Distributed Systems
Lecture 2

Distributed Systems *Lite*: Multiprocessing on a single machine

Joseph Phillips

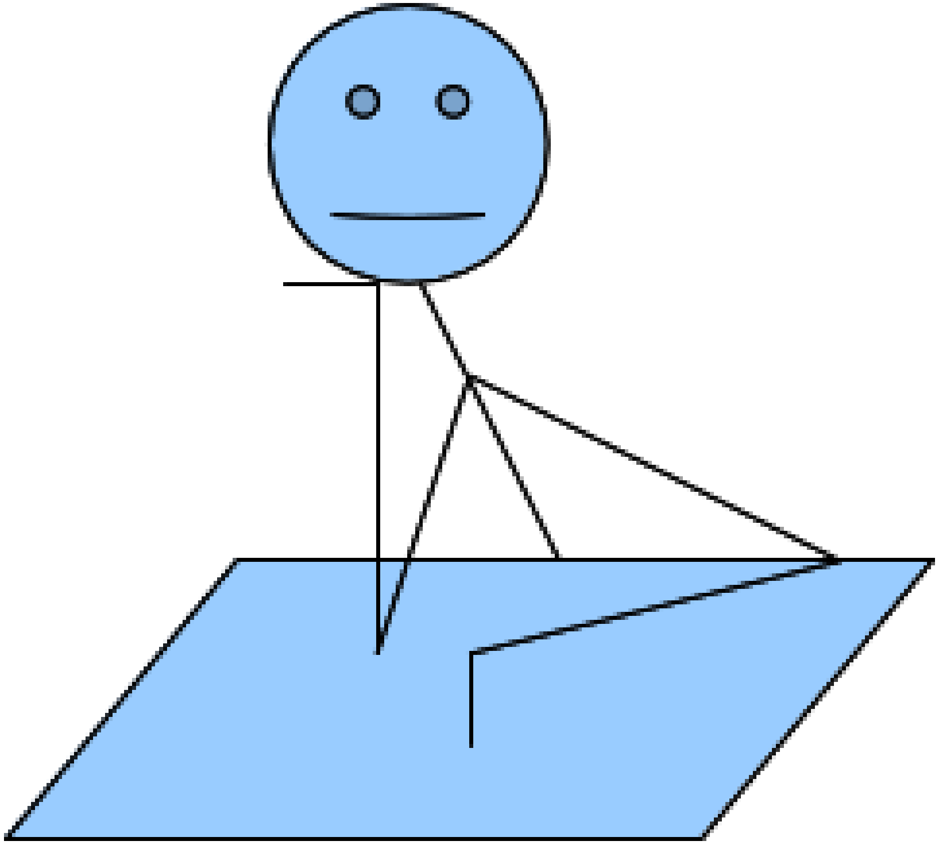
Last modified 2019 September 24

Topics

- Motivation
- Issues
- Inter-process communication
  - pipes
  - socketpairs
  - messages
  - shared memory

Motivation

Question: We have the Internet! Why limit ourselves to one machine?

