Inter-process Communication (IPC)

Tyler Bletsch
Duke University

Slides are adapted from Brian Rogers (Duke)
Recall Process vs. Thread

A process is –
- Execution context
  - PC, SP, Regs
- Code
- Data
- Stack

A thread is –
- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Registers
Cooperation

- Two or more threads or processes execute concurrently
- Sometimes cooperate to perform a task
  - Sometimes independent; not relevant for IPC discussion
- How do they communicate?
  - Threads of the same process: Shared Memory
    - Recall they share a process context
      - Code, static data, *heap*
    - Can read and write the same memory
      - variables, arrays, structures, etc.
  - What about threads of different processes?
    - They do not have access to each other’s memory
Models for IPC

• Shared Memory
  • E.g. what we’ve discussed for threads of same process
  • Also possible across processes
    • E.g. memory mapped files (mmap)

• Message Passing
  • Use the OS as an intermediary
  • E.g. Files, Pipes, FIFOs, Messages, Signals
Models for IPC

**Shared Memory**
- Thread A
- e.g. shared heap
- Thread B
- OS Kernel Space

**Message Passing**
- Thread A
- Thread B
- OS Kernel Space

- Write value
- Read value

- Write value
- Read value
Shared Memory vs. Message Passing

- **Shared Memory**
  - **Advantages**
    - Fast
    - Easy to share data (nothing special to set up)
  - **Disadvantages**
    - Need synchronization! Can be tricky to eliminate race conditions

- **Message Passing**
  - **Advantages**
    - Trust not required between sender / receiver (receiver can verify)
    - Set of shared data is explicit
    - Is synchronization needed?
  - **Disadvantages**
    - Explicit programming support needed to share data
    - Performance overhead (e.g. to copy messages through OS space)
Shared Memory Across Processes

- Different OSes have different APIs for this

**UNIX**
- System V shared memory (shmget)
  - Allows sharing between arbitrary processes
- Shared mappings (mmap on a file)
  - Different forms for only related processors or unrelated processes (via filesystem interaction)
- POSIX shared memory (shm_open + mmap)
  - Sharing between arbitrary processes; no overhead of filesystem I/O

- Still requires synchronization!
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

- Creates new mapping in virtual address space of caller
  - addr: starting address for mapping (or NULL to let kernel decide)
  - length: # bytes to map starting at “offset” of the file
  - prot: desired memory protection of the mapping
    - PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
  - flags: are updates to mapping are visible to other processes?
    - MAP_SHARED, MAP_PRIVATE
    - Other flags can be added, e.g. MAP_ANON (more later)
  - fd: file descriptor for open file
    - Can close the “fd” file after calling mmap()
  - Return value is the address where the mapping was made
mmap operation

- Kernel takes an open file (given by FD)
- Maps that into process address space
  - In unallocated space between stack & heap regions
  - Thus also maps file into physical memory
  - Creates one-to-one correspondence between a memory address and a word in the file
- Useful even apart from the context of IPC
  - Allows programmer to read/write file contents without read(), write() system calls
- Multiple (even non-related) processes can share mem
  - They open & mmap the same file
munmap

```c
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

- Removes mapping from process address space
  - `addr`: address of the mapping
  - `length`: # bytes in mapped region

```c
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- Flushes file contents in memory back out to disk
  - `addr`: address of the mapping
  - `length`: # bytes in mapped region
  - `flags`: control when the update happens
Synchronization

- Semaphores

```c
#include <fcntl.h>  /* For O_* constants */
#include <sys/stat.h>  /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, mode_t mode, int value);

or

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```c
sem_t *mutex;
mutex = sem_open("my_sem_name", O_CREAT | O_EXCL,
                 MAP_SHARED, 1);

sem_wait(mutex);
//Critical section
sem_post(mutex);
```
Example

- Show code & run in class
  - mmap_basic and mmap_basic2
• This required some work
  • Create file in file system
  • Open the file & initialize it (e.g. with 0’s)

• There is a better way if just sharing mem across a fork()
  • Anonymous memory mapping

• Use mmap flags of MAP_SHARED | MAP_ANON
  • File descriptor will be ignored (also offset)
  • Memory initialized to 0
  • Alternative approach: open /dev/zero & mmap it

• Can anonymous approach work across non-related processes?
Message Passing

- Messages between processes, facilitated by OS
- Several approaches:
  - Files
    - Can open the same file between processes
    - Communicate by reading and writing info from the file
    - Can be difficult to coordinate
  - Pipes
  - FIFOs
  - Messages (message passing)
#include <unistd.h>
int pipe(int pipefd[2]);

- Creates a unidirectional channel (pipe)
- Can be used for IPC between processes / threads
- Returns 2 file descriptors
  - pipefd[0] is the read end
  - pipefd[1] is the write end
- Kernel support
  - Data written to write end is buffered by kernel until read
  - Data is read in same order as it was written
  - No synchronization needed (kernel provides this)
  - Must be related processes (e.g. children of same parent)
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 1024

int main(int argc, char *argv[]) {
    int pipefd[2];
    char data_buffer[N];

    pipe(pipefd);
    int id = fork();
    if (id == 0) { //child
        write(pipefd[1], "hello", 6);
    } else {
        read(pipefd[0], data_buffer, 6);
        printf("Received data: %s\n", data_buffer);
    }

    return 0;
}
```
More Complex Uses of Pipes

- Can use pipes to coordinate processes
- For example, chain output of one process to input of next
  - E.g. command pipes in UNIX shell!
- Requires 1 additional (very useful) piece
  ```c
  #include <unistd.h>
  int dup2(int oldfd, int newfd);
  ```
- Creates a copy of an open file descriptor into a new one
  - After closing the new file descriptor if it was open
UNIX Pipes Example

- Show code & run in class
  - pipe_basic
UNIX FIFOs

- Similar to a pipe
  - Also called a “named pipe”
- Persist beyond lifetime of the processes that create them
- Exist as a file in the file system
  ```
  #include <sys/types.h>
  #include <sys/stat.h>
  int mkfifo(const char *pathname, mode_t mode);
  ```
- `pathname` points to the file
- Mode specifies the FIFO’s permissions (similar to a file)
UNIX FIFOs (2)

- After FIFO is created, processes must open it
  - By default, first open blocks until a second process also opens
  - One process opens for reading and the other process for writing
- Since FIFOs persist, they can be re-used
- No synchronization needed (like pipes, OS handles it)
Playing with FIFOs on the shell

- Can create a FIFO using `mkfifo` command
  - Note: need to be in a UNIX-style filesystem to do this. Your shared Duke home directory is a Windows-style filesystem, so try this in `/tmp` if using the Duke Linux environment.

- Can read/write FIFO using normal commands.
  - “tail -f” will monitor a file (or FIFO) over time.
Multiple Producers

• Multiple producers problem:
  • What if >1 producers and 1 consumer
  • Producers are performing write(…)
  • Consumer is performing (blocking) read(…)
  • What if consumer is blocked, but other IPC channels have data?

• Would like to be notified if one channel is ready
#include <sys/select.h>

```c
int select(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
```

- nfds = number of file descriptors to monitor
- readfds, writefds, exceptfds are bit vectors of file descriptors to check
- timeout is a maximum time to wait

- Macros are available to work with bit sets:
  - FD_ZERO(&fds), FD_SET(n, &fds), FD_CLEAR(n, &fds)
  - int FD_ISSET(n, &fds); //useful after select() returns
Poll

#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

- nfds = number of file descriptors to monitor
- fds is an array of descriptor structures
  - File descriptors, desired events, returned events
- timeout is a maximum time to wait
- Returns number of descriptors with events