Operating system concepts

Process and thread management:

Communication

Inter process/thread communication principles

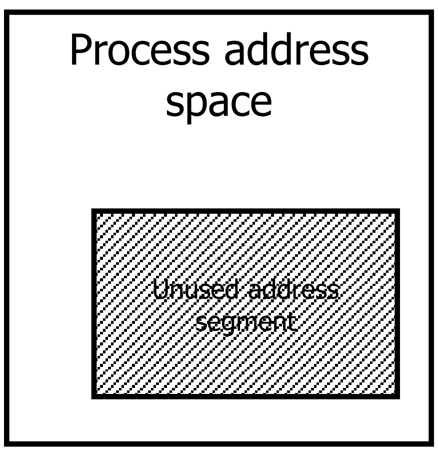
- Processes or threads that cooperate on common task:
  - operate on particular data set
  - share data between them, and/or
  - exchange data/messages/events/signals…, and/or
  - synchronize themselves (with some sync. mechanism)

- Basic communication principles include:
  - shared memory
  - messages (sending and receiving messges)
  - pipe (sending data into pipe, reading from pipe)
  - signals (events detected/generated by source thread that also require attention from receiving thread)

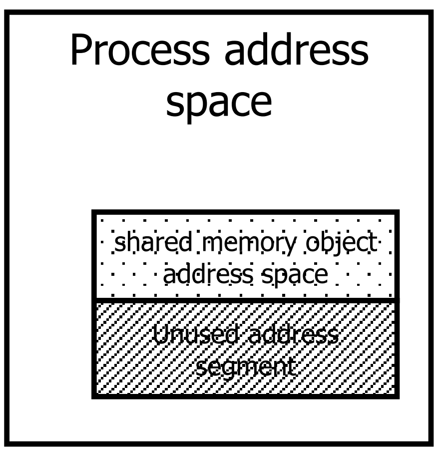
  - files ("offline" communication)

Shared memory

- All threads inside single process share that process address space – shared memory for threads
- Threads from different processes may create shared memory objects through system calls
  - part of address space of one process is used to map address space of shared object



Unused memory space of a process can be reserved as address segment for shared memory object using system functions.



Shared memory – system interfaces

- The usual interface can be described with:

`get_shared_segment (name, size, address, flags)`

  - **`name`** – identification for new or existing segment
    - might be number or string (even filename)
    - must be unique in given system
  - **`size`** – required shared memory size
  - **`address`** – where to place shared memory in process address space
  - **`flags`** – permissions, “create if doesn’t exist” flag, …
  - Function returns status or starting address or descriptor

- UNIX: **`shmget, shmat`**
- POSIX: **`shm_open, ftruncate, mmap`**
- Win32: **`CreateFileMapping, MapViewOfFile`**

Shared memory - protection

- Shared memory must be protected from simultaneous access (change) or data corruption may occur
- Critical section mechanisms, like mutex, binary semaphore, reader/writter locks, … can be used

- Shared memory may be the fastest communication method between threads (if synchronization is minimized)