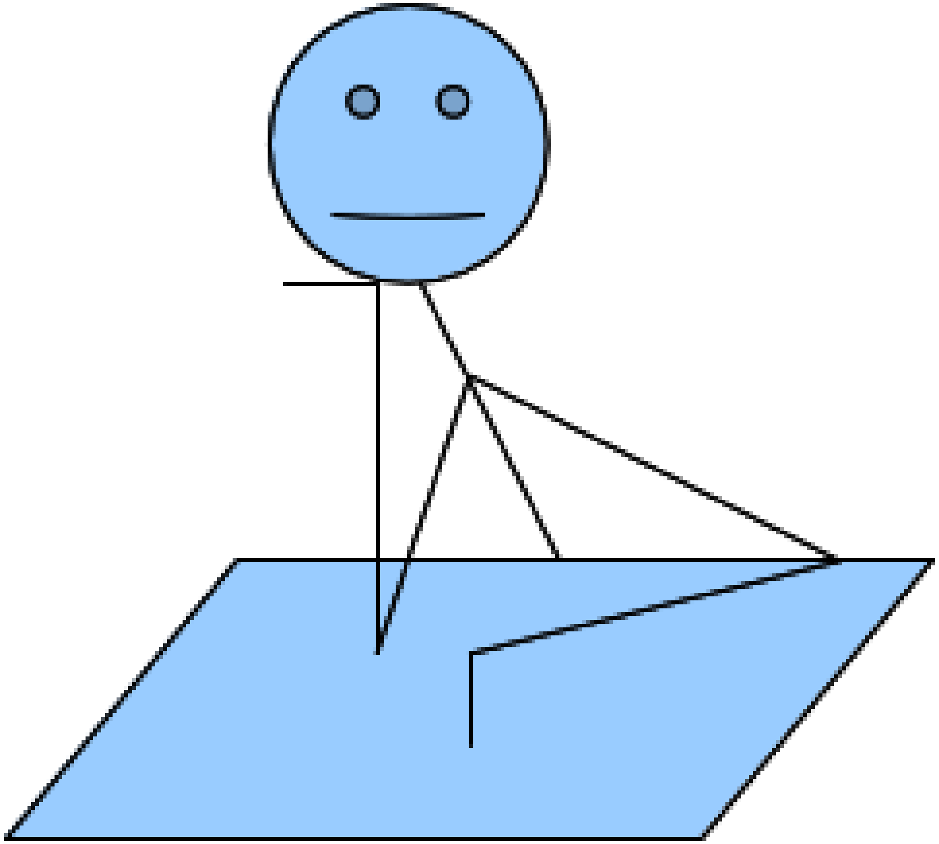


Motivation

- Answer #1: To debug a system before distributing it
- Answer #2: To make sure even a distributed system will work even if there is no network

Motivation

Question: Fine! Limit yourself to one machine. If you want to do two things at once, just use threads instead of inter-process communication



Motivation

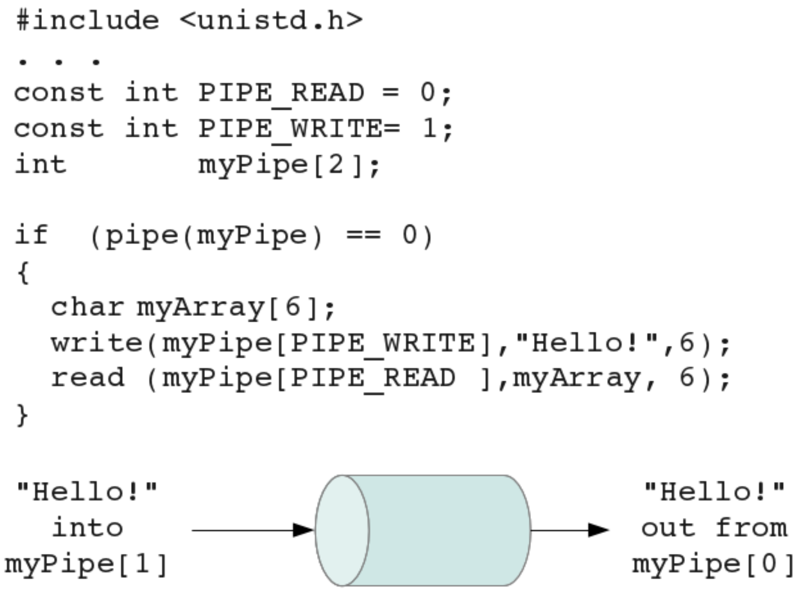
- fork()/execl() is more flexible than threads
  - Child process can run another program without destroying parent
- fork()/execl() is more robust than threads
  - If child crashes, does not crash parent

Issues

- Do not have to worry about networking (yet!)
- Do have to worry about
  - Inter-process communication

Inter-process communication: pipes (1)

