THE UNIVERSITY *of* EDINBURGH
informatics

Operating Systems (INFR10079)

2022/2023 Semester 2

Processes

(Interprocess Communication)

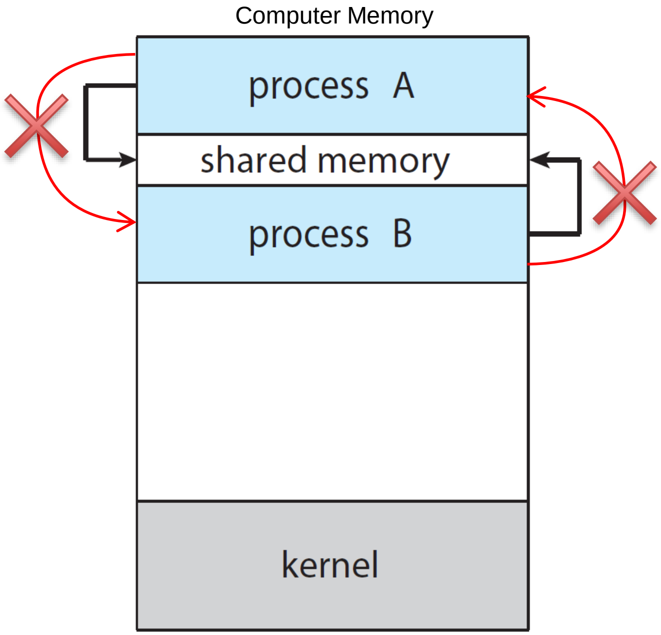
abarbala@inf.ed.ac.uk

Chapter 3.4, 3.5, 3.6, 3.7, 3.8, excluding: 3.7.2, 3.7.3, 3.8.2.1
4.6.2, 20.9.1, C3.3

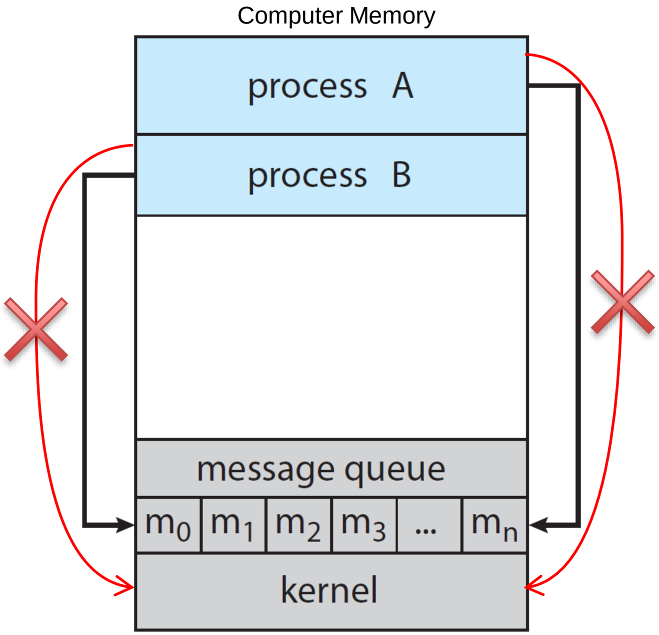
Interprocess Communication

- **Independent** processes
- **Cooperating** processes
  - Information **sharing**
    - More applications interested in the same information
  - Computation **speedup**
    - To exploit parallel hardware
  - **Modularity**
    - Reusability of components
- Require **interprocess communication** (IPC) mechanism to send and receive data
  - Shared memory
  - Message passing

Interprocess Communication

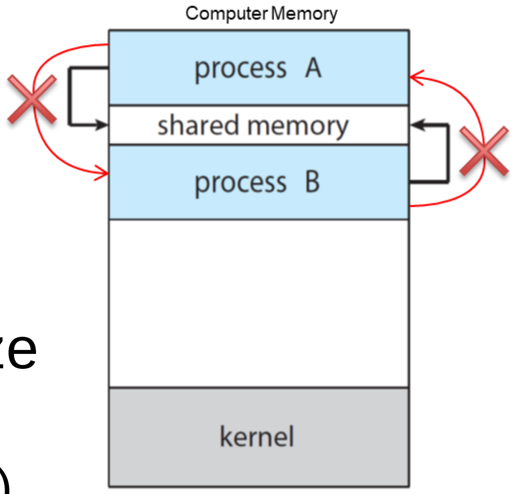


Shared memory



Message passing

Shared Memory



- Allow processes to communicate and synchronize
  - – **Sharing** part of address space
  - – **OS doesn’t mediates** communication (no overhead)
    - Usually, **OS prevents** processes to access each other memory
    - Processes should agree (have permission) to void this restriction

