Interprocess Communication (IPC)

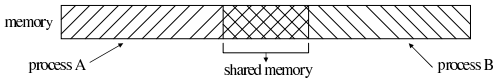
CS 537 - Intoduction to Operating Systems

Cooperating Processes

- How do we let processes work together?
- Possible solutions:
  - share memory
    - fast and simple
    - how to keep secure and synchronized?
  - message passing
    - no shared memory
    - send and receive messages
    - more difficult to program
    - makes security and synchronization simpler

Shared Memory

- Easiest way for processes to communicate
  - at least some part of each processes' memory region overlaps the other processes'



- If A wants to communicate to B
  - A writes into shared region
  - B reads from shared region
- Can use simple load and stores
  - no special commands for communication

Shared Memory

- How much memory to share?
  - none (this really isn't shared memory :)
  - some
  - all
- How to allocate shared memory?
  - automatically on creation of a child
    - parent and child share memory
  - explicitly through system calls
    - a process requests OS to set up a shared memory segment
    - a process makes another system call to share the memory
- No matter how it's shared, the memory resides in user space

Shared Memory

- Two major disadvantages to shared memory
  - synchronizing access
    - B needs to know when A has written a message into the shared memory
      - so B can read the message
    - A needs to know when B has read the message
      - so A can write another message
  - security
    - possible for A or B to accidentally write over a message
      - program bug
    - B may maliciously read or write some part of A's memory that it shares

Simple Examples

- Sharing memory

```
char* shareMem[80];   // memory accessible to A and B
int value;   // more memory shared by A and B
```

Process A	Process B
strcpy(shareMem, "hello");	strcpy(myBuf, shareMem);
value = 23;	myValue = value;

- notice that A and B communication does not involve the operating system or any special calls
  - just reading and writing regular memory
- also notice, if B performs it's read of *shareMem* before A writes to it, B will get garbage
  - we'll cover synchronization in a week

Shared Memory Review

- Intuitive to program
  - just access memory like any other program
- High performance
  - does not get the OS involved
- Must synchronize access to data
  - more on this later
- Have to trust other processes sharing the memory

No Shared Memory

- In the absence of shared memory, the OS must pass messages between processes
  - use system calls to do this
- Several major techniques for doing this
  - pipes, message passing, ports
- Several major issues to consider

IPC Issues

- direct or indirect communication
  - naming issues
- synchronous or asynchronous communication
  - wait for communication to happen or assume it does
- automatic or explicit buffering
  - how to store messages
- fixed messages or variable sized
  - greatly affects overhead and memory requirements

Indirect vs. Direct Naming

- Direct Naming
  - explicitly state which process to send/receive
  - code fragment
    - *send(Process P, char [] message, int size)*
    - *receive(Process P, char [] message, int size)*
  - must know before hand exactly where to send/receive message

Indirect vs. Direct Naming

- Indirect naming
  - use "mailboxes"
  - processes extract messages from a mailbox
    - may grab a message from a specific process
    - may grab a specific type of message
    - may grab the first message
  - send/receive to/from mailboxes
  - code fragment
    - *send(Mailbox A, char [] message, int size)*
    - *receive(Mailbox A, char [] message, int size)*

Synchronization

- Blocking send
  - suspend sending process until message received
- Non-blocking send
  - resume process immediately after sending message
- Blocking receive
  - suspend receiving process until data is received
- Non-blocking receive
  - return either a message or
  - null if no message is immediately available

Synchronization Trade-Offs

- Blocking
  - guarantees message has been delivered
  - drastically reduces performance
- Non-blocking
  - much better performance (hides latency of message sending)
  - could cause errors if messages are lost

Timeout

- One other option is to use timouts
  - typically with a blocking send/receive
- If a process blocks for a certain amount of time, call returns with a special error code
  - indicates message wasn't sent/received
- User can write special code to deal with this case

Timeout Example

```
char msg[80];
int errCode;

setTimeout(250);   // 250 ms before timing out
if((errCode = recv(mailbox, msg, 80)) == TIME_OUT) {
    // handle this case
}
else if(errCode < 0) {  // some kind of error
    // handle error
}
…
```

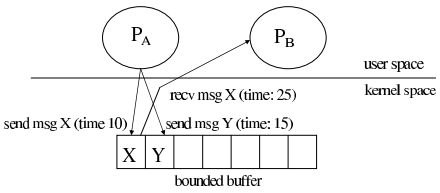
Buffering

- Buffering allows messages to be saved and read or transmitted later
- Requires sufficient memory to store messages
- Can drastically improve performance of applications

Types of Buffering

- Zero Capacity
  - no buffering at all
  - must be "listening" when a message comes in
- Bounded Capacity
  - some max, *n*, of messages will be buffered
  - be careful when queue gets full
- Unbounded Capacity
  - no limit to the number of messages
  - not usually very realistic assumption
  - this is not very realistic, buffers usually have finite capacity

Buffering Example



Note: OS stores message so that P_A can go back to work instead of waiting for P_B to do a receive

Bounded Buffers

- What to do if a bounded buffer gets full?
  - make the send call fail
    - return an error code to the user program
  - make the send call block
    - even if it normally wouldn't do so

Message Length

- Fixed Length
  - how big to make the messages?
- Variable Length
  - how to handle buffering?

