Multi-Threaded Application Case Study

Upon sensor trigger:
- Capture image from corresponding camera
- Process image
- Write image to disk
- If necessary, send e-mail report

NOTES:
- Image processing > minimal time between triggers
- Send e-mail may have large delay (retry)

IPC, Synchronisation and Multithreading

IPC Overview
- Pipes ✓
- System V IPC
  - Message Queues
  - Shared memory
  - Semaphores
- Windows-specific
  - Events
  - Critical section / Mutex
- Network
  - Sockets
  - Remote Procedure Call

SysV IPC Descriptor
- Message queue, shared memory and semaphore.
- Get: find/create.
- Ctl: retrieve information, change permissions, remove.

OS Table
- Integer id
- Descriptor
- Id mod #entries
- Perm.
- Name (key)
- Creator UID, GID
- Owner UID, GID
- Permissions: rwxrwxrwx
Message Queues

- System V
- A queue of variable-sized (small) messages.

msgget: Return message queue descriptor (may create new queue).
msgctl: Modify parameters, remove queue.
msgsnd: Send message.
msgrcv: Receive message.

Message Queue Structures

msgsnd(qid,msg,count,flag)

msg: positive integer type; byte data[count]
- Allocate space and copy message data.
- Allocate message header
  - Message type, size
  - Pointer to message data.
- Update statistics.
- Wake up sleeping reader processes.
- But if total bytes exceeds queue limit:
  - Sleep sender process unless IPC_NOWAIT flag.
**msgrcv (qid, msg, maxcount, type, flag)**

- Type = 0: Returns first message.
- Type > 0: Returns first of specified type.
- Type < 0: Returns lowest <= |type|
- If size > maxcount, fail or optionally truncate message.
- Remove message from queue and return.
- If no message found, sleep process; option IPC_NOWAIT immediate return.

---

**Message Queue Application**

- Server with multiple clients on a single host (i.e. not a network application)
- Bi-directional data flow in a single queue
  - Requests to server sent as type = 1
  - Replies sent as type = pid of client
- Collaborative design
  - Insecure against malicious client

---

**Other OS**

- Minix: Message passing kernel but no queue – messages are synchronous (rendezvous).
- Windows: User input events are sent to processes as queued messages.

---

**Shared Memory**

- Processes share a region of memory.
Shared Memory

- **shmget**: Create/find a shared memory segment by name
- **shmat**: Attach shared memory segment as a region of process address space
- **shmdt**: Detach shared memory segment
- **shmctl**: Retrieve statistics, modify ownership and permissions, remove segment.

shmget (key, segsize, flag)

- Allocate a region structure.
  - One region structure required for each region of a process’ address space.
- Find/create the segment.
- Return memory identifier (integer)

shmat (shmaddr, shmaddr, flag)

- Check permissions: (process address regions have access permissions).
- If caller has not specified address to use (shmaddr), find hole in process’ address space.
- Check limit on process’ memory use.
- Attach region to process’ address space.
- Update statistics.
- Return address of shared memory.

shmdt (shmaddr)

- Find region in process that is shared and starts at specified address.
- Detach region from process.
- Update statistics.
shmctl (shmid, cmd, buffer)
- Query status
- Change permissions and owner
- Lock segment in memory (underlying pages cannot be paged out)
- Unlock
- Remove segment

Semaphores
- Dijkstra (1965)
  - Process synchronisation (critical section, resource)
- Atomic operations
- Wait (try and decrease: p; down)
  - If sem > 0, decrement and proceed (semaphore acquired)
  - Else, sleep
- Signal (increase: v; up)
  - Increment semaphore, wake sleeping processes.

Semaphores
- semid = semget (key, numsems, flag):
  - Find/create a set of semaphores.
- semop (semid, semops, numsemops):
  - Simultaneously perform operations on selected semaphores associated with semid.
- Semctl (semid, cmd, semnum, arg):
  - Set/get value of semaphore(s), Retrieve statistics, Set permissions, Remove semaphore set.

Deadlock
- Process A
  - P(sem1)
  - P(sem2)
- Process B
  - P(sem2)
  - P(sem1)

Deadlock
- Process A
  - P(sem1,sem2)
- Process B
  - P(sem2,sem1)

No deadlock if P(...) is atomic
Multi-Threaded Application Case Study

Upon sensor trigger:
- Capture image from corresponding camera
- Process image
- Write image to disk
- If necessary, send e-mail report

NOTES:
- Image processing > minimal time between triggers
- Send e-mail may have large delay (retry)

Windows Thread/Process Comm.