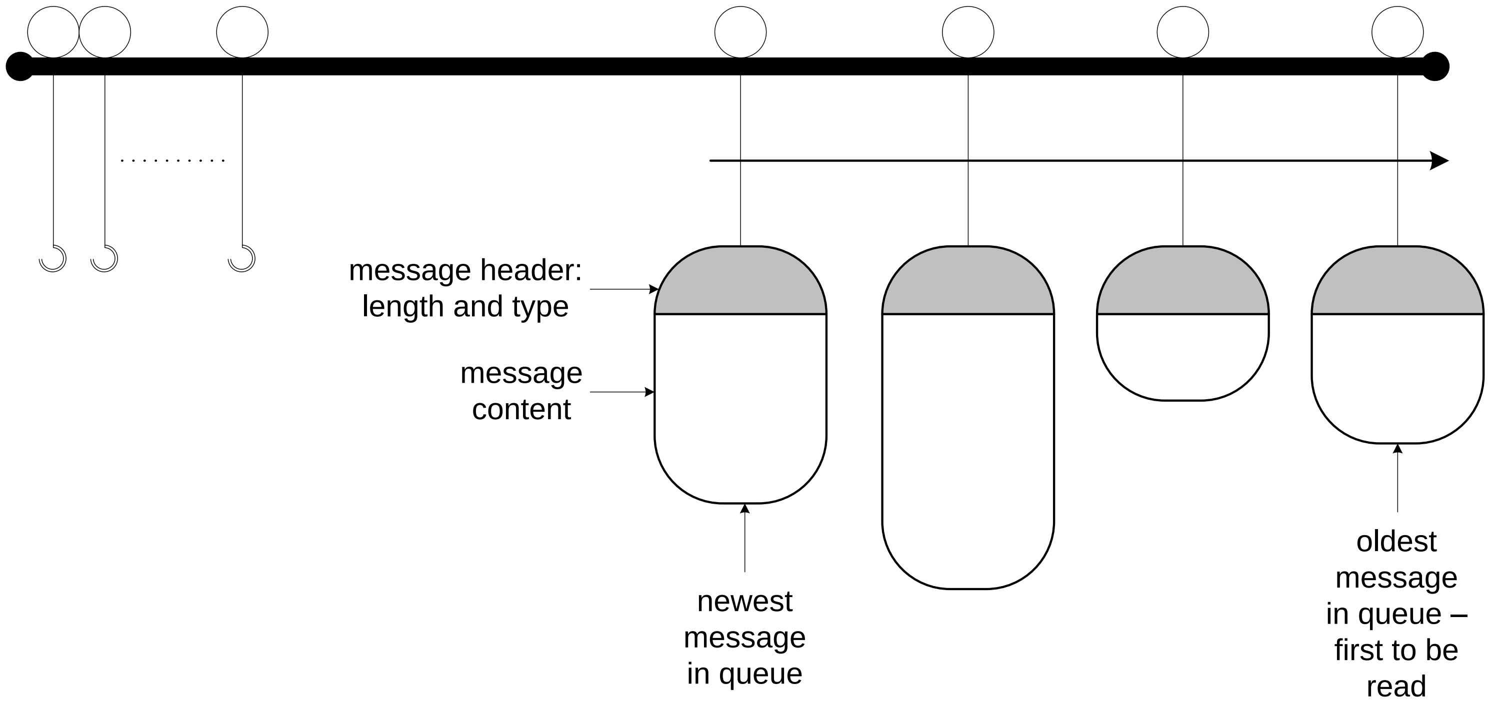
- Shared memory may be the source of hard to detect errors, due to neglected unprotected modification

Messages

- Message is a short information block sent from one thread to another
- Message is not directly delivered from thread to thread, operating system is used as communication channel instead
- When *sending a message*, message is put into message queue
- When *receiving a message*, message is taken from message queue
- *Message queues* are managed through operating system: creation, deletion, sending, receiving, statistic
- In some systems (i.e. Real-Time) for every thread there is an automatically created queue – messages are the primary communication mechanism

Message queues

- messages can be of different sizes and types
- message queues are *First In First Out* structures, first message put into queue will be first to be read and removed form queue
  - with some interface it is possible to read message of a specific type (even if is not first in queue)



Messages – system interfaces

- Creating message queue:
  `get_message_queue (name, flags)`
  - **`name`** – identification for new or existing message queue
    - might be number or string (even filename)
    - must be unique in given system
  - **`flags`** – permissions, “create if don’t exist” flag, …
  - returns descriptor (ID) of created message queue

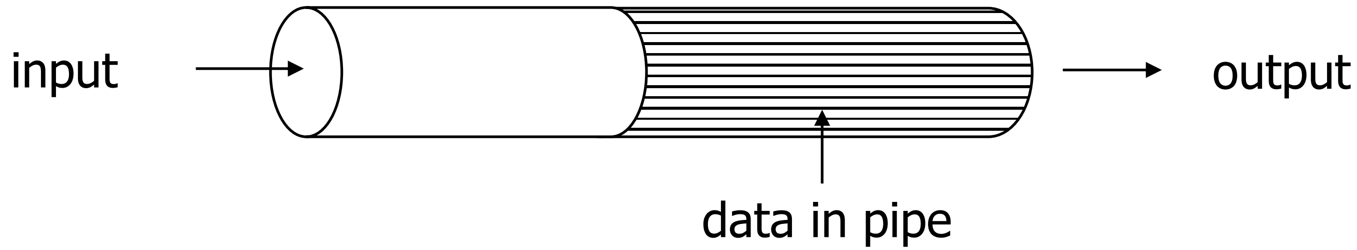
- Sending or receiving messages:
  `send_message(queue_id, pdata, len, flags)`
  `receive_message(queue_id, pdata, len, flags)`
  - pdata – pointer to message to send or where to save received message
  - len – length of message
  - flags – e.g. whether to block thread if queue is full/empty