- Pipes:
  - One way, file-descriptor based communication between related processes
    - child-to-parent
    - parent-to-child
    - child-to-child
  - Takes array of 2 ints
    - array[0]: reading fd (think STDIN_FILENO = 0)
    - array[1]: writing fd (think STDOUT_FILENO = 1)
- pipe(int[2])
- pipe2(int[2],int flags)
  - flags are bitwise OR of:
    - O_NONBLOCK: make reading non-blocking
    - O_CLOEXEC: for use in some multi-threaded apps
- ***Remember!* Processes should close unneeded end!**



```
#include <unistd.h>
. . .
const int PIPE_READ = 0;
const int PIPE_WRITE= 1;
int       myPipe[2];

if  (pipe(myPipe) == 0)
{
  char myArray[6];
  write(myPipe[PIPE_WRITE],"Hello!",6);
  read (myPipe[PIPE_READ ],myArray, 6);
}
```

myPipe: An OS-owned buffer

Inter-process communication: pipes (2)

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <unistd.h>

const int       BUFFER_LEN      = 256;

#define         TEXT            \
     "Something to send down the pipe."

int             main            ()
{
  int   fd[2];

  if  (pipe(fd) < 0)
    exit(EXIT_FAILURE);

  pid_t childId = fork();

  if  (childId < 0)
    exit(EXIT_FAILURE);

  if  (childId == 0)
  {
    char        buffer[BUFFER_LEN];
    int         numBytes;

    close(fd[1]);

    if  (read (fd[0],buffer,BUFFER_LEN) > 0)
    {
      printf("Child received \"%s\"\n",buffer);
    }

    close(fd[0]);
  }
  else
  {
    close(fd[0]);
    printf("Parent sending \"%s\"\n",TEXT);
    write(fd[1],TEXT,sizeof(TEXT));
    close(fd[1]);
  }

  return(EXIT_SUCCESS);
}
```

Inter-process communication: pipes (3)

- dup(int oldFd)
  - Copy file descriptor ‘oldFd’ into lowest available position
  - Lowest position generally obtained by close()-ing it immediately beforehand . . .
  - . . . or use dup2()

- dup2(int oldFd, int newFd)
  - close(newFd) (if open) then copy ‘oldFd’ into ‘newFd’
  - If ‘oldFd’ is invalid, then ‘newFd’ is not closed
  - if ‘oldFd==newFd’, then nothing happens

Your turn!

Write a program that

1. makes a pipe
2. forks() two child processes
3. one child runs ‘/usr/ls’, it sends it output to the pipe
4. the other child runs ‘/usr/bin/wc’, and gets its input from the pipe
5. output of wc sent to stdout

Remember! close() unneeded file descriptors!

Inter-process communication: sockets (1)

- socketpair()
  - Makes a pair of file descriptors (like pipe())
  - Two-way communication

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int
sv[2]);
```

  - `domain` should be `AF_UNIX` (= `AF_LOCAL`)
  - `type` could be `SOCK_STREAM` (for TCP), `SOCK_DGRAM` (for UDP), `SOCK_SEQPACKET` (like TCP)
  - `protocol` can be 0 if there is only one choice for the given `type`.
  - `sv[]` receives pair of socket file descriptors, `sv[0]` and `sv[1]` indistinguishable
- Remember: `close()` unneeded file descriptor.

Inter-process communication: sockets (2)

```
/*
 * From ibm.com downloaded 2019-03-31
 */

/* This program fragment creates a pair of connected sockets then
 * forks and communicates over them.  Socket pairs have a two-way
 * communication path.  Messages can be sent in both directions.
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#define DATA1 "In Xanadu, did Kublai Khan..."
#define DATA2 "A stately pleasure dome decree..."

int     main()
{
   int sockets[2], child;
   char buf[1024];
   if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
      perror("opening stream socket pair");
      exit(EXIT_FAILURE);
   }
   if ((child = fork()) == -1)
      perror("fork");
   else if (child) {     /* This is the parent. */
      close(sockets[0]);
      if (read(sockets[1], buf, 1024, 0) < 0)
         perror("reading stream message");
      printf("-->%s\n", buf);
      if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
         perror("writing stream message");
      close(sockets[1]);

   } else {     /* This is the child. */
      close(sockets[1]);
      if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
         perror("writing stream message");
      if (read(sockets[0], buf, 1024, 0) < 0)
         perror("reading stream message");
      printf("-->%s\n", buf);
      close(sockets[0]);
   }
}
```

