slot i in the signal buffer, and the type flag for the slot is set to "cap"

2) otherwise D_i itself is copied into the buffer and the type flag for the slot is set to "datum"

c) Check receiver R's status

(1) if R is hung doing a receive for this signal (i.e. the "class" or "port" matches) then go to step d.

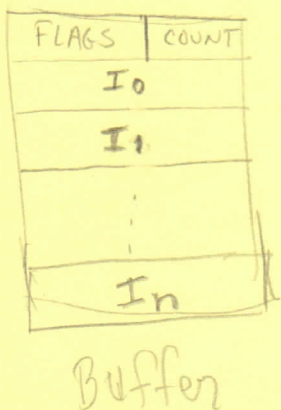
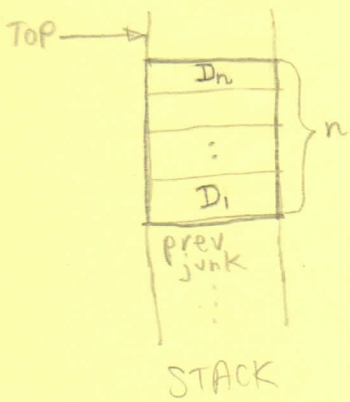
(2) otherwise, S hangs doing the send:

Ⓐ set state to "sending." (i.e. not ready)

Ⓑ store state (may need extra info since send instr is still in progress)

Ⓒ Remove S from ready-list and add to R's send queue

Ⓓ Run process now at head of list



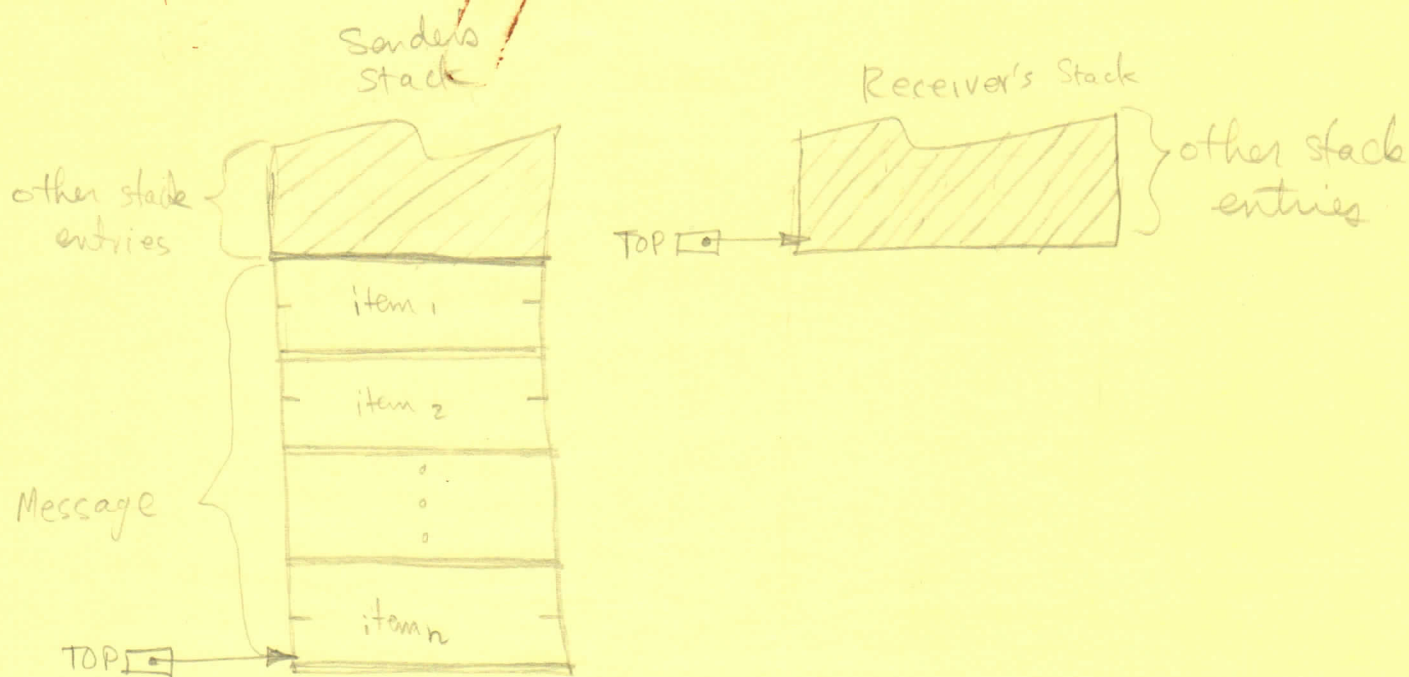
"block"
μsubroutine

(3) When process becomes ready again & reaches top of readylist, (i.e. receiver finally picked up the signal), need only pop stack and proceed.

