SYSC 5701
Operating System Methods for Real-Time Applications

Message Passing

Winter 2014

Message Passing

- kernel provides services for process interaction
  → communicate using messages:
  - send (message)
  - receive (message)
- establishes a logical link (channel) among processes involved
  – Several variations on this!

Link-Related Issues

- direct → process-to-process, blocking?
- indirect → buffered in mailbox, blocking?
- link capacity? buffering / queueing?
- message size? fixed? variable?
- pass message copy or reference?

To Block or Not To Block … ?

- blocking couples synchronization with messaging
  – increases determinism
  – determinism – simplicity, understanding ☺
- if not needed (i.e. not central to application objective) then may be contrary to asynchronous, event-driven goals (concurrency?) ☹
- may need to introduce extra "transport" processes to avoid blocking! – overhead! (later)

Un-Synchronized Services

send (…) send a message, no blocking
if receiver not ready – message lost

receive (…) receive a message, no blocking
if no message ready, none received

useful ?

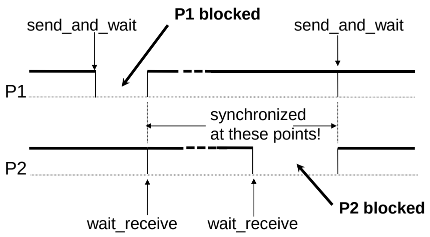
Synchronized Services

send_and_wait (…)
send message and wait (i.e. block) until received

wait_receive (…)
wait (i.e. block) until a message arrives

- requires no buffering of messages – sender and receiver synchronize @ message exchange
- shared memory impl^{n}: can pass message reference
- distributed system: must pass copy of message

Synchronized



How will Correct Processes be Involved?

1. identify both sender and receiver
2. identify only one of sender or receiver

1. identify <u>both</u> sender and receiver

send_and_wait(rcvP, msg)

wait_receive (sndP, msg)

2. Identify Only Receiver

send_and_wait(rcvP, msg)

wait_receive (msg)

- may have <u>multiple senders</u> waiting to synchronize with <u>same receiver</u>
- need queueing of senders for each receiver
  - FIFO? wait on sema4?
  - priority? queue structure?
- typical: PCB contains fields to support IPC

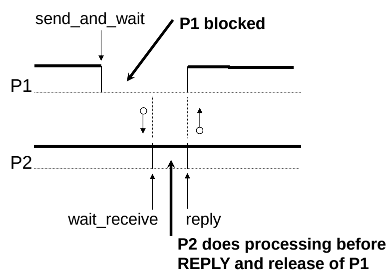
Variant: Non-Blocking Send, Blocking Receive

- typically identify only the receiver
- senders "give work to" receiver
- sent messages are queued, sender is never blocked
- receiver blocked only when no messages in queue
- more concurrency ☺ harder to synchronize! ☹
  - → use semaphores for synchronization!
- message issues (buffering?) – later!

Variant: Rendezvous

- **<u>blocking</u> send, <u>blocking</u> receive, <u>reply</u> to sender**
- sender/receiver synchronize
- **first** message: from sender to receiver
- receiver does some processing
  - → decides when to release sender
- **second** message: returned to sender
- 2 way communications!
- controlled/delayed release of sender

Rendezvous



Mailboxes: Indirect Communication

- mailbox = kernel supplied object to support message passing
- send to mailbox:
  - non-blocking
  - if receiver waiting, then receiver is given message and released
  - if no receiver waiting, message is queued

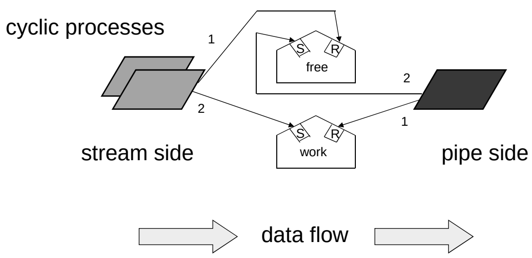
Mailboxes

- receive from mailbox:
  - block if no message ready
  - if message ready, obtain message from front of queue and leave
- may have multiple queued receivers
- messages passed to mailbox, not to explicit process(es) !

