CSCE 313 Introduction to Computer Systems

Instructor: Dr. Guofei Gu

http://courses.cse.tamu.edu/guofei/csce313

Inter-Process Communication (IPC)

- IPC concept & principle
- POSIX IPC
  - Message Queues
  - Shared Memory
  - Semaphore sets

- Reading: R&R, Ch 15
Interprocess Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need interprocess communication (IPC)

Fundamental Communication Models

(a)

(b)
Interprocess Communication - Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) - message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
Direct Communication

- Processes must name each other explicitly:
  - `send(P, message)` - send a message to process P
  - `receive(Q, message)` - receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  send(A, message) – send a message to mailbox A
  receive(A, message) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P₁, P₂, and P₃ share mailbox A
  - P₁, sends; P₂ and P₃ receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. **Zero capacity** - 0 messages
     Sender must wait for receiver (rendezvous)
  2. **Bounded capacity** - finite length of n messages
     Sender must wait if link full
  3. **Unbounded capacity** - infinite length
     Sender never waits
POSIX IPC: Overview

<table>
<thead>
<tr>
<th>primitive</th>
<th>POSIX function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message queues</td>
<td>msgget, msgctl, msgsnd/msgrecv</td>
<td>create or access control send/receive message</td>
</tr>
<tr>
<td>semaphores</td>
<td>semget, semctl, semop</td>
<td>create or access control wait or post operation</td>
</tr>
<tr>
<td>shared memory</td>
<td>shmget, shmtl, shmat/shmdt</td>
<td>create and init or access control attach to / detach from process</td>
</tr>
</tbody>
</table>

Accessing IPC resources from the shell: ipcs [-a]

**xxGET: It’s all about Naming!**

- **Condition variables, mutex locks:**
  - Based on a memory variable concept.
  - Does not work across memory spaces!!
- **Pipes**
  - Uses file descriptors
  - Works across memory spaces.
  - Relies on inheritance of file descriptors -> does not work for unrelated processes.
- **Named Pipes**
  - Uses file system as name space for pipe.
  - Works for unrelated processes.
  - Carry the overhead of the file system.
- **IPC Objects**
  - Use system-global integer keys to refer to objects.
IPC Object Creation: Message Queues

Object key identifies object across processes. It can be assigned as follows:

- Create some unknown key (IPC_PRIVATE)
- Pass explicit key (beware of collisions!)
- Use file system to consistently hash key (using ftok)

```c
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
/* create a message queue with given key and flags. */
```

Object id is similar to file descriptor. It can be inherited across fork calls.

```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
/* create a message queue with given key and flags. */
```

Operations on Message Queues

```c
#define PERMS (S_IRUSR | S_IWUSR)

int msqid;
if ((msqid = msgget(IPC_PRIVATE, PERMS)) == -1)
    perror("msgget failed");
```

```c
struct mymsg { /* user defined! */
    long msgtype; /* first field must be a long identifier */
    char mtext[1]; /* placeholder for message content */
}
```

```c
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg)
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
               long msgtyp, int msgflg);
```

<table>
<thead>
<tr>
<th>msgtyp</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>remove first message from queue</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>remove first message of type msgtyp from the queue</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>remove first message of lowest type that is less than or equal to absolute value of msgtyp</td>
</tr>
</tbody>
</table>
Operations on Message Queues (cont.)

```c
int msgctl(int msqid, int cmd, struct msgid_ds *buf)
```

<table>
<thead>
<tr>
<th>Cmd</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC_RMID</td>
<td>remove the message queue msqid and destroy the corresponding msqid_ds</td>
</tr>
<tr>
<td>IPC_SET</td>
<td>Set members of the msqid_ds data structure from buf</td>
</tr>
<tr>
<td>IPC_STAT</td>
<td>Copy members of the msqid_ds data structure into buf</td>
</tr>
</tbody>
</table>

Message Queue Program Example

- Program 15.9 msgqueuelog.c (R&R ch15)
  - Utility functions that access and output to a message queue

- Program 15.10 msgqueuesave.c (R&R ch15)
  - A program that copies messages from a message queue to standard output
POSIX Shared Memory

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

Ok, we have created a shared-memory segment. Now what?