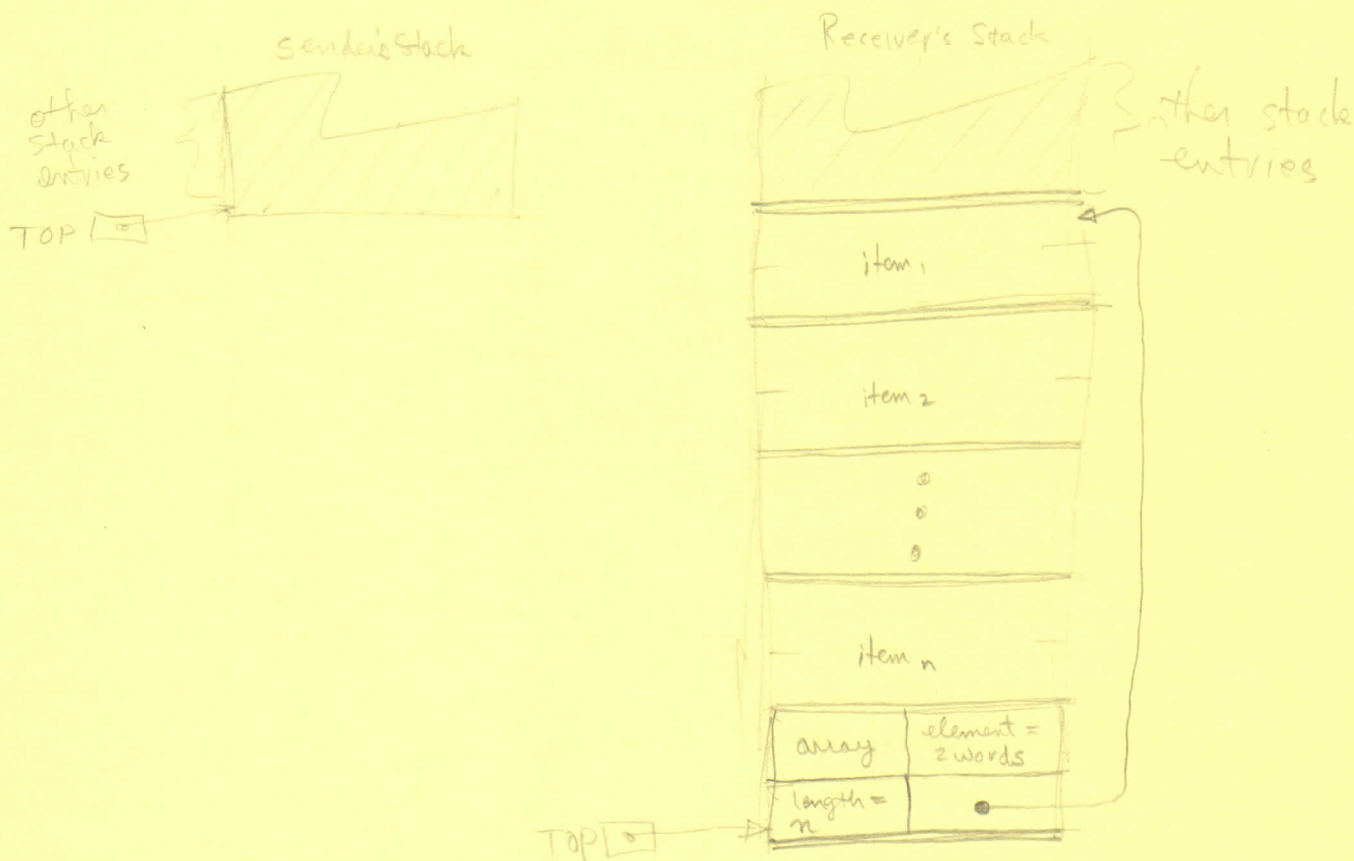
(4) If process S is stopped, it must be removed from R's send queue. State is left as stopped with p-counter pointing at the send. Restart should restart the send, thus reloading the buffer and rehashing (or, perhaps, succeeding this time)

c) Receiver already hung

- 1) Copy signal from S's buffer to R's buffer
- 2) Wakeup (R)