Messages - implementations

- UNIX
  - `msgget, msgsnd, msgrcv`
- POSIX
  - `mq_open, mq_send, mq_receive`
- Win32
  - `MQCreateQueue, MQSendMessag, MQReceiveMessage`

Pipe

- Pipe is a FIFO structure (like message queue)
  - FIFO is the *only* way to send and read data (no searching)
- Pipe has two sides: *input* and *output*



- Data is sent to pipe through input side, and read from output side
  - pipe has two *descriptors* – one for each side

- There is no granularity and type (unlike with messages)
- Reading from pipe removes read data

Pipe – system interface

- Accessing pipes is similar to files: open, read/write, close

- Creating/opening pipe:

```
create_pipe (name, descriptor(s), flags)
```

  - **`name`** – identification for new or existing pipe
    - might be number or string (even filename)
    - must be unique in given system
    - **may not be supported in all implementations!**

  - **`descriptor(s)`** – descriptor for requested pipe side, or both descriptors

  - **`flags`** – “which side: input or output or both”, “block until other side is open?”, permissions, …

Pipe – system interface

- Reading/writing to/from pipes

```
write_to_pipe  (input_desc, data,size,flags)
read_from_pipe (output_desc,data,size,flags)
```

  - **input/output_desc** – input/output pipe descriptor
  - **data** – address of data to be sent to pipe, or where to be put if reading from pipe
  - **size** – "**data**" size to be written to pipe or read from
  - **flags** – "block if full/empty or not", …

  - returned value is usually data size sent to or read from pipe, or error code if unsuccessful

- Implicit pipes in shell
  - e.g.: **cat** ***file1*** **|** **grep** ***name*** **|** **sort** **>** ***file2***

Pipe - implementations

- UNIX
  - **`pipe`** – anonymous pipes
    - processes must be *related* (parent – child) to use
  - **`mknod, open, close`** – named pipes
    - name exists in file system, processes do not have to be related
  - **`read, write`**

- Win32
  - **`CreatePipe`** – anonymous pipes
  - **`CreateNamedPipe`** – named pipes
  - **`CreateFile, CloseHandle`**
  - **`ReadFile, WriteFile`**

Signals

- Signals announce asynchronous events
- Similarity with interrupts:
  - *interrupts* are used on *hardware level*, where they signal the processor with request for “special” processing
    - devices generate interrupts – processor (OS) handles them
  - *signals* are used on *operating system level*
    - OS or threads generate them – targeted threads handle them
- OS send signal to process to request special processing from threads – as reaction to event
    - e.g. when key is pressed on keyboard, interrupt is generated; in interrupt processing routine a signal is sent to thread
- Thread can send a signal to another thread (through OS interface, not directly), e.g. to ask for termination

Signals

- Thread can react on signal in several ways:
  - ignore signal
  - handle signal with user defined function
  - handle signal with default function (most default behaviors include thread/process termination!)
  - hold signal for now (delay its handling until some future time when behavior changes)

- Signals usually don’t carry additional information – only signal number
  - In extended interfaces (e.g. Real-Time), signal may carry additional value or pointer

Signals - interface

- Define “signal mask” – behavior for particular signals

```
signal_set (signal_id, pfunction, param)
```

  - **`signal_id`** – signal identification number
  - **`pfunction`** – signal processing function – called on signal reception, or may be an constant indicating:
    - ignore signal
    - handle signal with default function
    - hold signal
  - **`param`** – optional parameter to function

Signals - interface

- Send signal to another thread

```
signal_create (task_id, signal_id, param)
```

  - **`task_id`** – target task whom signal will be sent
  - **`signal_id`** – signal to be sent
  - **`param`** – optional parameter to be sent with signal

Signals - implementations

- On some (UNIX) systems signals are process oriented – mask is defined for the process
- Thread signal handling is a newer principle
  - differently implemented on different systems
    - read manuals carefully!
  - interfaces are defined, but rarely fully implemented (or as specified!)
- UNIX/POSIX
  - **`signal, sigset, sigaction`** – define behavior for particualr signal
  - **`kill, sigqueue, raise, pthread_kill`** – send a particular signal to thread/process

Files as communication mechanism ?!

- Communicating using files:
  - *sender* thread creates file and fills it with data
  - *receiver* thread (later) opens file and read its content

- Communication is "static"
  - *sender* must produce all data and only then send them to *receiver* (through file system)
  - thread synchronization might be required – receiver must wait till sender complete its operation

- Positive aspects
  - data in file can wait for receiver much longer than with other communication mechanisms
    - it will persist even if computer goes offline or is restarted
  - data size may be much larger than with other comm.