CS3510
Operating Systems

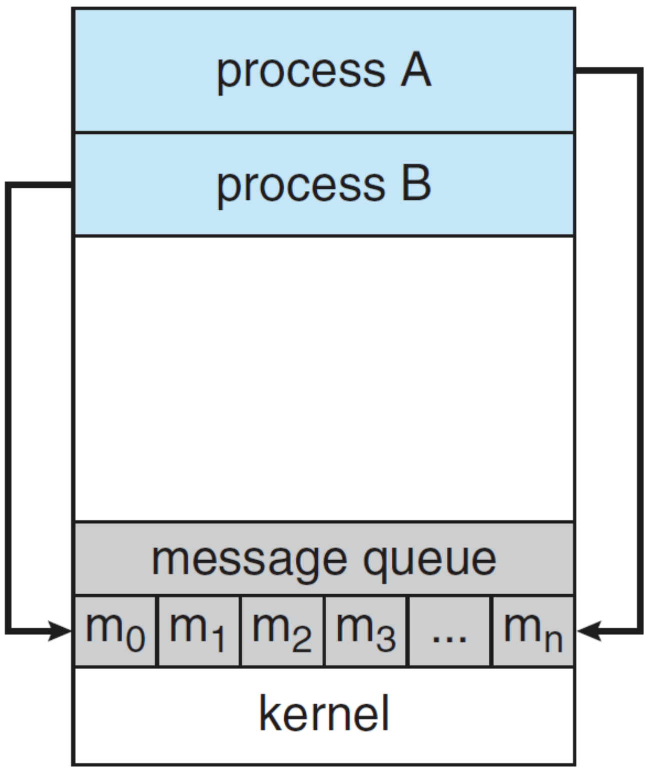
IPC

Bheemarjuna Reddy
IIT Hyderabad

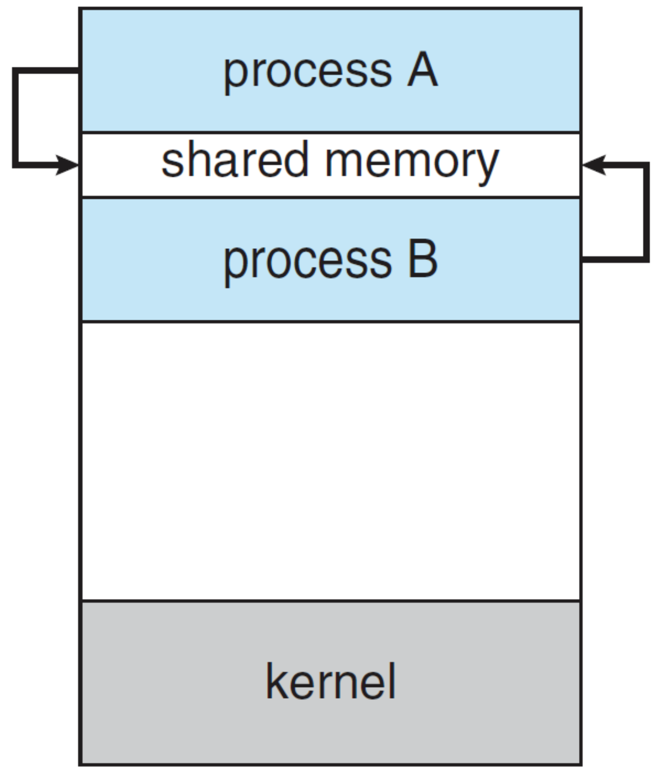
Inter Process Communication (IPC)

- 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)
- Two models of IPC
  - *Shared memory*
  - *Message passing*