1a. Before Transfer

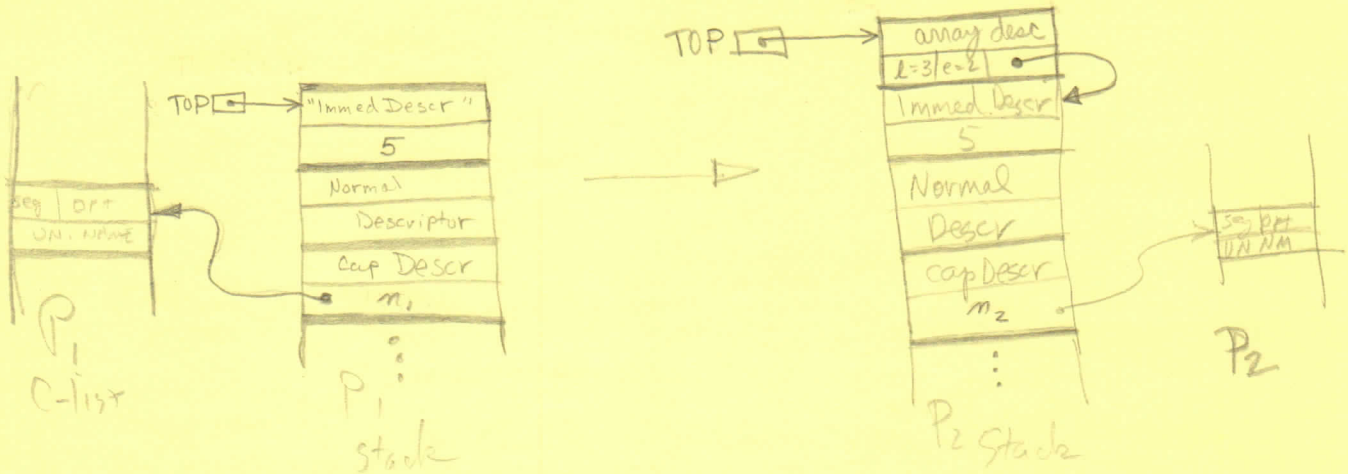


1b. After Transfer

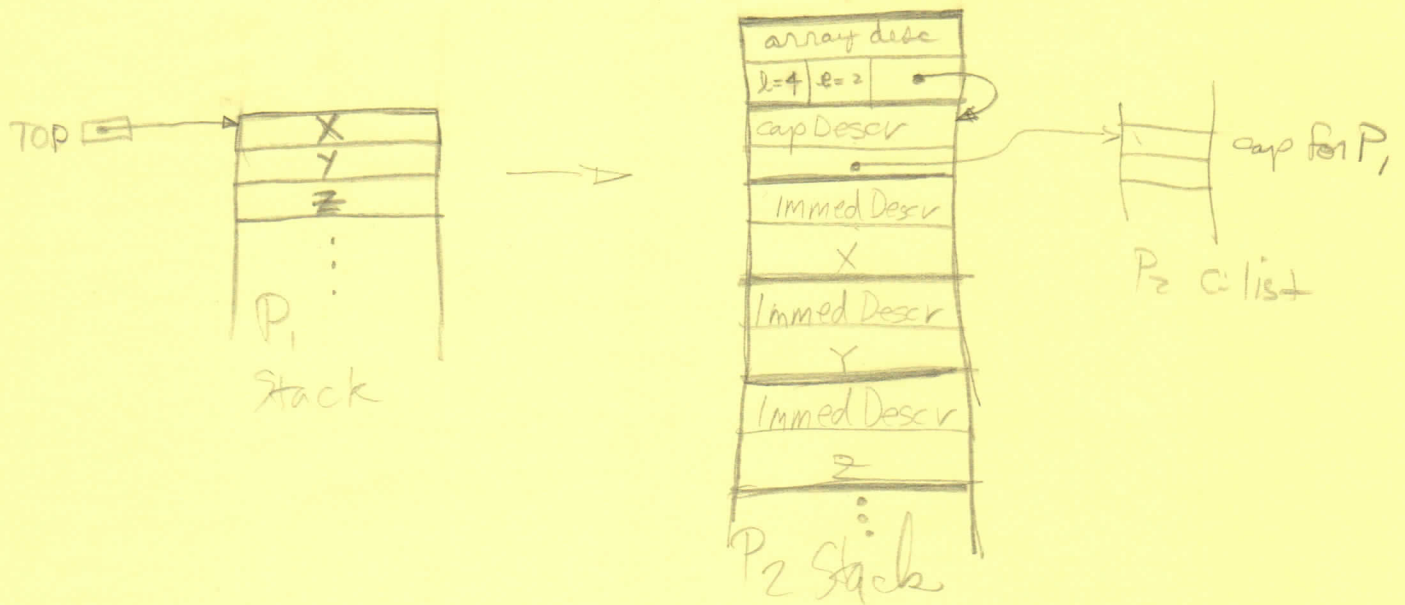
Figure 1: Message Transfer

message formats

1. SIMPLE $\rightarrow$ SIMPLE



2. APL $\rightarrow$ SIMPLE (trap)



Main Question on IPC as I/O Interrupt Replacement:

Can messages really serve as the medium for transmission of the data and control information, or are stop/start to be used for the synchronization and data to be passed through buffers, command & status words, etc. (Or some compromise)

Problems with using messages:

1) Possibly too much overhead

(mainly memory references) for

high volume / bandwidth transmissions.

Need to look at:

a) Optimum message passing algorithm
(e.g. "open stack" package)

b) Other schemes (circular buffers, etc)
using shared memory.

2) Synchronization problems, esp. with controllers attempting to send a message to a process which is already running. Note that

For process $\rightarrow$ controller starts, wakeup waiting problem does not arise, since test $\{$ "stop" $\}$ are atomic (i.e. in ucode). Controller waking up process is problem case. Wakeup-waiting is represented by a queued-up PTE. For normal processes, this means either:

a) The sender is hung

or

b) a buffering pseudo process is hung, and original sender continued.

For I/O, need to hang a special pseudo PTE which doesn't trap (or a regular one, and have I/O process catch own trap?) and let μ -coded controller go on. Assuming that the controller talks to only 1 process, only 1 pseudo PTE is needed (wakeup-waiting is switch, not counter) - controller wakes up process by calling routine which:

- 1) Checks whether its pseudo PTE is already hung.
If so, done.
- 2) If not, check whether target process is waiting for the "message".