- Data
  - – Format **decided by application**
  - – Direct access (not mediated by the OS) – **very fast**
  - – Application programmer fully manages the data transfer – **not trivial**

- Possible use cases
  - – Passing of large (single) objects (C&P, image, table, etc.)
  - – Notification variable

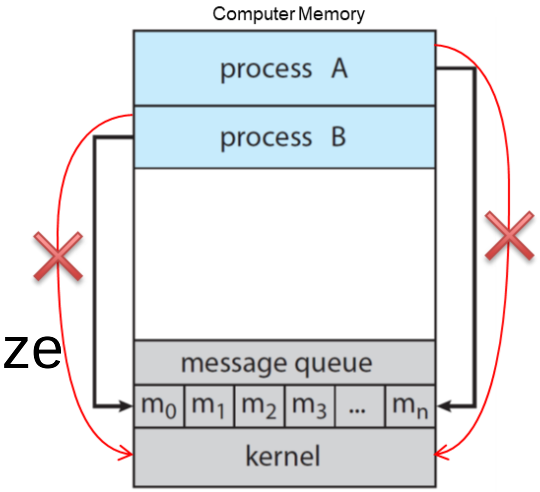
Example of Shared Memory Code (POSIX)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */
  const char *message 0 = "Hello"; /* written to shared memory */
  const char *message 1 = "World!"; /* written to shared memory */
  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* create the shared memory object */
  fd = shm_open(name,O CREAT | O RDWR,0666);
  /* configure the size of the shared memory object */
  ftruncate(fd, SIZE);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* write to the shared memory object */
  sprintf(ptr,"%s",message 0);
  ptr += strlen(message 0);
  sprintf(ptr,"%s",message 1);
  ptr += strlen(message 1);

  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */


  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* open the shared memory object */
  fd = shm_open(name, O RDONLY, 0666);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* read from the shared memory object */
  printf("%s",(char *)ptr);
  /* remove the shared memory object */
  shm_unlink(name);

  return 0;
}
```

Try at home!

Message Passing



- Allow processes to communicate and synchronize
  - **Without sharing** part of address space
  - **OS mediates** communication (introduce overhead)
- Processes on the **same** machine and **among different** inter-networked machines
  - Not possible with shared memory

- Message-passing facility provides at least two operations
  - **send(message)**
  - **receive(message)**
- Communication link
  - Several **implementation tradeoffs**, e.g., messages size
    - Fixed
    - Variable

Message Passing – Naming

Naming is a problem also in shared memory

- Communicating processes must **refer to each other**
- **Direct** communication
  - **Symmetric:** Explicit name of sender and receiver
    - `send(P, message)` – send message to process P
    - `receive(Q, message)` – receive message from process Q
  - **Asymmetric:** Explicit at least one end
    - `send(p, message)` – send to p
    - `receive(id, message)` – receive from any process, sender saved in id
- **Indirect** communication
  - No need to know/explicit in advance sender and/or receiver
  - Mailboxes (e.g., POSIX Mailbox)
    - `send(A, message)` – into mailbox A
    - `receive(A, message)` – from mailbox A
  - A mailbox can be accessed by more than two processes
    - Between processes multiple mailboxes may exist

Message Passing – Synchronization

- Different **design options** to implement `send()/receive()`
  - **Blocking** or **synchronous**
  - **Nonblocking** or **asynchronous**

- Different **combinations** may be offered
  - Blocking send
  - Nonblocking send
  - Blocking receive
  - Nonblocking receive

- **Rendezvous**
  - When both send() and receive() **are blocking**
  - Simple solution to the consumer-producer problem***

Message Passing – Buffering

- Message exchanged reside in **temporary buffers/queues**
  - **Zero** capacity (no buffering)
    - No message waiting in it
    - Sender must block until the recipient receives the message
  - **Bounded** capacity
    - Finite length ***n***; at most ***n*** messages can reside in it
    - If the queue is not full when a new message is sent, that is placed in the queue and the sender can continue execution
    - If the link is full, the sender has to wait
  - **Unbounded** capacity
    - Infinite queue
    - The sender never blocks

Example of message passing: Pipes

- A pipe acts as a **conduit** allowing two processes to communicate
  - One way data flow
- **Anonymous** (ordinary)
  - Between parent and child
- **Named**
  - between any pair of processes
  - In UNIX, FIFO

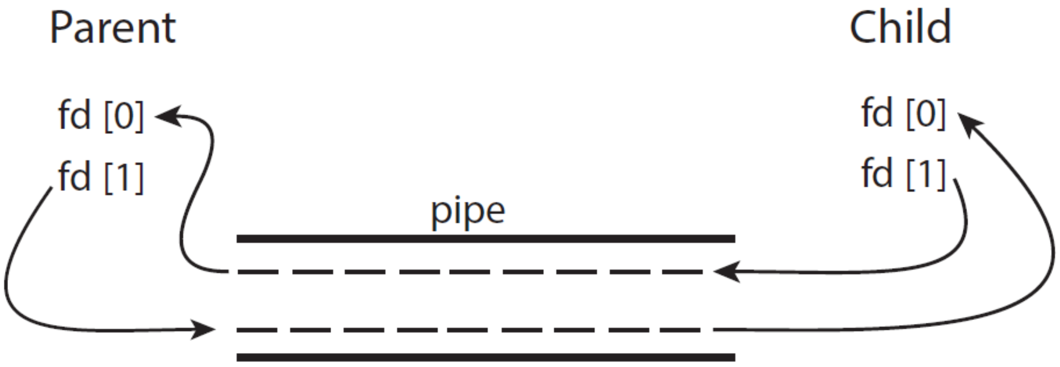


Figure 3.20 File descriptors for an ordinary pipe.

Example of Pipe Code (POSIX)

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define READ_END 0
#define WRITE_END 1
int main(void) {
    char write msg[256] = "Greetings";
    char read msg[256];
    int fd[2];
    pid t pid;

    if (pipe(fd) == -1) {/* create the pipe */
        fprintf(stderr,"Pipe failed");
        return 1;
    }
    pid = fork(); /* fork a child process */
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    if (pid > 0) { /* parent process */
        close(fd[READ_END]); /* close the unused end of the pipe */
        write(fd[WRITE_END], write msg, strlen(write msg)+1); /* write to the pipe */
        close(fd[WRITE_END]); /* close the write end of the pipe */
    }
    else { /* child process */
        close(fd[WRITE_END]); /* close the unused end of the pipe */
        read(fd[READ_END], read msg, 256); /* read from the pipe */
        printf("read %s",read msg);
        close(fd[READ_END]); /* close the read end of the pipe */
    }
    return 0;
}
```

