High-level message passing

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Edited by Byoungyoung Lee and Pedro Fonseca

Admin

- Thank you for the feedback — definitely useful!
- Lab 2 grades released
- New lab coming out this week on virtual memory
- Piazza note to summarize the piazza discussion

Previous lecture

- Device management
- Clock and timer management
  - Timestamp counter and real time counter
  - Timed events: preemption and sleep
  - Operation timeout

Review of message passing design choices

- Potential synchronization:
  - Sender blocks
  - Receiver blocks
  - Neither blocks
  - Both block
- Message outstanding at a given time
  - Arbitrary number
  - Small, fixed

Review of message passing design choices

- Message storage:
  - Associated with sender
  - Associated with receiver
  - Independent of sender and receiver
- Destination
  - A specific process
  - Intermediate pickup point accessible to multiple processes

Review: Xinu-low level message passing

- Asynchronous (non-blocking) transmission
- Synchronous (blocking) reception
- Asynchronous message clear
- Message buffer holds one message
- Destination is a specific process
- API:
  - send(msg, pid)
  - msg = receive()
  - msg = recvclr()

Motivation for high-level message passing

- Permit synchronous message transfer
  - Block sender until receiver ready
  - Block receiver until sender ready
  - Example: data pipeline
- Make a message available to any process in a set
- Example:
  - Concurrent server
  - Set of processes that can handle requests
  - Next process in set handlers each incoming request
  - Allows short requests to be serviced quickly

Xinu high-level message passing

- Separate abstraction, unrelated to low-level message passing
- Independent from processes
- Allows arbitrary process to:
  - Send messages
  - Recieve messages
- Called *"port"*

Port details

- Port:
  - Created dynamically
  - Provides a synchronous interface using a producer and consumer semaphore per port
    - Receiver blocks when port empty
    - Sender blocks when port full
- At port creation:
  - Maximum number of messages is fixed and storage is allocated
  - Semaphores are created

Xinu port functions

- **ptinit()**:
  - Called once at startup
  - Initializes port system
- **ptcreate()**:
  - Creates a new port
  - Argument specifies number of messages
- **ptsend()**:
  - Sends a message to a port
- **ptrecv()**:
  - Retrieves a message from a port

- **ptreset()**:
  - Resets existing port
  - Disposes of existing messages
  - Allows waiting processes to continue
- **ptdelete()**:
  - Deletes an existing port
  - Disposes of existing messages
  - Allows blocked processes to continue

Xinu code

- ports.h
- ptinit.c
- ptsend.c
- ptcreate.c
- ptreceive.c

Port reset and deletion

- **ptreset** and **ptdelete**:
  - Dispose of existing messages, if the port contains any
  - Unblock processes that are waiting
    - To send
    - To receive
  - Semaphores are either reset or deleted
  - Processes are informed that an abnormal termination occurred

Disposing of messages

- Needed during reset and deletion
- Alternatives:
  - Fixed set of choices
  - Allow arbitrary processing
- Arbitrary processing
  - More general
  - Must allow to specify disposition function function as argument to **ptreset** or **ptdelete**
  - Disposition function is called for each existing message

Xinu code

- _ptclear

Semaphore reset and deletion

- Resetting or deleting a port will reset or delete semaphores
- If processes are blocked on the semaphore, they will become ready
- The rescheduling invariant means a higher priority process may execute
- Additional processes may attempt to use the port
- Consequence: we need to handle attempts to use the port during the reset

Handling resets

Three mechanisms for handling reset

- Accession numbers: assign a sequence number to a port
  - Increment sequence number when port created and deleted or reset
  - ptsend and ptrecv:
    - Record sequence number when operation begins **and** check sequence number after wait returns
  - If sequence number changed, port was reset, so operation should abort

Three mechanisms for handling reset

- New state for the port: assign each port a state variable
  - Value of the state variable
    - PTFREE if not in use
    - PTALLOC if in use
    - PTLIMBON if in transition
  - Have ptsend and ptrecv examine state variables
  - If state is PTLIMBO port is being reset or deleted and cannot be used

Three mechanisms for handling reset

- Deferred rescheduling: temporarily postpone scheduling decisions:
  - Call resched_cntl(DEFER_START) at start of reset or delete
  - Call resched_cntl(DEFER_STOP) after all operations are performed
- Q: Does deferred rescheduling introduce any potential problems?

