Slide 1

Week 4

COMP3231 Operating Systems

2005/S2

① Concurreny Control
  - Interprocess Communication (IPC)
  - Deadlocks

② Memory Management

Slide 2

IPC Mechanisms

Several different mechanisms can be used for communication

① Shared memory: can be used to exchange information, but synchronisation issues remain
  - threads: run in common memory space
  - `mmap()` system call

② File system
  - normal files
  - pipes (FIFOs)
  - sockets

③ Message passing
  - more abstract communication mechanism

Slide 3

A closer look at `mmap` and Pipes

Memory mapped files:

➜ Processes can share files to communicate

➜ `mmap` maps by default a file into memory

Type of `mmap`:

```
void  *mmap (void      *addr,   /* dst address for map   */
             size_t    len,     /* length of data to map */
             int       prot,    /* protection            */
             int       flags,   /* misc flags            */
             int       fildes,  /* file descriptor       */
             off_t     offset)  /* offset into file      */
```

By using `mmap` can be used to simulate "shared memory".

Slide 4

Example: simplified, no error checking!

```
int main () {
  char   *data, *fname = "foo";
  int    fd;
  struct stat sbuf;
  fd = open (fname, O_RDONLY);
  stat (fname, &sbuf);

  data = mmap ((caddr_t) 0,      let the system choose the dest.
               sbuf.st_size,     will be rounded to pagesize
               PROT_READ,        mem. area will be read only
               MAP_SHARED,       shared: changes visible to all proc
               fd,               file desc. of previously opened file
               0);               offset

  printf ("Test mmap:...%c", data[0]);}
```

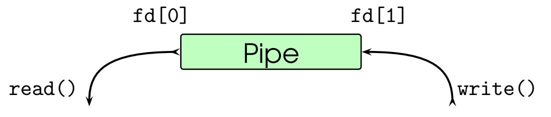
Slide 5

Pipes

What is a file descriptor?

- ➜ `fopen`, `fwrite`, etc are implemented in terms of systems calls like `open`, `write`
- ➜ `fopen` returns a `FILE` pointer
- ➜ `open` returns a file descriptor (`int`)
- ➜ `stdin`, `stdout`, `stderr` are file descriptors

`pipe()` returns a pair of file descriptors:



Slide 6

Example: (no error checking)

```
int  fd[2];
char buf[20];

pipe (fd);

write (fd[1], "test1", 6);
write (fd[1], "test2", 6);
read  (fd[0], buf, 6);
printf ("read %s from pipe \n", buf);
```

Usage:

- ➜ pipes can be used in combination with `fork`
- ➜ named pipes: `mknod`

```
mknod ("testFIFO", S_FIFO | 0644, 0);
```


Slide 7

Assignment 1

Implementing (a sort of) Pipes

```
void pipe_create   (pipe_in **p_in, pipe_out **p_out);
void pipe_destroy_in    (pipe_in  *p);
void pipe_destroy_out   (pipe_out *p);
void pipe_read     (pipe_out *p, void *dest,  int n_bytes);
void pipe_write    (pipe_in  *p, void *src, int n_bytes);
```

Assignment 1:

- ➜ Will be out in the next few days
- ➜ Anybody not in a group on Wednesday will automatically have a partner assigned

Slide 8

Sockets

What is a socket?

- ➜ two-way communication pipe
- ➜ can be used to communicate in a wide variety of domains (e.g., internet)

Communication between Processes:

Sockets are also a file in the Unix file system, but offer a different interface

- ➜ `socket()`, `bind()`, `receive()` instead of `open()`, `read()` and `write()` (`read()` and `write()` are actually also available on sockets)
- ➜ typically used in client/server style programs

Slide 9

Server:

① create socket:

```
s = socket(AF_UNIX, SOCK_STREAM)
```

② bind socket to local address:

```
bind (s, <socket name>)
```

③ listen for incoming connections:

```
listen (s, <max size of incomming connection queue>)
```

④ main server loop:

① accept connection

```
s2 = accept (s, &<remote socket name>)
```

② receive/send

```
recv (s2, &<request>)
send (s2, <answer>)
```

③ close

```
close (s2)
```

Slide 10

Client:

① create socket:

```
s = socket(AF_UNIX, SOCK_STREAM)
```

② connect:

```
connect (s, <socket name>)
```

③ send/receive:

```
send (s, <request>)
recv (s, &<answer>)
```

④ close:

```
close (s)
```


Slide 11

MESSAGE-PASSING IPC

Primitives:

- ➜ Sending a message: `send(dest, msg)`
- ➜ Receiving a message: `receive(source, &msg)`

Different message passing styles:

- ➜ synchronisation: blocking (synchronous) vs. non-blocking (asynchronous)
  - blocking send, blocking receive
  - non-blocking send, blocking receive
  - non-blocking send, non-blocking receive
- ➜ addressing: direct vs. indirect
  - identifier of destination process
  - message to shared data structure (mailbox, port), one-to-one, one-to-many, many-to-many
- ➜ message format: depends on objectives, single computer vs. distributed system, fixed vs. variable-length messages

Slide 12

IPC Implementation Issues:

- ➜ Security & Safety
  - messages may be lost
  - authentication
- ➜ How are links established?
  - automatically
  - have to be set up explicitely
- ➜ What is the capacity of a link?
- ➜ Is the message format fixed or variable?
- ➜ Is a link uni-directional or bi-directional?

Slide 13

IPC: Direct Communication

Processes must name each other explicitly

- ➜ `send(pid, &msg)`
- ➜ `receive(pid, &msg)` — sometimes id of sender cannot be anticipated

Properties of communication link :

- ➜ links established automatically
- ➜ link associated with pair of processes
- ➜ exactly one link between each pair
- ➜ link may be uni- or bi-directional

Slide 14

Indirect Communication

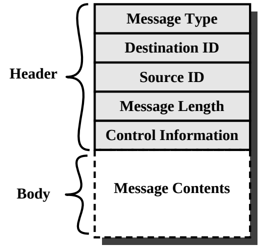
- ➜ Messages go via mailboxes (aka. ports)
  - each port has unique ID
  - communication requires sharing of a port
- ➜ Properties of communication link:
  - links established if processes share a port
  - link may be associated with many processes
  - each pair may share many links
  - link (port) may be uni- or bi-directional
- ➜ Operations: create, delete, send, receive

Slide 15

General Message Format

The format of a message depends on

- ➜ objectives of message facility
- ➜ local or distributed
- ➜ security and safety requirements



Slide 16

Message Buffering:

- ➜ Associate message buffer with link:
  - Zero capacity 0 messages
    - sender blocks until receive (rendezvous)
  - Bounded capacity: finite # messages
    - if full sender blocks or fails
  - Unbounded capacity: infinite # messages
    - sender never blocks

IPC Exception Conditions:

- ➜ Partner process terminated
- ➜ Partner uncommunicative (protocol failure)
- ➜ Message buffer overflow
- ➜ Message lost
- ➜ Message scrambled