If so, make it ready.
- 3) If not, hang the pseudo PTE on the target's queue

3) Buffering problems -

Really another side of synchronization problem: Controller must dispose of data (e.g. input tty chars) in a buffer. Message approach tries to use stack of manager process as this buffer. This is a problem, since process may be running:

a) Due to previous wakeup from controller
(see notes on "Timing for TTY Manager".)

b) Due to message from another process
(e.g. tty manager must accept output messages from user processes too!)

Even if process is so fast that it can guarantee to be ready to receive message in time (a above), it still might be busy servicing a call from another process (b above)

4) Complication of M-code - relative complication of, say, a ring buffer and the stack push/pop routines is not clear. Seems plausible that message approach is more complicated.

Because process will want to talk to controller and to other processes.

Conclusions: Messages should be used (not stop/start) to synchronize controllers and manager processes. However, these messages should carry little or no information; in particular messages from controller to manager should be null (or perhaps 1 word could be held in pseudo PTE - is this useful?) All data and control info should be passed through shared memory.

I Introduction

II Goals

III Intervention operations

Stop
Start

IV Messages

V Message operations

VI Traps

VII I/O Devices

messages and/or special memory

VIII Details - bottom up

Process table - manipulation operations

Ready list

Ready/Unready - interprocessor pre-emption

Quantum timers $\frac{1}{2}$ overflow traps to scheduler

Send - receive - exchange - swapping and traps to
memory/message handler.