- Event
- Critical section / Mutex
- Message
- Pipe

Event

- A system object: TRUE/FALSE
  - hEvent = CreateEvent (security, manual, initial, name)
    - security, name: for sharing between processes
    - initial: initial state TRUE/FALSE
    - manual: Auto reset after wait (FALSE)
- Settable, resettable
  - SetEvent (hEvent)
  - ResetEvent (hEvent)
- Waitable (wait until set)
  - WaitForSingleObject (hEvent, timeout)
    - Auto reset option

Using Event to Control a Thread

- Create myEvent with manual ← FALSE
- Worker thread:
  while (1)
  {
    WaitForSingleObject (myEvent, …);
    // Do heavy computation
  }
- Main thread:
  // When computation is to be activated
  SetEvent (myEvent);
### Monitor Thread

- **Monitor thread:**
  - For each thread \( t \) to be monitored
    ```c
    SetEvent (heartBeatRequest[t]);
    ```
  - Sleep (responseTime);
  - For each thread \( t \) to be monitored
    ```c
    // Check for heartbeat response
    s = WaitForSingleObject (heartBeat[t], 0);
    if (s == WAIT_TIMEOUT)
        // Declare thread \( t \) unhealthy
    ```
- For threads requiring longer timeout, allow multiple chances

### Worker Thread

```c
while (1)
{
    // Wait on heartBeatRequest[me] or
    // thread activation object(s)
    s = WaitForMultipleObjects (…);
    if (s == HEARTBEATOBJECT)
        SetEvent (heartBeat[me]);
}
```

### Critical Section & Mutex

- A system object that can be owned by only one thread at a time
  - Mutex: can be used in multiple processes
- **Critical Section**
  - `EnterCriticalSection`: Wait, claim ownership
  - `LeaveCriticalSection`: Current owner calls to release ownership
- Mutex:
  - Wait for object to claim ownership (e.g. `WaitForSingleObject`)
  - `ReleaseMutex` to release ownership

### Semaphore

- Wait (e.g. `WaitForSingleObject`) is try-and-decrement (\( p; \text{down} \)).
- **ReleaseSemaphore** (\( \ldots, \text{count}, \ldots \)) is increment by count (\( v; \text{up} \)).
- Semaphore is created with initial value and maximum value.
- A \((0,1)\) semaphore is similar to an event.
**Messages**

- GUI actions come to threads as a queue of messages.

  - **WaitMessage**: suspend process until new message is queued.
  - **GetMessage**: removes from queue for processing. Select by type, window.
  - **SendMessage**: synchronous.
  - **PostMessage**: asynchronous.

  - Messages are directed to windows
    - Except **PostThreadMessage**
  - MFC, .NET: abstract interface

---

**Threaded Class**

- Encapsulates thread
  - **Start** and **stop** methods control thread

  - thread... methods: override in derived class → thread calculations

---

**ThreadedConsumer**

- Consumes items from its own queue.
- Append method used by producers.

  - **Start**
  - **Stop**
  - **Append**

---

**Shared Objects**

- **Updater thread**
  - **Local copy**
  - **Put**
  - **Get**

- **Reader thread**
  - **Local copy**
  - **Get**

- **Critical section encapsulation**
  - **Master copy**
Multi-Threaded Application Case Study

Upon sensor trigger:
• Capture image from corresponding camera
• Process image
• Write image to disk
• If necessary, send e-mail report

NOTES:
• Image processing > minimal time between triggers
• Send e-mail may have large delay (retry)

Multithreaded Design

Network IPC

• Sockets
  • Remote Procedure Call

Sockets

• Represent network access point
  – One end of a TCP connection
  – UDP port on particular IP address

Server TCP
  • bind socket to port
  • listen for connect requests
  • accept connection on new port

Client TCP
  • connect to remote port
**TCP Connection**

A TCP connection is specified by the two end IP addresses and the two port numbers.

**UDP Connectionless**

- **sendto**: specify destination address (IP address and UDP port)
- **recvfrom**: Returns sender IP address and port, along with data.

**RPC**

- Request-reply model
- Request is a procedure call
- Marshall parameters into network message.
- Server decodes message, calls actual procedure, encodes reply, waits for another message.
- Client procedure decodes reply and returns result
XDR
- Standard data representation for network transmission (cf ASN.1)
- Integer: 4 bytes, big endian (high byte first)
  - E.g. Integer value 3
    - Big endian: 0 0 0 3
    - Little endian: 3 0 0 0
- Float: IEEE standard

RPC call
XID = transaction ID
CALL = 0
RPC version = 2
Program number
Program version number
Procedure number
Credential (variable length)
Verifier (variable length)
RPC parameters

RPC reply
XID = transaction ID
REPLY = 1
Reply Status: 0=MSG_ACCEPTED; 1=MSG_DENIED
Verifier (var length)
Accepted status
0=ACCEPTED
Rejected status
0=RPC_MISMATCH
1=AUTH_ERROR
Auth status
Low
High
RPC results
Lo
Hi

IPC Summary
- System V IPC
  - Message Queues
  - Shared memory
  - Semaphores
- Windows-specific
  ** Application
  - Events
  - Critical section / Mutex
- Network
  - Sockets
  - Remote Procedure Call