

Chapter 14: Interprocess Communication





Plans

- This week: Chapter 14
- Next week:
 - Networked IPC
 - Other?
- Last week
 - Something
 - Review



Introduction

- Interprocess Communication (IPC) enables processes to communicate with each other to share information
 - **Pipes (half duplex)**
 - **FIFOs (named pipes)**
 - Stream pipes (full duplex)
 - Named stream pipes
 - Message queues
 - **Semaphores**
 - **Shared Memory**
 - **Sockets**
 - Streams



Pipes

- Oldest (and perhaps simplest) form of UNIX IPC
- Half duplex
 - Data flows in only one direction
- Only usable between processes with a common ancestor
 - Usually parent-child
 - Also child-child



Pipes (cont.)

- `#include <unistd.h>`
- `int pipe(int fildes[2]);`
- `fildes[0]` is open for reading and `fildes[1]` is open for writing
- The output of `fildes[1]` is the input for `fildes[0]`



Understanding Pipes

- Within a process
 - Writes to `filides[1]` can be read on `filides[0]`
 - Not very useful
- Between processes
 - After a `fork()`
 - Writes to `filides[1]` by one process can be read on `filides[0]` by the other



Understanding Pipes (cont.)

- Even more useful: two pipes, `fildes_a` and `fildes_b`
- After a `fork()`
- Writes to `fildes_a[1]` by one process can be read on `fildes_a[0]` by the other, and
- Writes to `fildes_b[1]` by that process can be read on `fildes_b[0]` by the first process



Using Pipes

- Usually, the unused end of the pipe is closed by the process
 - If process A is writing and process B is reading, then process A would close `filides[0]` and process B would close `filides[1]`
- Reading from a pipe whose write end has been closed returns 0 (end of file)
- Writing to a pipe whose read end has been closed generates SIGPIPE
- `PIPE_BUF` specifies kernel pipe buffer size



Example

```
int main(void) {
    int n, fd[2];
    pid_t pid;
    char line[maxline];

    if(pipe(fd) < 0) err_sys("pipe error");
    if( (pid = fork()) < 0) err_sys("fork
error"); else if(pid > 0) {
        close(fd[0]);
        write(fd[1], "hello\n", 6);
    } else {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
}
```

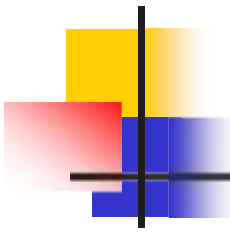


Example: Piping output to child process' input

```
int fd[2];
pid_t pid;

pipe(fd);
pid = fork();

if(pid == 0) {
    dup2(fd[0], STDIN_FILENO);
    exec(<whatever>);
}
```



Using Pipes for synchronization and communication

- Once you have a pipe or pair of pipes set up, you can use it/them to
 - Signal events (one pipe)
 - Wait for a message
 - Synchronize (one or two pipes)
 - Wait for a message or set of messages
 - You send me a message when you are ready, then I'll send you a message when I am ready
 - Communicate (one or two pipes)
 - Send messages back and forth



popen()

- `#include <stdio.h>`
- `FILE *popen(const char *cmdstring, const char *type);`
- Encapsulates a lot of system calls
 - Creates a pipe
 - Forks
 - Sets up pipe between parent and child (type specifies direction)
 - Closes unused ends of pipes
 - Turns pipes into FILE pointers for use with STDIO functions (fread, fwrite, printf, scanf, etc.)
 - Execs shell to run cmdstring on child



popen() and pclose()

- Popen() details
 - Directs output/input to stdin/stdout
 - "r" -> parent reads, "w" -> parent writes
- `int pclose(FILE *fp);`
- Closes the STDIO stream
- Waits for command to terminate
- Returns termination status of shell



Assignment

- Simulated audio player with shared memory and semaphores
- We will discuss this at the end of class today



FIFOs

- First: Coprocesses – Nothing more than a process whose input and output are both redirected from another process
- FIFOs – named pipes
- With regular pipes, only processes with a common ancestor can communicate
- With FIFOs, any two processes can communicate
- Creating and opening a FIFO is just like creating and opening a file



FIFO details

- `#include <sys/types.h>`
- `#include <sys/stat.h>`
- `int mkfifo(const char *pathname, mode_t mode);`
 - The mode argument is just like in `open()`
- Can be opened just like a file
- When opened, `O_NONBLOCK` bit is important
 - Not specified: `open()` for reading blocks until the FIFO is opened by a writer (same for writing)
 - Specified: `open()` returns immediately, but returns an error if opened for writing and no reader exists



Example: Using FIFOs to Duplicate Output Streams

- Send program 1's output to both program2 and program3 (p. 447)
- `mkfifo fifo1`
- `prog3 < fifo1 &`
- `prog1 < infile | tee fifo1 | prog2`



Example: Client-Server Communication Using FIFOs

- Server contacted by multiple clients (p.448)
- Server creates a FIFO in a well-known place
 - And opens it read/write
- Clients send requests on this FIFO
 - Must be < PIP_BUF bytes
- Issue: How to respond to clients
- Solution: Clients send PID, server creates per-client FIFOs for responses



System V IPC

- IPC structures for message queues, semaphores, and shared memory segments
- Each structure is represented by an identifier
 - The identifier specifies which IPC object we are using
 - The identifier is returned when the corresponding structure is created with `msgget()`, `semget()`, or `shmget()`
- Whenever an IPC structure is created, a key must be specified
 - Matching keys refer to matching objects
 - This is how two processes can coordinate to use a single IPC mechanism to communicate