How many IPC methods in Linux?

Linux IPC

1. Signals
2. Anonymous Pipes
3. Named Pipes or FIFOs
4. SysV Message Queues
5. POSIX Message Queues
6. SysV Shared memory
7. POSIX Shared memory
8. SysV semaphores
9. POSIX semaphores
10. FUTEX locks
11. File-backed and anonymous shared memory using mmap
12. UNIX Domain Sockets
13. Netlink Sockets
14. Network Sockets
15. Inotify mechanisms
16. FUSE subsystem
17. D-Bus subsystem

17 IPC mechanisms in Linux!!!

http://www.chandrashekar.info/articles/linux-system-programming/introduction-to-linux-ipc-mechanims.html

Signals

- Cheap communication form
- Typically used to notify
- Similar to interrupt (i.e., user-mode interrupts)
- Typically uses: Ctrl+C, timers, etc.

V2: Pipes

- Provide a mechanism for a process to send data to another
- A two-way data stream interfaced through standard input and output and is read character by character
- Anonymous (**mkfifo**) (fix: not “mkpipe”)
- Named (**mknod**)

Message queues

- Message queue: An anonymous data stream similar to the pipe, but stores and retrieves information in packets.
- In Linux it supports priorities

```
// SENDER

#include "fcntl.h"
#include "sys/stat.h"
#include "mqueue.h"
#include <stdio.h>
main() {

mqd_t q1;
char *buf1="hello1";
char *buf2="hello2";
char *buf3="hello3";
char *buf4="hello4";

q1 = mq_open("/test_q",O_CREAT|O_RDWR,0666,NULL);
if(q1 == -1) {
    printf("Error");
}

mq_send(q1,buf1,strlen(buf1),1);
mq_send(q1,buf2,strlen(buf2),2);
mq_send(q1,buf3,strlen(buf3),3);
mq_send(q1,buf4,strlen(buf4),4);

}
```

```
// RESULT

Priority= 4
 Message = hello4
Priority= 3
 Message = hello3
Priority= 2
 Message = hello2
Priority= 1
 Message = hello1
```

```
// RECEIVER

#include "fcntl.h"
#include "sys/stat.h"
#include "mqueue.h"
#include <stdio.h>
main() {

mqd_t q2;
char *buf;
struct mq_attr *attr1;
int prio;
attr1 = malloc(sizeof(struct mq_attr));

q2 = mq_open("/test_q",O_RDWR);

if(q2 == -1) {
    printf("Error");
}

buf = malloc(10*sizeof(char));
mq_getattr(q2, attr1);

mq_receive(q2,buf,attr1->mq_msgsize,&prio);
printf("Priority= %d",prio);
printf("\n Message = %s\n",buf);

mq_receive(q2,buf,attr1->mq_msgsize,&prio);
printf("Priority= %d",prio);
printf("\n Message = %s\n",buf);

mq_receive(q2,buf,attr1->mq_msgsize,&prio);
printf("Priority= %d",prio);
printf("\n Message = %s\n",buf);

mq_receive(q2,buf,attr1->mq_msgsize,&prio);
printf("Priority= %d",prio);
printf("\n Message = %s\n",buf);
}
```

https://tuxthink.blogspot.com/2012/03/ipc-messgae-queues-with-message.html

Other mechanisms

- Shared memory
- Sockets (network and unix domain)

Discussion

- What are the pros and cons of message passing vs. shared memory?
- How to implement priorities with Xinu ports?
- How to implement a web server?
- What is the impact of multiprocessors on IPC efficiency?

Summary

- Xinu offer low-level message passing
  - Only one message outstanding per process
- Xinu high-level message passing
  - Dynamically created ports
  - Number of messages and size fixed when port created
  - Arbitrary senders an receivers
  - Synchronous interface
- Port reset/deletion is tricky because of concurrency
  - Unblocked processes, new processes
  - Use three techniques to handle transition
- Linux IPC