Fixed Length Messages

- makes buffering simpler
  - know exact size needed for each message
- to send a large message, break to small bits
- can hurt performance for large messages
  - overhead of creating many small messages
- What size?
  - too big wastes buffering space in memory
  - too small hurts performance for large messages
  - some systems provide several message sizes

Variable Length Messages

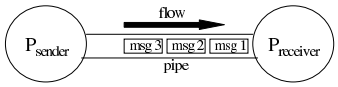
- Provides great flexibility
  - send as much data as you want and only incur overhead of setting up message once
- How do you guarantee buffering space?
  - what if buffer is almost full when a new, large message comes in?
  - what if one message is larger than entire buffer?
- Consider variable size up to some max
  - this is one of the most common methods

Common Message Passing Methods

- Pipes
  - direct connection between 2 or more processes
- Message Queues
  - shared buffer in OS where processes place and retrieve messages
- Ports
  - process specific buffer in OS

Pipes

- Conceptually
  - a pipe is a link between two processes
  - one process writes into one end of the pipe
  - another process reads out of the other end
    - messages are read in the order they are written



  - for both processes to be able to read and write simultaneously, two pipes are necessary

Pipes

- Unix implementation
  - a pipe is represented as a file without any data
    - the "file" is created using the *pipe()* system call
  - when the file is created, the operating system allocates kernel space for storing messages
    - no messages actually go to a file
  - any *read()* or *write()* system call operations on a pipe, actually read and write to the reserved kernel space

Using Unix Pipes

```
int main(int argc, char** argv) {
  int fd[2];   // 2 ints, one for reading, one for writing
  int pid;     // will be used for a fork later

  pipe(fd);  // open 2 pipes: fd[0] - read, fd[1] - write
  pid = fork();
  if(pid == 0) {   // child process - receiver
    char buf[80];
    read(fd[0], buf, 80);
    printf("%s\n", buf);
  }
  else   // parent process - sender
    write(fd[1], argv[2], 80);

  return 0;
}
```

Named Pipes (FIFO)

- Problem with pipes: only way for two processes to share a pipe is to share a common ancestry
  - no way for two unrelated processes to communicate
- Solution: named pipes
- A named pipe is represented as a special file
  - file is created using *mknod()* or *mkfifo()* system calls
  - these files contain no data

Named Pipes (FIFO)

- Using named pipes
  - must be opened - like a regular file
    - use the *open()* system call
    - this creates space in the kernel for reading and writing to
  - from here it looks just like a regular pipe
    - use *read()* and *write()* to receive/send messages
  - user can close the pipe when finished using it
    - use the *close()* system call

Pipes

- Pipe characteristics
  - indirect naming
    - multiple processes can read and write a pipe
  - asynchronous or synchronous
    - determined at creation time of pipe
  - bounded buffer
    - pipe only has a certain amount of capacity
  - message length is variable up to some maximum

Message Queues

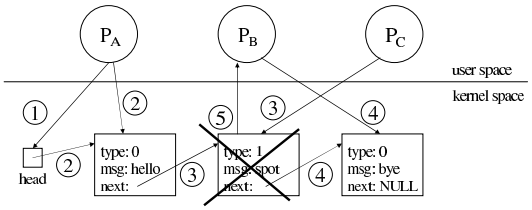
- Conceptually
  - a message queue is a repository for named messages
  - these messages can be removed in any order based on their name
    - very similar to a post office
      - all messages get sent to the post office
      - multiple people can enter the post office
      - each person has their own box at the post office

Message Queues

- Unix System V implementation
  - use the *msgget()* system call to allocate space in the kernel for a group of messages
    - it is possible to create separate queues
  - each message sent to a specific group, may contain a type field
    - type field can be used to implement priority, specific processes to receive message, etc.
    - to send a message, use the *sendmsg()* system call
    - maximum number of messages in queue is limitted
  - request to receive message may contain a specific type
    - usually retrieve the first message in queue of a specific type
    - if no type specified, grab the first message in queue

Message Queues

- Message queue is usually implemented as a list
  - so it's not a queue in the true data structure sense
- Example



Message Queues

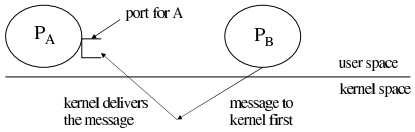
1) P_A create a message queue
   - initially it is empy
2) P_A sends a message (type 0) to the queue
   - node added to list
3) P_C sends a message (type 1) to the queue
   - node added to list
4) P_B sends a message (type 0) to the queue
   - node added to list
5) P_B receives a message (type 1) from the queue
   - queue is searched for first type 1 message
   - message is removed from queue

Message Queues

- Message queue characteristics
  - indirect naming
    - multiple processes can read and write to same queue
  - asynchronous or synchronous
    - determined at creation time of queue
  - bounded buffer
    - buffer only accepts a maximum number of messages
  - fixed size header, variable length messages
    - a fixed amount of space is allowed for all the messages in a queue - can't exceed this

Ports

- Conceptually
  - a port has a certain amount of memory allocated for it
  - any message sent to a specific port is put into the memory for that port
  - each port is associated with a single process only
  - a port is identified by a single integer number
  - messages are sent to specific ports



Ports

- A process can have multiple ports
  - which port a message is delivered to depends on which one it is intended for
- The memory for each port can be located in either the kernel or the user space
  - more flexible if in user space
  - more copying required if in user space
    - message gets copied to kernel and then to port

Ports

- Sending a message
  - you send the message to a specific port
  - *send(port, msg, size);*
- Receiving a message
  - indicate which port to receive a message from
  - *recv(port, buf, size);*
- More details on this is left to a networking course

Ports

- Port characteristics
  - direct naming
    - a message gets delivered to a specific process
    - the process associated with the port
  - asynchronous
    - assume message sent once given to OS
    - can simulate synchronous
  - bounded buffer
    - port only has a certain amount of capacity
  - message length is variable up to some maximum