Message format details

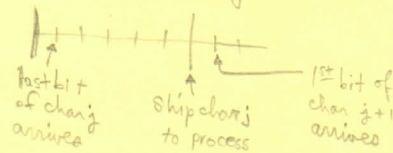
1. received as array

2. converted during APL $\rightarrow$ SIMPLE xmit in traps

17 July 72
①

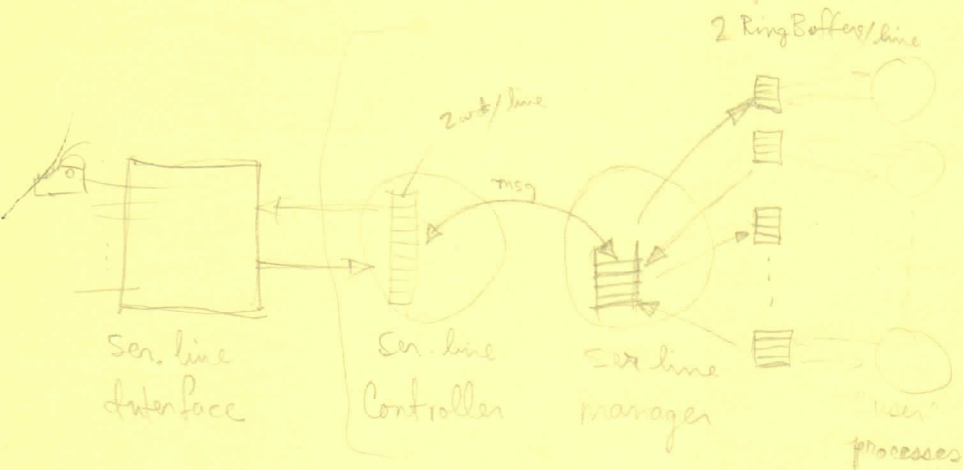
TTY MUX - Samples 660 times/sec, so char can complete at most every 1.5 msec. (with 64 lines at 10 chars/sec, couldn't quite get this rate - i.e 640 chars/sec). At each input-clock (ICLOCK) period, Tty MUX μ code could send a message consisting of all the newly completed input characters. As long as the Tty manager process could guarantee to dispose of these in < 1.5 msec, and do another receive, all is cool.

Could also slow down to OCLOCK rate (110 cps). Doesn't even require extra buffer word, since, at worst, start bit of next char will arrive 1.5 ms after current char is shipped out:



Actually for Tty, is even better, since char can be shipped before the 2 stop bits arrive, which leaves ~ 20 Ms before start bit of next char can arrive.

Each OCLOCK time, must also see if current outgoing character for each line has finished, and if so, request a new one from Tty manager.

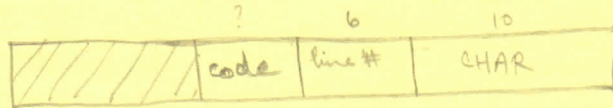


Manager process: This must handle panics. Does it do any echoing?

Break tables, etc?

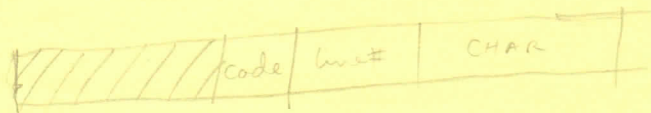
Can't ring buffers be internal to Tty manager, and all char flow be messages?

So, message from controller is var length seq of items:



where code = input char
output req.
other?

and, message to controller is also var length seq:



code = output char
disable line
other?

Note cruddy 2-word immediate descriptor business doubles amount of stuff transferred. Ugh. This is probably not prohibitive though.

(Also what is scheme for various devices (GSR, etc))

Considerations in message passing by TTY Mux μ -code.

- ① Does resultant message need to have 2 word items; wouldn't it be OK to pass an array of 1 word data.

What are more general implications for message system?
 1-word/item messages would work for APL traps, except for the process capref. Maybe this should be passed as an integer? Then trap messages could have 1-word elements. More generally, any all data message of n items could be sent by SEND($P, -n$)
 Hmm....?