void *shmat(int shmid, const void *shmaddr, int shmflg);

Use POSIX Shared Memory

- Process first creates shared memory segment
  segment id = shmget(IPC PRIVATE, size, S_IRUSR | S_IWUSR);
- Process wanting access to that shared memory must attach to it
  shared memory = (char *) shmat(id, NULL, 0);
- Now the process could write to the shared memory
  sprintf(shared memory, "Writing to shared memory");
- When done a process can detach the shared memory from its address space
  shmdt(shared memory);
**Shared Memory Program Example**

- **Program 15.5 monitorshared.c**
  - Monitor two file descriptors and keep information in shared memory. The parent waits for the child, to ensure mutual exclusion

- **Program 15.6 sharedmemsum.c**
  - A function that keeps a synchronized sum and count in shared memory

- **Program 15.7 showshared.c**
  - Display the shared count and sum when it receives a SIGUSR1 signal

---

**POSIX Semaphore Sets**

```c
#include <sys/sem.h>

#include <sys/sem.h>
define PERMS (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
define SET_SIZE 2

int main (int argc, char * argv[]) {
  key_t mykey;
  int semid;

  mykey = ftok(argv[1], atoi(argv[2]));
  semid = semget(mykey, SET_SIZE, PERMS | IPC_CREAT)
  return 0;
}
```

# Semaphore Set Control

```c
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETVAL</td>
<td>set value of a specific semaphore element to arg.val</td>
</tr>
<tr>
<td>SETALL</td>
<td>set values of semaphore set from arg.array</td>
</tr>
<tr>
<td>GETVAL</td>
<td>return value of specific semaphore element</td>
</tr>
<tr>
<td>GETALL</td>
<td>return values of the semaphore set in arg.array</td>
</tr>
<tr>
<td>GETPID</td>
<td>return process id of last process to manipulate element</td>
</tr>
<tr>
<td>GETNCNT</td>
<td>return number of processes waiting for element to increment</td>
</tr>
<tr>
<td>GETZCNT</td>
<td>return number of processes waiting for element to become 0</td>
</tr>
</tbody>
</table>

etc....

# Semaphore Set Operations

```c
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
/* The operations are defined in the array pointed to by 'sops'. */
```

<table>
<thead>
<tr>
<th>sem_op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>add the value to the semaphore element and awaken all processes that are waiting for element to increase</td>
</tr>
<tr>
<td>(sem_op == 0) &amp;&amp; (semval != 0)</td>
<td>block calling process (waiting for 0) and increment count of processes waiting for 0.</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>add sem_op value to semaphore element value provided that result would not be negative. If result negative, block process on event that semaphore value increases. If result == 0, wake processes waiting for 0.</td>
</tr>
</tbody>
</table>
Semaphore Operations: Example

/* pseudo code */
struct sembuf get_tapes[2];
struct sembuf release_tapes[2];

setsembuf(&(get_tapes[0]), 0, -1, 0);
setsembuf(&(get_tapes[1]), 1, -1, 0);
setsembuf(&(release_tapes[0]), 0, +1, 0);
setsembuf(&(release_tapes[1]), 1, +1, 0);

/* Process 1: */ semop(S, get_tapes, 1);
<use Tape 0>
semop(S, release_tapes, 1);

/* Process 2: */ semop(S, get_tapes, 2);
<use both tapes 0 and 1>
semop(S, release_tapes, 2);

What if there are three tapes and process to use both tape 1 and 2?
(Check example 15.10 in the book.)

Semaphore Operations: Another Example

int main(int argc, char * argv[]) {
    int semid;
    struct sembuf sem_signal[1];
    struct sembuf sem_wait[1];

    semid = semget(IPC_PRIVATE, 1, PERMS);
    setsembuf(sem_wait, 0, -1, 0);
    setsembuf(sem_signal, 0, 1, 0);
    init_element(semid, 0, 1); /* Set value of element 0 to 1 */

    for (int i = 1; i < n; i++) if fork() break;

    semop(semid, sem_wait, 1); /* enter critical section */
    /* in critical section */
    semop(semid, sem_signal, 1); /* leave critical section */
    /* in remainder section */
    return 0;
}