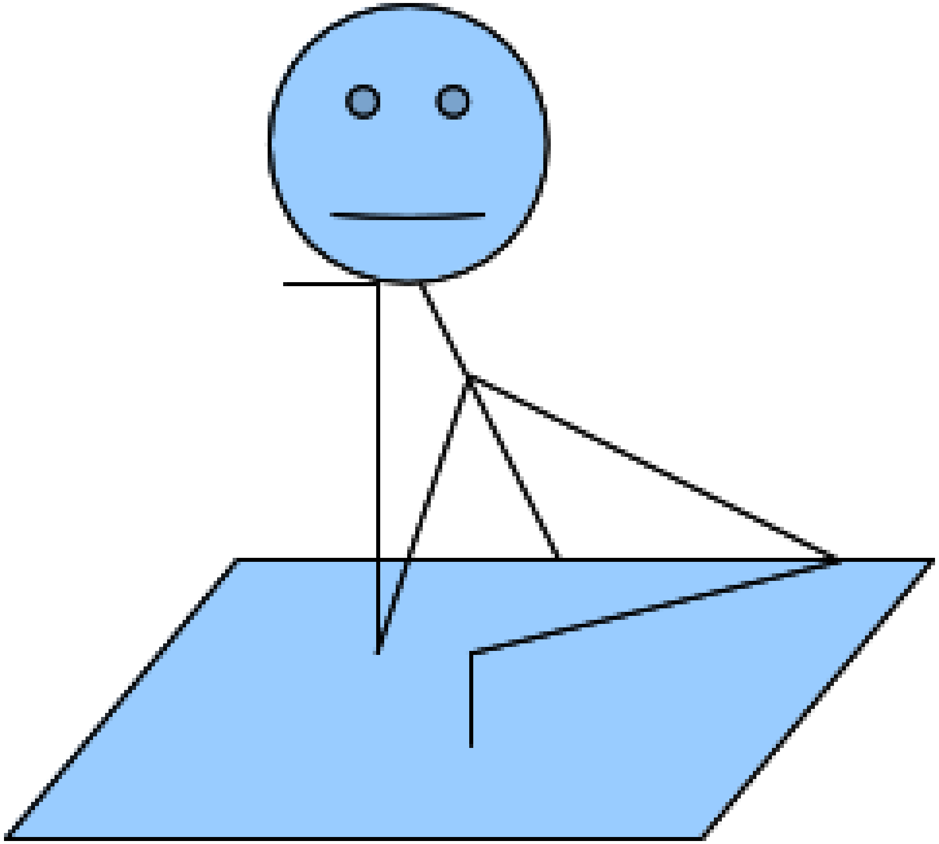
Your turn!

Write a program that:

1. creates a socketpair
2. `fork()`s a child process
3. the parent process gets text from user, send to socket
4. the child process gets text from socket, uppercases it, sends back to socket
5. the parent process gets text from socket and prints it

Inter-process communication: sockets (2)

- ***Question:*** “*Hey! I thought sockets were for communication between different machines!*”
- Server-side:
  - socket()
  - bind()
  - listen()
  - accept()
- Client-side:
  - getaddrinfo()
  - socket()
  - connect()
- ***Stay tuned!***



Inter-process communication: Message Queues (1)

- Asynchronous communication between processes
- Send “structs” between process
  - A little higher level than pipes/sockets, which send bytes

- int msgget (int key, int flags)
  - `key` is like a filename
  - `flags` is like for open() for files:
    - (0660|IPC_CREAT) to create
    - 0 to attach to existing queue
  - return msg queue id (like a file descriptor)

Say!
Where should that key come from?

- It is meant to come from ftok():
  - key_t ftok(const char *pathname, int proj_id);
- Idea:
  - This user has access to files other users don’t.
  - Therefore, build key based upon a file this user can access
- pathname
  - An existing, accessible file
- proj_id
  - A non-zero integer, whose lowest byte will be used
- (We will just enter an integer.)