- ② How is message transferred?
 Could be:

- Mux puts it in wired down buffer and calls copy-to-stack μ routine
- Mux has special code and stores into Manager's stack directly.
- Hybrid scheme: $\exists$ open, close, push, pop stack routines

Open(P, σ)

"opens" stack of PTE# P and returns σ , a one word

open-stack-descriptor:

LIMIT	TOP
-------	-----

 also addresses

Close(P, σ)

"closes" stack of PTE# P , updating TOP in accordance w/ σ

Push(X, σ)

pushes X onto σ - error (skip?) if overflow

Pop(X, σ)

pops item from σ and returns it in X . No underflow test is performed.

This seems pretty good.

18 July 72

(2)

Timing for TTY Manager

Worst case for TTY Manager

- Must get back in 1 bit time = 9 msec
- Suppose all lines but 1 (63 of them) need both input and output attention ...
And, at next bit time, 64th needs a 4th
- Then, must be able to handle each of 1st 63 lines in ~140 msec real time (i.e. disk/tape interference included)

Disk interference is of two kinds

- 1) Slowdown of CPU due to disk ctrl u-code
- 2) Loss of CPU due to disk manager process

Relative priority:

- 1) Clearly, disk controller is highest priority u-code
4 drives transferring continuously burn up ~50% of uproc (incl overhead)
- 2) Not really necessary for disk manager to be higher priority than tty manager, since
a) no lost data problems
b) chained commands $\Rightarrow$ no lost next prob (?)

assume tty manager highest priority process, and 50% of uproc burned up by disk. Then, have 70 msec CPU time per line in worst case.

i.e. 35 msec to get rid of input char
35 msec to provide new output char.

This is almost reasonable, but seems too close for comfort.

I/O as Send/Receive

Questions

1. What facilities are necessary to allow pseudo-processes to use Send/Receive?

A. Storage: PTE, data seg w/ stack?

B. Microcode: send & receive - similar to existing routines.

2. How would pseudo-processes use Send/Receive? How would this be better than start/stop + shared memory?

I/O Examples

A.TTY MUX

1. Output a character

(S) → (F) msg = char, line #

or, should it take a string?

2. Input a character

(F) → (S) msg = char, line #

3. Need output

(F) → (S) msg = line #

$$64 \text{ tys} \times 10 \text{ ch/sec}$$

$$= 640 \text{ ch/sec}$$

($< 1 \text{ ch/millisecond}$)

suppose CPU time
incl proc sw (x2)

$$= 200 \mu\text{sec/char.}$$

Then 20% of CPU to
handle tty i/o.

IPC Source language forms

I. SIMPLE:

a. Send:

SEND(D, X, Y, Z)

D = destination cap
X, Y, Z = message items

b. Receive

A ← RECEIVE(N)

N = port #

A = array holding
message (e.g. X, Y, Z)

c. Exchange

A ← EXCHANGE(N, D, X, Y, Z)

A, N, D, X, Y, Z
as above

II. APL - causes trap ("APL Runtime Escape" trap)

a. Send

SEND[C, X]

C = Channel no.
X = value (scalar or array)

compiles push C

push X

push n_s

trap 3 ← (# items to send)

n_s = APL runtime
action code
for SEND

b. Receive

RECEIVE[C, @X]

push C

push @X

push n_R

trap 3

C = Channel no.
@X = ref to var
 n_R = APL runtime
action code
for RECEIVE

[SIMPLE APL] IPC differences

a) No C-list on APL machine.

No caps in messages: in particular, no return link cap.

b) all items = 1 word on APL machine

conflicts w/ SIMPLE message format

(seq of 2 word items)

Ways to resolve:

1) Do conversions when moving message between memories of differing-language processes. (Conversion consists only of scalar $\leftrightarrow$ immed-descriptor.

2) Reformat SIMPLE messages into 'data' part and 'capability' part. Only 'data' parts in messages w/ APL procs.

a) restricts order of items

b) ruins array formatting for receiver.

3) APL $\rightarrow$ SIMPLE messages could become arrays with 1-word elements. Other direction is messier

Doesn't work for traps (i.e. all APL $\rightarrow$ SIMPLE messages) due to cap ref to implicit capability for trapped process.