Rendezvousing with IPC Structures

- Process 1 can specify a key of `IPC_PRIVATE`
 - This creates a unique IPC structure
 - Process 1 then stores the IPC structure somewhere that Process 2 can read
- Process 1 and Process 2 can agree on a key ahead of time
- Process 1 and Process 2 can agree on a pathname and project ID ahead of time and use `ftok` to generate a unique key



IPC Permissions

- System V associates an ipc_perm structure with each IPC structure:

```
struct ipc_perm {  
    uid_t uid; // owner's eff. user ID  
    gid_t gid; // owner's eff. group ID  
    uid_t cuid; // creator's eff. user ID  
    gid_t cgid; // creator's eff. group ID  
    mode_t mode; // access modes  
    ulong seq; // slot usage sequence  
    nbr key_t key; // key  
}
```



Issues w/System V IPC

- They are equivalent to global variables
 - They live beyond the processes that create them
- They don't use file descriptors
 - Can't be named in the file system
 - Can't use `select()` and `poll()`



Message Queues

- Linked list of messages stored in the kernel
- Identifier by a message queue identifier
- Created or opened with `msgget()`
- Messages are added to the queue with `msgsnd()`
 - Specifies type, length, and data of msg
- Messages are read with `msgrcv()`
 - Can be fetched based on type



msqid_ds

- Each message queue has a msqid_ds data structure

```
struct msqid_ds {  
    struct ipc_perm msg_perm;           //  
    struct msg *msg_first;              // ptr to first msg on queue  
    struct msg *msg_last;              // ptr to last msg on queue  
    ulong msg_cbytes;                  // current # bytes on queue  
    ulong msg_qnum                      // # msgs on queue  
    ulong msg_qbytes                   // max # bytes on queue  
    pid_t msg_lspid;                   // pid of last msgsnd()  
    pid_t msg_lrpid;                   // pid of last msgrcv()  
    time_t msg_srttime;                // last msgsnd() time  
    time_t msg_rtime;                  // last msgrcv() time  
    time_t msg_ctime;                  // last change time  
};
```




Limits

- MSGMAX – size of largest message
 - Usually 2048
- MSGMNB – Max size in bytes of queue
 - Usually 4096
- MSGMNI – Max # of msg queues
 - Usually 50
- MSGTQL – Max # of messages, systemwide
 - Usually 40



msgget()

- `#include <sys/types.h>`
- `#include <sys/ipc.h>`
- `#include <sys/msg.h>`
- `int msgget(key_t key, int flag);`
 - flag specifies mode bits
 - returns msg queue ID



msgctl()

- `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`
 - Depends on cmd
 - `IPC_STAT` – fills buf with `msqid_ds`
 - `IPC_SET` – sets various fields of `msqid_ds`
 - `IPC_RMID` – removes message queue from system



msgsnd()

- `int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);`
- `ptr` points to the data of the message, with type:
 - `struct mtypes {`
 - `long mtype;`
 - `char mtext[512];`
 - `}`



msgrcv()

- `int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);`
- `type == 0`: return the first message
- `type > 0`: return first message with specified type
- `type < 0`: return first message whose type is lowest with value `<=` specified type



Semaphores

- Create semaphore: `semget()`
- Test value: `semop()`
 - If > 0 , decrement and continue
 - if < 0 , sleep till > 0
- Increment value: `semop()`



semid_ds

```
struct semid_ds {
    struct ipc_perm; //
    struct sem *sem_base; // ptr to 1st sem in set
    ushort sem_nsems; // # of sems in set
    time_t sem_otime; // last-semop() time
    time_t sem_ctime; // last-change time
};
struct sem {
    ushort semval; // semaphore value
    pid_t sempid; // pid for last operation
    ushort semncnt; // # of procs awaiting semval > curval
    ushort semzcnt; // # of procs awaiting semval = 0
}
```



semget()

- `#include <sys/types.h>`
- `#include <sys/ipc.h>`
- `#include <sys/sem.h>`
- `int semget(key_t key, int nsems, int flag);`
 - `nsems` is the number of semaphores in the set



semctl()

- `int semctl(int semid, int semnum, int cmd, union semun arg);`
- `union semun {`
 - `int val; // for setval`
 - `struct semid_ds *buf; // for IPC_STAT and IPC_SET`
 - `ushort *array; // for GETALL and SETALL`
- `}`
- `IPC_STAT`: get the `semid_ds`
- `IPC_SET`: set `semid_ds` fields
- `IPC_RMID`: remove semaphore
- `GETVAL`: return the value of `semval` for `semnum`
- `SETVAL`: set the value of `semval` for `semnum`
- `GETPID`: return the value of `sempid` for `semnum`
- `GETNCNT`: return the value of `semcnt` for `semnum`
- `GETZCNT`: return the value of `semzcnt` for `semnum`
- `GETALL`: fetch all semaphore values in the set
- `SETALL`: set all semaphore values in the set



semop()

- `int semop(int semid, struct sembuf semoparray[], size_t nops);`
- `struct sembuf {`
 - `ushort sem_num; // member #`
 - `short sem_op; // operation`
 - `short sem_flg; // IPC_NOWAIT, SEM_UNDO`
 - `};`
- `sem_op > 0`: `sem_op` is added to sems value
- `sem_op < 0`: reduce sem by `sem_op` (if possible), otherwise block depending upon `IPC_NOWAIT` value
- `sem_op == 0`: wait until value becomes 0
- `semop` is atomic



Shared Memory

- See p. 464