Try at home!

Client-Server Communication

- **Sockets** Abstraction
  - Endpoint for communication
  - Identified by an **IP address** concatenated with a **port number**
  - Servers implementing specific services (SSH, FTP, and HTTP) listen to well-known ports
    - an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80

- **Remote Procedure Call** (RPC)
  - Abstract the **procedure-call mechanism**
  - For use between systems with network connections
  - **Similar** in many respects **to the IPC**
    - Uses message-based communication to provide remote service

Signals

- **OS mechanism** to **notify a process** (one way)
  - Doesn’t carry information
- From **the OS**
  - Can be thought as a **software-generated interrupt/exception**
    - Synchronous – e.g., division by zero error
    - Asynchronous – e.g., timer/alarm
  - **Syscall:** process -> OS; **Signal:** OS -> process
- From **other processes** (including the process itself)
  - Only notification, no data **communication method**
    - Management – e.g., kill a process
    - Synchronization – e.g., POSIX RT Signals
    - etc.

Handling Signals (Linux Example)

- **Signal handler**
  - Code to process a signal
  - No return value (void)
- Handler must be registered
  - Or OS default action
- Send a signal
  - `kill()`
  - `raise()`

https://www.cs.auckland.ac.nz/references/unix/digital/APS33DTE/DOCU_006.HTM#realtime-handler-sec

Try at home!

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#define SIG_STOP_CHILD SIGUSR1

main() {
 pid_t pid;
 sigset_t newmask, oldmask;

 if ((pid = fork()) == 0) { /* Child */
     struct sigaction action;
     void catchit();
     sigemptyset(&newmask);
     sigaddset(&newmask, SIG_STOP_CHILD);
     sigprocmask(SIG_BLOCK, &newmask, &oldmask);
     action.sa_flags = 0;
     action.sa_handler = catchit;
     if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {
         perror("sigusr: sigaction");
         _exit(1);
     }
     sigsuspend(&oldmask);
 }
 else {  /* Parent */
     int stat;
     sleep(10);
     kill(pid, SIG_STOP_CHILD);
     pid = wait(&stat);
     printf("Child exit status = %d\n", WEXITSTATUS(stat));
     _exit(0);
 }
}
void catchit(int signo) { /* Signal Handler */
     printf("Signal %d received from parent\n", signo);
     _exit(0);
}
```