Communication Models



Message Passing



Shared Memory

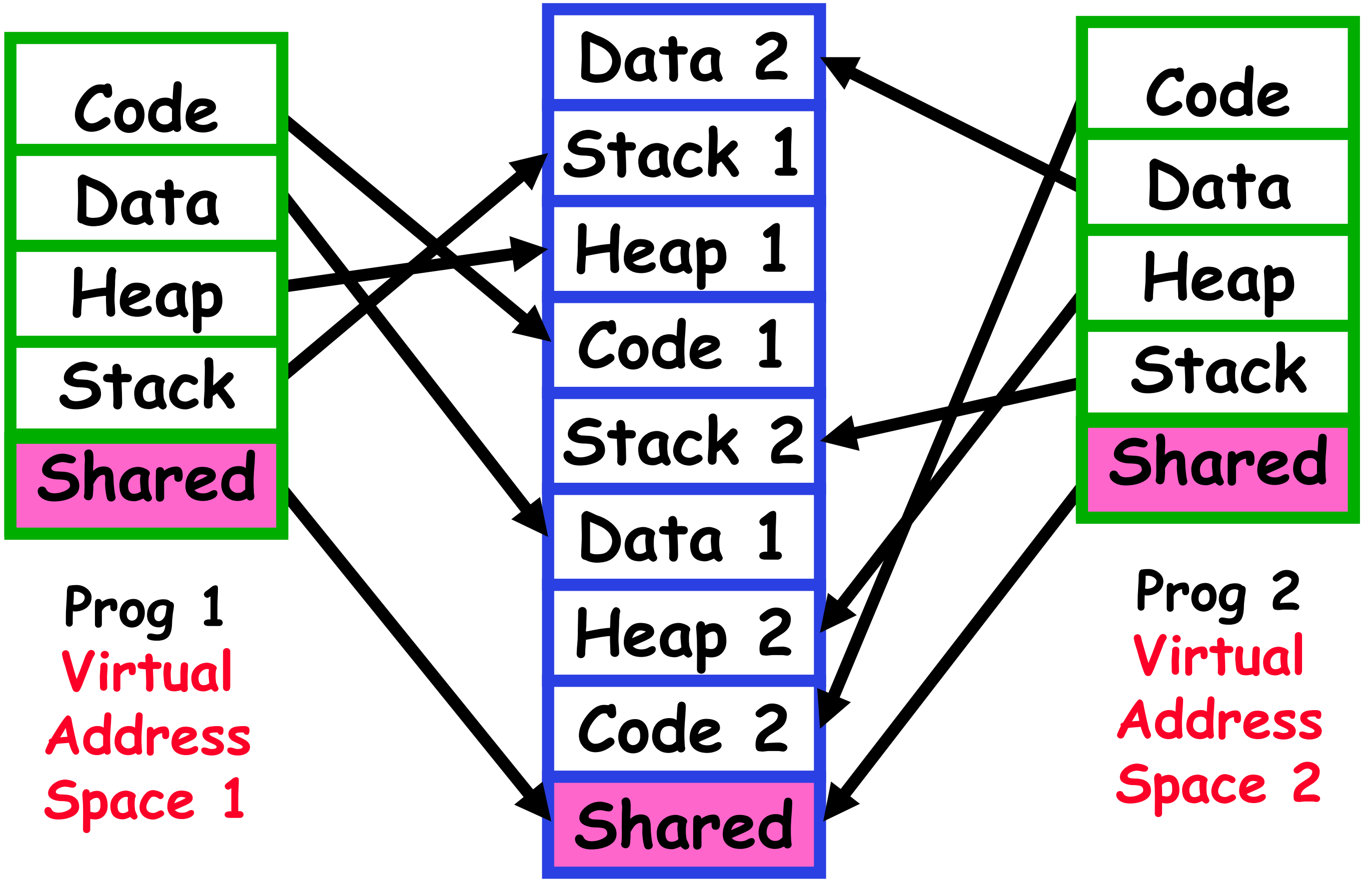
Producer-Consumer Problem

- Paradigm for cooperating processes
- *Producer* process produces information that is consumed by a *consumer* process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size

Shared Memory

- Processes establish a segment of memory as shared
  - Typically part of the memory of the process creating the shared memory.
  - Other processes attach this to their memory space.
  - Good when we have to share a lots of data
  - The communication is under the control of processes not the OS
- Requires processes to agree to remove memory protection for the shared section
  - Recall that OS normally protects processes from writing in each others memory

Shared Memory Communication



- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication and faster way of communication
  - But introduces complex synchronization problems
  - Cache coherence problem in multi-core systems