Mailbox Primitives

- typical service primitives:
  send (mailbox, message)
  receive (mailbox, message)
- often: dynamic create/delete of mailboxes

Mailbox Solution to Stream-2-Pipe Example



Messaging Implementation Issues

1. **Addressing**
2. Message Format
3. Memory Issues

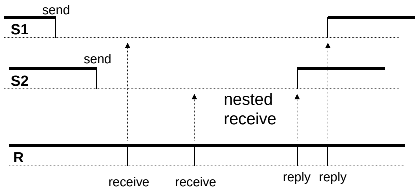
1. Addressing

- naming processes creates tighter coupling!
- how many named per communication?
  - sender & receiver?
  - just one?
- send to many → broadcast! (vs. multicast?)
  - useful mechanism in distributed systems

Rendezvous Addressing

- sender names receiver
- receiver accepts from any sender
  - receives sender's id, too (message format!)
- what about reply in a rendezvous?
  - if only one outstanding sender, no real choice
- nested rendezvous?
  - implicitly: reply to most recent sender first
  - explicitly: receiver decides order of replies

Nested Rendezvous



- might be preferred to allow S1 to be released first?
  - would require explicit naming in reply

How Can Processes be Identified?

- **"physical" id** – identifier assigned dynamically when process is created
  - e.g. pointer to PCB – simple, fast lookup
    - alternatives?
- requires "knowing" kernel services
  - e.g. "myID" in previous examples
- distributed systems?
  - could have two processes with same ID?
  - include "node" identifier in ID
  - larger names

Logical Names

- unique "globally known" names – assigned at design stage
  - **limitation**: no dynamically created processes ?
- kernel maintains lookup tables
  - map logical name to run-time id
  - run-time id's are hidden from applications
- add name to table when process created
- remove name when process deleted

Recall: Messaging Implementation Issues

1. Addressing
2. **Message Format**
3. Memory Issues

2. Message Format

- **How is message stored in buffer ?**
  - → "syntax" issue
- Is message one field of info? or multiple fields of info?
- Variable length? Need length field too?
- Multiple: may need message type id field
  - more overhead!

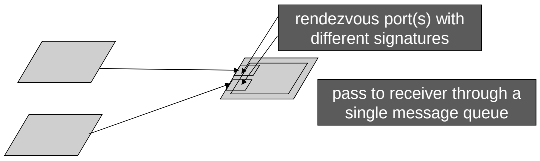
Why Might Message have Multiple Fields/Formats?

e.g. Ada: senders “call” a rendezvous “port” on receiver

- similar to calling a function defined by receiver
  – port call may have parameters
    - similar to param’s to function calls
- receiver <u>may wait for messages at multiple ports</u>
- each port may have different # parameters!

(con’t)

Multiple Rendezvous



Multiple Format Issues con’t

- messages for receiver are queued in a single queue
- messages may have multiple fields and different formats!
- message must include:
  – port identifier (message format id)
  – field for each parameter

Fixed Buffer Size

- kernel always deals with single sized buffers
  – fast, efficient services ☺
- may pack several different formats into one maximum sized buffer – variant records
  – all messages have single (max.) size ☺
  – some may have some unused space ☹

Variable Buffer Size

- more powerful → no wasted space ☺
- more overhead ☹
- buffer must include a size field
  – If variable sized fields → need size sub-fields too!

Recall: Messaging Implementation Issues

1. Addressing
2. Message Format
3. **Memory Issues**

3. Memory Issues

- does kernel require dynamic memory?
  - **yes**: where is it obtained from? *(gnarly?)*
  - **no**: (i.e. supplied by caller of services)
  - static pool → compromise
- access protection problems in different contexts?
  - e.g. Does memory manager h/w get in the way of sender/receiver accessing the same buffer?

