Real-Time Programming

Lecture 6

Farhang Nemati

Spring 2016

Repetition

- Semaphores/Mutexes
  - Deadlock, Task Starvation
- Condition Variables
- Monitors
- Barriers

Message Passing Based Synchronization and Communication

- Synchronization and communication among processes
- Thread/Tasks within a process can also use it
  - With threads, using shared variable synchronization and communication has higher performance

Message Passing Based Synchronization and Communication

- Synchronization Model
  - Asynchronous
  - Synchronous
  - Remote Procedure Call
- Naming of Source/Destination
  - Direct Naming
  - Indirect Naming
  - Symmetry
- The Structure of Message

Synchronization Model; Asynchronous Message Passing

- The sender continues; it does not wait for the message to be received
- Example: Posting a letter

```
…
async_send (message);
//does not wait ...
```

- + No need to wait
- + Program decomposition; decreases dependency
- - It might need too many buffers for keeping messages
- - More complex compared to synchronous
- - Testing and verification is more difficult

Synchronization Model; Synchronous Message Passing

- The sender can proceed only if the message has been received by the receiver
- Example: Phone call; the caller waits until the receiver is verified before giving any message

```
//sender                                  //receiver
sync_send (message);                      sync_receive (message);
//waits until message is sent ...         //waits until message is received ...
```

- + Simple
- - Introduces dependency among tasks/objects
- - Tasks might wait too long

Synchronization Model; Remote Procedure Call (RPC)

- The sender (caller) waits for a reply from receiver
- Example: Phone call if the receiver can reply immediately

Naming of Source/Destination; Direct, Indirect Naming

- Direct: The sender directly names the receiver

```
//task1
send_to (task2, message);
```

- Indirect: The sender names an intermediate entity, e.g., socket, channel, mailbox

```
//task1
send_to (mail_box1, message);
```

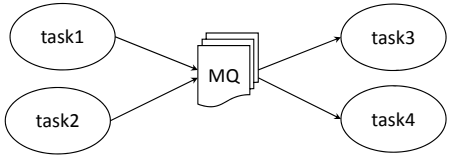
- Symmetry: Both sender and receiver name each other

```
//task1                          //task2
send_to (task2, message);        receive_from (task1, message);
```

Or:

```
//task1                          //task2
send_to (mail_box1, message);    receive_from (mail_box1, message);
```

Message Queues



- A number of messages can be queued in a message queue
- Variable size of messages
- Tasks can send messages to a message queue or receive messages from it
- Multiple tasks can write to or read from same message queue

POSIX Message Queues

- Used for asynchronous message passing among tasks/processes
- The waiting tasks are queued in priority based manner
- The messages in a message queue can be prioritized
- The messages with the same priority level are queued in FIFO manner

POSIX Message Queues

```
#include <mqueue.h>
```

- Open a message queue

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- `oflag` (a bit mask):
  - `O_RDONLY`: Can only receive messages from the queue
  - `O_WRONLY`: Can only send messages to the queue
  - `O_RDWR`: Can receive messages from the queue and send messages to it
  - `O_CREAT`: Is together with two more arguments; mode_t and mq_attr . To create a message. It fails if `O_EXCL` is set and the queue already exists
  - `O_EXCL`: If this flag is set and O_CREAT is not set undefined behavior!
  - `O_NONBLOCK`: Whether the senders or receivers should block on the queue or not. Blocked tasks are waited in a priority based queue

POSIX Message Queues

- Open a message queue

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- `mode` (bit mask): Access permissions like file access permissions in Linux, for example 666 means all users may read and write the file
- `attr` (attribute object): contains the attributes of the queue

POSIX Message Queues

- Message queue attributes

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);


struct mq_attr {
    long mq_flags;   /* Message queue description flags: for example O_RDWR */
    long mq_maxmsg;  /* Maximum number of messages on the queue */
    long mq_msgsize; /* Maximum message size (in bytes) */
    long mq_curmsgs; /* Number of messages currently in queue */
};
```

POSIX Message Queues

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- Create/Open message queue example:

```
mqd_t q1;
struct mq_attr q1Attr;
q1Attr.mq_msgsize = 20;
q1Attr.mq_maxmsg = 50;
q1 = mq_open("/myqueue", O_CREAT | O_WRONLY | O_NONBLOCK, 0700, &q1Attr);
```

- open an existing message queue