Bounded-Buffer – Shared-Memory Solution

- **Shared data**

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; // points to the next produced item
int out = 0; //points to the next consumed item
```

- **Solution is correct, but can only use `BUFFER_SIZE-1` elements**

Producer Process – Shared Memory

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

E
x
a
m
p
l
e

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
// Example using System V shared memory objects
// shm_open and mmap in POSIX
main(int argc, char **argv) {
   char* shared_memory;
   const int size = 4096;
   int segment_id = shmget(IPC_PRIVATE, size, IPC_CREAT | 0666);
   int cpid = fork();
   if (cpid == 0) //producer
   {
      shared_memory = (char*) shmat(segment_id, NULL, 0);//attach
      sprintf(shared_memory, "Hi from process %d",getpid());
   }
   else //consumer
   {
      wait(NULL);
      shared_memory = (char*) shmat(segment_id, NULL, 0);//attach
      printf("Process %d read: %s\n", getpid(), shared_memory);
      shmdt(shared_memory);//detach
      shmctl(segment_id, IPC_RMID, NULL);//remove segment
   }
}
```

E
x
a
m
p
l
e

```
#include <fcntl> #include <string.h>
#include <sys/shm.h> #include <sys/stat.h>
#define MAX_LEN 10000
struct region {        /* Defines "structure" of shared memory */
    int len;
    char buf[MAX_LEN]; };
struct region *rptr;
int fd; char * msg="Hello";
/* Create shared memory object and set its size */
fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR |
S_IWUSR);
if (fd == -1)   …  /* Handle error */;
if (ftruncate(fd, sizeof(struct region)) == -1) //set Size
   … /* Handle error */;
/* Map shared memory object to process' address space */
rptr = mmap(NULL, sizeof(struct region), PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);
if (rptr == MAP_FAILED)
    /* Handle error */;
/* Now we can refer to mapped region using fields of rptr */
sprintf(rptr,"%s",msg); //write to shared memory by producer
rptr+=strlen(msg);
...
```

Example

```
#include <fcntl> #include <string.h>
#include <sys/shm.h> #include <sys/stat.h>

int main()
{
/* name of the shared memory object */
const char *name = “/myregion";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;
/* open the shared memory object for reading by consumer*/
shm_fd = shm_open(name, O_RDONLY, 0666);
/* memory map shared memory object to process’ address space */
ptr = mmap(0, sizeof(struct region), PROT_READ, MAP_SHARED,
shm fd, 0);
/* read from the shared memory object */
printf("%s",(char *)ptr);
/* remove the shared memory object */
shm_unlink(name);
return 0;
}
```

Message Passing

- Useful for exchanging small amount of data w/o any conflicts
- Send(P, msg): Send msg to process P
  - Fixed vs variable size msg
- Recv(Q, msg): Receive msg from process Q
- Typically requires kernel intervention
  - User mode to kernel mode for Sending
  - Kernel mode to User mode for Receiving
- Communication link is needed for msg passing
  - Physical link realization
    - » shared memory, hardware bus, network
  - Logical implementation of link and its basic operations
    - » Direct vs indirect communication
      - Direct communication
        - Hardcode sender/receiver IDs (Symmetry)
        - Hardcode sender only (Asymmetry)
      - Indirection using mailboxes/ports
        - Owned by Process (owner/User) vs Owned by OS
        - Send(boxA, msg) and Receive (boxA, msg)

Message Passing

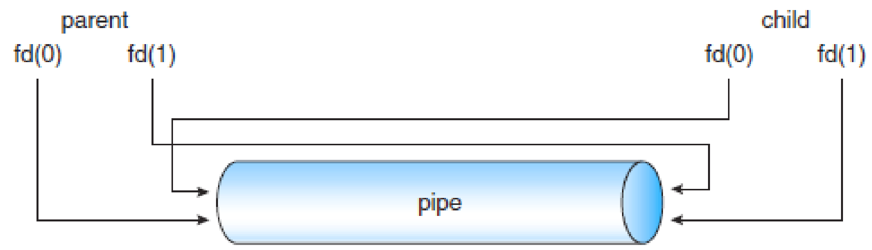
- Logical implementation of link and its basic operations
  - » Synchronous vs asynchronous communication
  - » Automatic vs explicit buffering
- Possible impl. of send()/receive() primitives
  - Blocking send/receive: process is blocked till msg is tx/rx
  - Non-blocking send/receive: Send msg & resume; Receive valid msg or return NULL w/o blocking
- When both send and receive are blocking, no buffer is needed. Other combinations need *buffering*.
  - Zero capacity buffer
    - » Needs synchronous sender.
  - Bounded capacity buffer
    - » If the buffer is full, the sender blocks.
  - Unbounded capacity buffer
    - » The sender never blocks
- Easier to implement in distributed and multi-core sys with NUMA compared to shared memory, but slower than that (sys calls)