Buffer Management

- how many buffers involved?
  - one from sender & one from receiver?
- pass message by copying **pointer** to buffer?
  - simple, fast @ message exchange ☺
- access protection h/w problems? ☹
  - processes can't share memory
  - overhead → buffer management policy ☹
- No shared memory? **copy message** from sender's to receiver's memory
  - copying overhead ☹

Static Buffer Scheme (Shared Memory)

- pool of "free" static buffers
- sender obtains buffer from pool
- sender copies message into buffer
- pass receiver a pointer to buffer
- receiver removes message from buffer
- receiver returns buffer to pool
- Simple; static memory, pool overheads

Dynamic Buffer Scheme (Shared Memory)

- create/delete as needed
- sender must create a buffer
- sender copies message into buffer
- pointer to buffer is given to receiver
- receiver disposes of buffer when done
- Simple? Dynamic memory?

Shared Memory Persistence Concerns?

- recall monitor examples
  - with shared memory: buffers might be created as dynamic variables (say in sender's stack) and then pass pointer to buffer
  - programmer must ensure that buffer still exists when receiver accesses stored message

No Shared Memory

- Sender arrives in kernel with message
- Receiver arrives in kernel with buffer
- Kernel copies message from sender's buffer to receiver's buffer
- Sender and receiver each manage their separate buffers after copy
- How to implement a non-blocking Send?
  - Kernel manages sender's buffer after copy?
  - Kernel copies to kernel's buffer before receive?

Summary: Enhanced Process Model with IPC Message Passing

- couples synchronization with message passing
  - kernel IPC handles details
- no "protection" burden on programmer ☺
  - kernel overhead ☹
- some architectural issues may influence kernel
  - not necessarily shared memory
  - distributed kernel in distributed system
- May be only communication mechanism that works for a strict process model

BOTTOM LINE

- process model creates an abstraction for the development of real-time systems
  - concurrency issues can be addressed in design! ☺
  - implementation may have overhead ☹
- if it goes "fast enough" – does it matter?
- **Tradeoff**:

  s/w engineering gains **vs**. overhead

Customizing a Process Model

- if a process model does not support a particular desired IPC mechanism
  - can often implement support using existing IPC
- already seen some monitor-style examples:
  - priority blocking when only FIFO available
  - timed services – a bit vague about the process that called TICK ☺ (timed services?)
  - synchronous message passing

Non-Monitor Constructs?

- using packages that are not based on monitor mutex assumption
- requires some design thinking – how to simulate IPC behaviour using existing kernel primitives?
- may be able to customize to application ☺
- often less-efficient than kernel-supported services ☹
- if services not available, may be only choice ??

Example: Readers and Writers

- "classical" example in o/s courses
- a resource (e.g. database) is shared
- readers: wish to read values **RReq**, **REnd**
- multiple readers can proceed concurrently
  - no interference
- writers: wish to write values **WReq**, **WEnd**
  - potential for interference !
  - must have mutual exclusion

Readers / Writers Issues

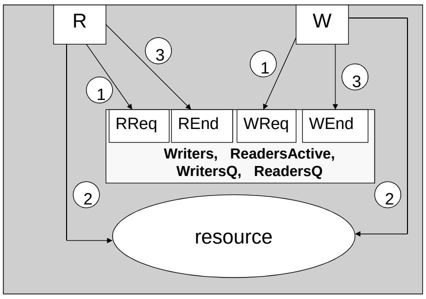
- priority (readers vs. writers), fairness / starvation
- allow: concurrent reads, mutually exclusive writes
- if writer active: make all newcomers wait
- once writer finishes: priority to waiting readers or writers?
- if reader(s) active: make new writer(s) wait
- should what new readers be allowed to start reading if a writer is already waiting?
- priority to writers (?) why ? starve readers?