Inter-process communication: Message Queues (2)

- msgctl(int msgQId, int cmd, struct msqid_ds *buf)
  - msgQId: which msg queue
  - cmd:
    - `IPC_STAT`: get info
    - `IPC_SET`: set info
    - `IPC_RMID`: delete queue
  - buf:
    - Information about the queue
    - `buf.msg_qbytes`: How big the queue should be

Inter-process communication: Message Queues (3)

- msgsnd(int msgQId, const void* msgp, size_t msgs, long msgtyp, int msgflg)
  - msgQid: which queue
  - msgp: ptr to message
    - A struct with a `long` member var as first member
    - ***long value must be positive!***
  - msgs: size of message
    - Does not count beginning long var
  - msgtyp:
    - “For whom”: same queue can be used by multiple readers.
    - ‘msgtyp’ tells for whom msg is for
  - msgflg: flags
  - Return value
    - 0 on success
    - -1 on failure

- msgrcv(int msgQId, void* msgp, size_t msgs, long msgtyp, int msgflg)
  - msgQid: which queue
  - msgp: ptr to message
  - msgs: size of message (not including long var)
  - msgtyp:
    - “For whom”: same queue can be used by multiple readers.
    - ‘msgtyp’ tells for whom msg is for
    - If 0 then first message of any type
  - msgflg: flags
  - Return value
    - number of bytes received
    - -1 on failure

Inter-process communication: Message Queues (4)

```
// msg.h
#define        TEXT_LEN 80

struct         AMessage
{
  long         msgType_;
  float        floatPt_;
  int     integer_;
  char         text_[TEXT_LEN];
};
```

```
//  makeAndWriteMsgQueue.c

#include   <stdlib.h>
#include   <stdio.h>
#include   <string.h>
#include   <sys/msg.h>

#include   "msg.h"


intgetMsgKey  ()
{
  char   text[TEXT_LEN];
  int  toReturn;

  printf("Please enter a msg id integer: ");
  fgets(text,TEXT_LEN,stdin);
  toReturn = strtol(text,NULL,0);
  return(toReturn);
}
```

Inter-process communication: Message Queues (5)

```
int makeMsgQueue ()
{
  int   toReturn;
  struct msqid_ds msgQInfoBuffer;

  do
  {
    int key  = getMsgKey();

    toReturn   = msgget(key,0660 | IPC_CREAT);
  }
  while  (toReturn < 0);

  msgctl(toReturn,IPC_STAT,&msgQInfoBuffer);
  msgQInfoBuffer.msg_qbytes = 4096;
  msgctl(toReturn,IPC_SET,&msgQInfoBuffer);

  return(toReturn);
}
```

```
void            writeMsgs       (int    msgQId
                                 )
{
  struct AMessage       aMsg;
  char                  format[TEXT_LEN];
  char                  line[TEXT_LEN * 2];

  aMsg.msgType_ = 1;
  snprintf(format,TEXT_LEN,"%%f %%d %%%ds",TEXT_LEN);

  while  (1)
  {
    aMsg.floatPt_       = 0.0;
    aMsg.integer_       = 0;
    memset(aMsg.text_,'\0',TEXT_LEN);

    printf("Please enter a float, int and word (or blank line to quit): ");
    fgets(line,TEXT_LEN*2,stdin);

    if  (sscanf(line,format,&aMsg.floatPt_,&aMsg.integer_,aMsg.text_) <= 0)
    {
      msgsnd(msgQId,&aMsg,sizeof(aMsg)-sizeof(long),0);
      break;
    }

    aMsg.text_[TEXT_LEN-1]      = '\0';
    msgsnd(msgQId,&aMsg,sizeof(aMsg)-sizeof(long),0);
  }

}
```

Inter-process communication: Message Queues (6)

```
int       main       ()
{
  int msgQId= makeMsgQueue();
  writeMsgs(msgQId);
  return(EXIT_SUCCESS);
}
```

```
//  readAndDelMsgQueue.c

#include   <stdlib.h>
#include   <stdio.h>
#include   <string.h>
#include   <sys/msg.h>

#include   "msg.h"


intgetMsgKey  ()
{
  char   text[TEXT_LEN];
  int  toReturn;

  printf("Please enter a msg id integer: ");
  fgets(text,TEXT_LEN,stdin);
  toReturn = strtol(text,NULL,0);
  return(toReturn);
}
```