```
mqd_t q2;
q2 = mq_open("/myqueue", O_RDONLY | O_NONBLOCK); /* mode and attr not needed*/
```

POSIX Message Queues

- Retrieving attributes of a message queue

```
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

- Modifying message queue attributes

```
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
```

  - The maximum message size and the maximum number of messages are not changeable

POSIX Message Queues

- Send a message to a message queue

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
```

- Receive a message from a message queue

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
```

POSIX Message Queues

- Close a message queue

```
int mq_close(mqd_t mqdes);
```

- Remove a message queue

```
int mq_unlink(const char *name);
```

  - removes the queue and the queue is marked to be destroyed when there are no links to the queue

Message Queues; Exercise

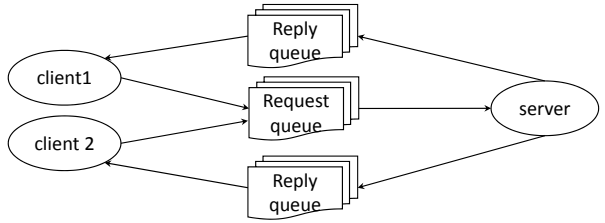
- Create two tasks; task1 and task2
- task1 creates a message queue which can contain at most 10 messages and the size of each message is at most 20 bytes.
- Every 100ms task1 generates a message and puts it into the queue. It should block if the queue is full
- task2 gets the messages from the queue and print it with period of 50ms. It should block if the queue is empty
- The task1 will send 100 messages and then it sends a final message (for example “stop”) and ends. task2 should end when it receives a stop message.
- Which task will be blocked mostly?

Message Queues; Solution

- See the attached program (msgqueue.c)

Client-Server Communication

- A special case of message passing based communication
- Clients issue requests to a server asking for some service
- Server accepts requests from clients and provides services to them
- Inter-task communication (usually message queues or pipes) are used to implement client-server communication



Client-Server Communication

- The server creates a request queue where clients can send their requests
- Each client creates its own reply queue. The client sends the queue id of its reply queue as a part of the request message. The server sends the reply to queue with this queue id.

Pipes

- An alternative for message queue
- A pipe has a read end and a write end
- Is treated as a (virtual) I/O device.
  - Uses I/O operations, e.g., the same operations that are used with files
  - read(), write() are used to read from, write to a pipe respectively
- Data written to the write end of a pipe can be read from the read end of the pipe

POSIX Pipes

```
#include <unistd.h>
```

- Create a pipe:

```
int pipe2(int pipefd[2], int flags);
```

  - Creates a pipe which is a unidirectional data channel
  - `pipefd[0]` refers to the read end of the pipe and `pipefd[1]` refers to the write end of the pipe
- Reading from and writing to a pipe:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- Close a pipe:

```
int close(int fd);
```

Sockets

- End points for communication over network
- Data streaming (TCP): Uses I/O operations, e.g., read(), write() are used to read/write.
- Data packets (UDP)

Signals

- Software interrupts
- There are 31 signals that a process or task can handle
- Mostly used for error and exception handling and not for normal communication
- To notify processes/tasks of occurrence of important events such as hardware exceptions, killing a process, etc.
- A routine is associated with a signal; if a signal is arrived the routine is executed
- There is a default routine for each signal, e.g., signal CTRL+C breaks the running process

Signals

- Signals can be disabled; the process will not receive them
- Signals received by a process is sent to one of its tasks that has enabled receiving of signals
- A task can send signals to
  - Itself
  - Any other task in its own process
  - Any system-wide public task
  - Its own process
  - Any other process in the system
- No history; no record of how many times a signal has arrived
  - POSIX queued signals record the history

Signals

- Associate a handler routine with a signal:

```
signal (int signalNumber, void(* function)())
```

- Send a signal to a task

```
int pthread_kill(pthread_t thread, int sig);
```

- Send a signal to the current task (itself)

```
raise (int signalNumber)
```

Signals

Example

```
void start()
{
    signal (SIGTERM, sighandler1);
    signal (SIGBUS, sighandler2);
}

void sighandler1(int sigNumber)
{
    ...
}

void sighandler2(int sigNumber)
{
    ...
}
```