Implementation 1: Monitor

- monitor coordinates access rights
- underlying assumption: **mutex in monitor**
- variables:
  - **Writers** – # yet to finish writing
  - **ReadersActive** – # actively reading
  - **WritersQ**, **ReadersQ**
    - hold blocked processes

Readers/Writers Monitor

(actually a wrapper)



WReq

- reader(s) XOR writer could be active

wait (**mutex**) ;
Writers ++;
if (Writers > 1) **||** (ReadersActive > 0)
  EnQueue (**WritersQ**, myID) ;
  sleep and signal (**mutex**) ;
// obtained mutually exclusive access to resource
signal (**mutex**) ;

WEnd

- only this writer has access to resource

wait (**mutex**) ;
Writers −−;
if Writers > 0 then
  awake(DeQueue(**WritersQ**))
else – no writers waiting, release readers?
  while **ReadersQ** not empty
    awaken (dequeue(**ReaderQ**))
    ReadersActive ++ ;
  signal (**mutex**) ;

awakened writer will signal mutex as it leaves

released readers do **not** signal mutex

RReq

- reader(s) XOR writer could be active

wait (**mutex**) ;
if Writers > 0
  EnQueue (**ReadersQ** ; myID) ;
  sleep and signal (**mutex**)
else – requested and obtained read access
  ReadersActive ++;
  signal (**mutex**) ;

when awoken: leave without signalling

REnd

- only reader(s) accessing resource

wait (**mutex**)
ReadersActive −−;
if (ReadersActive = = 0) && (Writers > 0)
  (i.e. this is the last active reader and a writer waiting)
  awake(DeQueue(**WritersQ**))
else – no writers to release
  signal (**mutex**) ;

awakened writer signals mutex as it leaves

Issues

- only calls kernel when necessary
  - low overhead ☺
- only block when necessary ☺
- mutex in monitor
  - gnarly programming ☹

Implementation 2: Message Passing

- Skeduuler process coordinates access rights
- Reader and Writer processes **rendezvous** with Skeduuler
- **send** – must explicitly identify receiver
- **receive** – from any sender
- sender's id is received as parameter
- **reply** – must identify reply-to process
  - **reply (reply-to-process-id, message)**
- can block sender until selected for reply

Skeduuler Process

- local variables:
  - **ReaderQ**, **WriterQ**
  - hold blocked processes for later reply
- **ReadersActive** – as before
- **Writers** – as before

Skeduuler: loops forever

receive (request, sender_id) ;
case request of
WREQ: // → writer arrives
  Writers++ ;
  if (Writers > 1) || (ReadersActive > 0)
    EnQueue (**WriterQ**, sender_id);
  else **reply**(sender_id, write_access);

WEnd case

WEnd: // → writer leaves
  Writers – – ;
  if Writers > 0 // release another writer
    reply (write_access, DeQueue (**WriterQ**));
  else // release any waiting readers
    while **ReadersQ** not empty
    reply (DeQueue (**ReaderQ**), read_access);
    ReadersActive + + ;

RReq case

RReq: // → reader arrives
  if Writers > 1 // block – writer yet to finish
    EnQueue (**ReaderQ**, sender_id);
  else // reader may proceed
    reply (sender_id , read_access);
    ReadersActive + + ;

REnd case

REnd: // → reader leaves

ReadersActive – – ;

if ((**ReadersActive** = = 0) && (Writers > 0))
// release a writer
reply (DeQueue (**WriterQ**), write_access);

Issues

- no explicit mutex manipulation – mutual exclusion is ensured **implicitly** by Skeduuler process
  - less gnarly burden to programmer! ☺
  - easier to understand and modify ☺
  - overheads! ☹
- for every call to monitor – **ALWAYS** context switch to Skeduuler
  - the penalty for using implicit process' mutual exclusion vs. explicit mutex semaphore! ☹

More Issues

- message passing vs. function invocation
  - making a request involves kernel service ☹
- extra process in system (**Skeduuler**)
- system resources ☹

So . . . why do most organizations use implementation 2 instead of implementation 1 ????