Trap vector — in data segment

1. APL — system functions which
can take other functions

(i.e. compiler checks actual param
and inserts function # [as
integer] in code)

e.g. SETTRAP(TrapNo, Func)
etc

2. Simple — vector in data segment
containing function descriptors —
set & changed as with any other
such vector.

e.g. TRAPVECTOR [TrapNo] ← FUNC;

In both cases, need operation to declare position
of trap vector (source form = inline function)

1. Traps by both CPUs to SIMPLE CPU
2. Explicit sends, receives, exchanges
stops, starts only by SIMPLE CPU
3. SIMPLE CPU can STOP SIMPLE or APL CPU.

Traps

1) Signals - need to restart at other than next inst?
(i.e. restart)

2) Function calls - may want to monkey with, say, prev stack frame. Use external seg mechanism?

In general, local trap processing will be invoked to cope with failure of an n -arg operator where failure is described by k words of trap info. So, want $(n+k)$ argument function which returns same kind of result as the operation would have

Scalar $\longleftrightarrow$ immediate descriptor relation:

a) scalar $\Rightarrow$ immed descr

used: ① by operators prior to trap generation
(must be by operator, since only it knows format of top-of-stack)
② by APL $\rightarrow$ SIMPLE message xfer

b) immed descr $\Rightarrow$ scalar

Used: ① by trap handler processes to return scalar result on top of trapped process' stack.

Note: this fights with notion of array on receiver's stack.

② by SIMPLE $\rightarrow$ APL message xfer. (anything except immediate descriptor causes error.)

Note : Both "double" atomic actions (exchange and trap) are a little strange internally, since the send cannot really finish up as usual but must allow the stop or receive to be done.

In particular, exchange has two cases

① If target was waiting, message is xferd, target readied, exchanging process hung in receive, and rescan done.

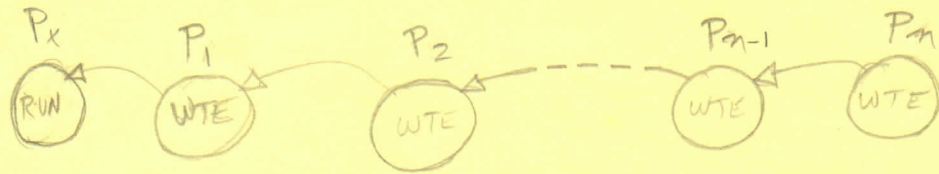
② If target not waiting, exchanging process must be hung on target's queue. When target does receive, exchanging process must then be hung doing its receive.

(Can't be done later due to original reasons for exchange.) So, original hang is in a new state, like wait-to-send, but with pending receive lurking.

Problem: what if the receive hits an already waiting message? Iccck!

Exchange-Interference Rules : Motivation

Gross problem - suppose $\exists$ chain of processes, each waiting to exchange to process to its left.



where x is computing. Now x does receive. CPE is then obliged to do a receive on behalf of all of $P_1 \dots P_n$.

Complication and timing problems are bad, and this situation is useless anyway.

Solution: prohibit it in the 1st place: Extra constraints on exchange

a) If Exchanging process already has a message on its reply-port, then give error.

b) If a Send is directed to a process which is in WTE state and targetport = reply port, then error.

also: port#0 (trap port) cannot be used for anything else (in particular, not for exchanges)

Otherwise, a situation could arise where the send half of a trap ended up violating the exchange interference rules, which is clearly unacceptable.

Note that checking is done on input param of receive; exchange; output is checked at capability creation time.

Note

Buffering messages via traps and pseudo PTEs

doesn't really solve wired-down-buffer problem at all, since I one wired down pseudo PTE for each message. This is somewhat better than a large (or at least vbl size) buffer, since limit is on # messages, rather than $\sum_{\text{messages}} \text{length}$. Need limit on # buffered messages which can be hung on a given process

Dynamic C-list allocation

Pro

- 1) allows multiplexing of C-list slots to be dynamic (RELEASE command) rather than static - easier to understand
- 2) Eliminates parameter-area in C-list - (i.e. makes cap-refs in stack really useful) also eliminates need for limit on # caps allowed.
- 3) allows C-lists to be shared between incarnations of same procedure segment.

Con

- 1) Complication in microcode
- 2) Requires indirection thru C-list references rather than compiled in constants. (Logically OK, since compiler can generate an initial "template" for C-list free chain and initialize some descriptor vbls to hold the references. Caps must then be loaded into C-list at process creation time to match the template.)