Producer-Consumer Solution - Message Passing

```
message next_produced;

while (true) {
       /* produce an item in next_produced */

       send(next_produced); //blocking send
}
```

```
message next_consumed;

while (true) {
       receive(next_consumed)//blocking receive

       /* consume the item in next_consumed */
 }
```

Additional Communication mechanisms: Pipes



- Pipes
  - A pipe is a stream of communication between two processes
  - You can think of it as a virtual file stream shared between two processes: producer and consumer
  - A process can read and/or write to a pipe
  - Two processes can communicate via a pipe without even knowing it.
    - » Example: $cat helloWorld.c | less
  - This forms the backbone of Unix-like environments.
  - The pipe() function gets two descriptors (integer labels)
  - Read descriptor fd(0)– read from the pipe
  - Write descriptor fd(1) – write to the pipe
  - Both processes must know the descriptors
  - read() and write() API calls are used with the pipe

Pipes

- An ordinary pipe cannot be accessed from outside the process that created it.
  - A parent will create a pipe, then fork so the child can access it
  - So, ordinary pipes can only be used with processes on the same machine
    - » i.e., between parent and child processes in Windows and Unix systems
  - Allow only one-way communication
    - » $ ls | more
  - Child processes inherit all open files (pipes are a special kind of file) from the parent
  - Use two pipes for two-way communication
  - Once the processes end, the pipes no longer exist

```c
int main(void)
{
  int pid;
  char buffer[1024];
  int fd[2];

  pipe(fd); /* ordinary pipes; fd[0] is for read-end, fd[1] is for write-end of pipe */

  pid = fork();

  if (pid == 0) /* child */
  {
    int count;
    close(fd[0]); /* close unused READ end, child will write */

      /* prompt user for input */
    printf("input: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf("child: message is %s", buffer);

      /* write to the pipe (include NUL terminator) */
    count = write(fd[1], buffer, strlen(buffer) + 1); //pipe is a special type of file
    printf("child: wrote %i bytes\n", count);
    close(fd[1]);
    exit(0);
  }
  else /* parent */
  {
    int count;
    close(fd[1]); /* close unused WRITE end  */
    wait(NULL);   /* reap the child */
      /* read from the pipe */
    count = read(fd[0], buffer, sizeof(buffer));
    printf("parent: message is %s", buffer);
    printf("parent: read %i bytes\n", count);
    close(fd[0]);
}

}
```

Pipes

- Named pipes are more powerful than ordinary pipes.
  - They can be used by several processes at once.
  - They don't require a parent-child relationship.
  - They exist independently of the process that created them.
    - » Much like how files created on the disk by a process exist after the process ends.
  - See mkfifo on Unix-based systems and CreateNamedPipe on Windows
  - Unix named Pipes: FIFOs
    - » Bidirectional, but half-duplex; appear like files in fileSystem
    - » Communicating processes must reside in the same machine
    - » Manipulated with open, read, write, close system calls
  - Windows named Pipes:
    - » Offers a richer communication mechanism than Unix FIFOs
    - » Full-duplex
    - » Processes may reside on different machines

Summary

- IPC for cooperating processes
  - Shared memory
    - » OS provides shared memory and the application program has to take care of communication
  - Message passing
    - » Slower, but OS takes care of communication b/w processes
  - Pipes
    - » Simple way of communication between processes
- Sockets, Remote Procedure Calls (RPC) and named pipes are used for communication in client-server systems

- We discuss other IPC mechanisms, related to synchronization of processes, later in OS-2

Reading Assignment

- **Chapter 3 from OSC by Galvin et al**
- **Chapter 2 from MOS by Tanenbaum et al**
- **Chapter 3. Processes and** Chapter 19. Process Communication **from Understanding the Linux Kernel by** Daniel P. Bovet and Marco Cesati (available on Intranet)
- The Linux Programming Interface by Michael Kerrisk
- **http://man7.org/linux/man-pages/man3/shm_open.3.html**
- **http://pubs.opengroup.org/onlinepubs/009695399/functions/shm_open.html**
- **http://man7.org/linux/man-pages/man2/open.2.html**
- **http://man7.org/linux/man-pages/man2/shmget.2.html**