Inter-process communication: Message Queues (7)

```
int main()
{
  struct AMessage    msg;
  int   msgKey    = getMsgKey();
  int   msgQId    = msgget(msgKey,0);

  if  (msgQId < 0)
  {
    fprintf(stderr,"Sorry!\n");
    exit(EXIT_FAILURE);
  }

  while  (1)
  {
    if  ( msgrcv(msgQId,&msg,sizeof(msg),1,0) < 0)
      break;

    if  ((msg.floatPt_ == 0.0) && (msg.integer_ == 0) && (msg.text_[0] ==
'\0'))
      break;

    printf("Float:\t%g\nInt:\t%d\nWord:\t%s\n\n",
   msg.floatPt_,msg.integer_,msg.text_
  );
  }

  msgctl(msgQId,IPC_RMID,NULL);
  return(EXIT_SUCCESS);
}
```

$ ipcs

- Command line tool for seeing active message queues (and shared memory, and semaphores)

$ ipcrm -Q <key>

- Delete message queue by its key

Your turn!

Write an application:

Sender generates 2 random ints, sets calculation_:

- QUIT_CALC
- ADD_CALC
- MULT_CALC

Receiver receives message and either adds or multiplies or quits

```
typedef enum
    {
       QUIT_CALC,
       ADD_CALC,
       MULT_CALC
    }
   calc_ty;

struct MathProblem
{
  long msgType_;
  calc_ty calculation_;
  int int0_;
  int int1_;
};
```

Inter-process communication: shared memory (1)

- Allows multiple processes to access ***exact*** same pages
- If more than one process can write then should protect with semaphores
  - Much like pthread_mutex_t for threads
- Less of a use for this because threads already share memory

Inter-process communication: shared memory (2)

void* mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)

- Memory map, does several things including making shared memory
  - addr: where to make shared (if NULL then OS will choose)
  - len: number of bytes
  - offset: starting byte (if ‘fd’ is a file)
  - fd: file descriptor to map into memory
  - prot:
    - PROT_EXEC: may be executed
    - PROC_READ: may be read
    - PROC_WRITE: may be written
    - PROC_NONE: may be not be accessed
  - flags:
    - MAP_SHARED: visible to other processes

int munmap (void* addr, size_t length)

- Gets rid of memory map

Inter-process communication: shared memory (3)

```c
//  From a user named "slezica"
//  From https://stackoverflow.com/questions/5656530/how-to-use-shared-memory-with-linux-in-c
//  Downloaded 2019 Feb 3
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>

void* create_shared_memory(size_t size) {
  // Our memory buffer will be readable and writable:
  int protection = PROT_READ | PROT_WRITE;

  // The buffer will be shared (meaning other processes can access it), but
  // anonymous (meaning third-party processes cannot obtain an address for it),
  // so only this process and its children will be able to use it:
  int visibility = MAP_ANONYMOUS | MAP_SHARED;

  // The remaining parameters to `mmap()` are not important for this use case,
  // but the manpage for `mmap` explains their purpose.
  return mmap(NULL, size, protection, visibility, 0, 0);
}
```

```c
int main() {
  char* parent_message = "hello";  // parent process will write this message
  char* child_message = "goodbye"; // child process will then write this one

  void* shmem = create_shared_memory(128);

  int pid = fork();

  memcpy(shmem, parent_message, sizeof(parent_message));

  if (pid == 0) {
    sleep(1);
    printf("Child read: %s\n", shmem);
    memcpy(shmem, child_message, sizeof(child_message));
    printf("Child wrote: %s\n", shmem);

  } else {
    printf("Parent read: %s\n", shmem);
    sleep(2);
    printf("After 1s, parent read: %s\n", shmem);
  }
}
```

Inter-process communication: shared memory (4)

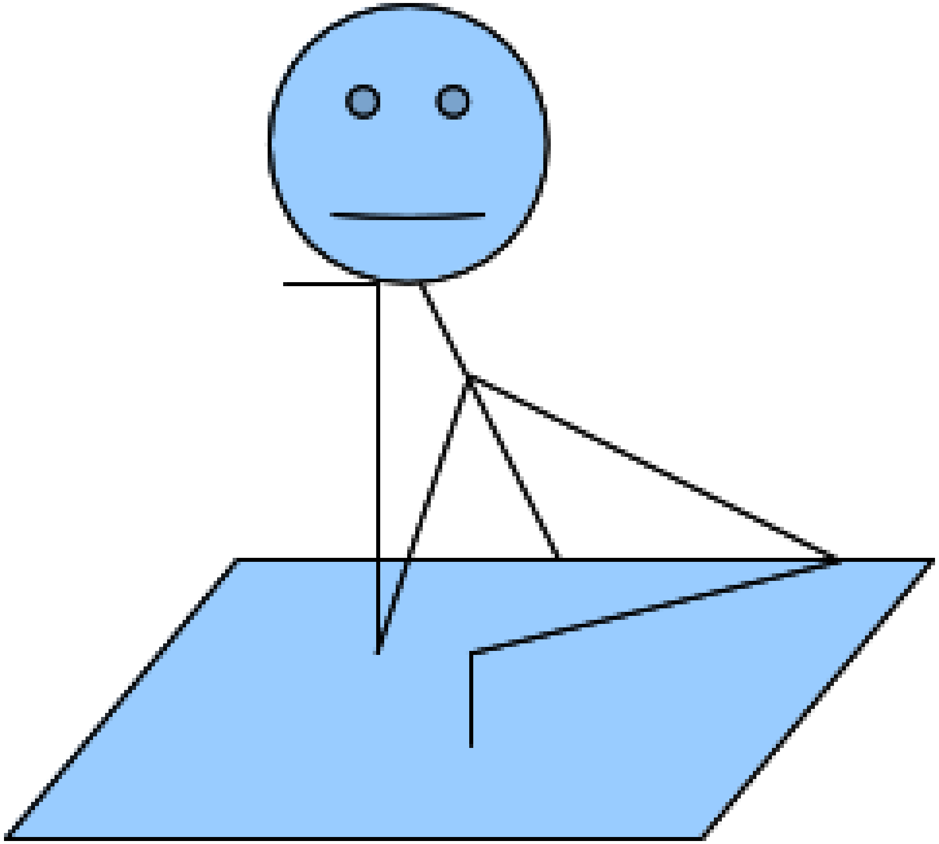
- Two other interfaces if want finer detail over which processes can access:
- Access with a “filename”
  - shm_unlink(), shm_open()
- Access with a key integer
  - shmget(), shmctl(), shmat(), shmdt()

Inter-process communication: shared memory (5)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#define SHARED_MEM_PATH    "/TestSMem"

int main()
{

  const char* parent_message = "hello";
  const char* child_message = "goodbye";

  int fd;
  size_t size;
  void* addr;

  shm_unlink(SHARED_MEM_PATH);

  fd = shm_open(SHARED_MEM_PATH, O_CREAT|O_EXCL|O_RDWR, S_IREAD|
S_IWRITE|S_IRGRP);
  size = 128;
  ftruncate(fd, size);

  addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

  memcpy(addr, parent_message, sizeof(parent_message));
  printf("Original contents: %s\n",(char*)addr);

  int pid = fork();
```

```
  if (pid == 0) {
    printf("Child closing %d\n",fd);
    close(fd);
    printf("munmap()\n");
    munmap(addr,size);

    printf("shm_open()\n");
    errno= 0;
    fd = shm_open(SHARED_MEM_PATH, O_RDONLY, S_IRUSR|
S_IWUSR|S_IRGRP|S_IWGRP);
    size = 128;
    printf("fd = %d %s\n",fd,strerror(errno));

    addr = mmap(NULL, size, PROT_READ , MAP_SHARED, fd, 0);

    printf("Child read: %s\n", addr);

    printf("Child will attempt to write, but fail:\n");
    memcpy(addr, child_message, sizeof(child_message));
    printf("Child tried to write: %s\n", addr);

  } else {
    printf("Parent read: %s\n", addr);
    sleep(10);
    printf("After 10 seconds, parent read: %s\n", addr);
    shm_unlink(SHARED_MEM_PATH);
  }

}
```

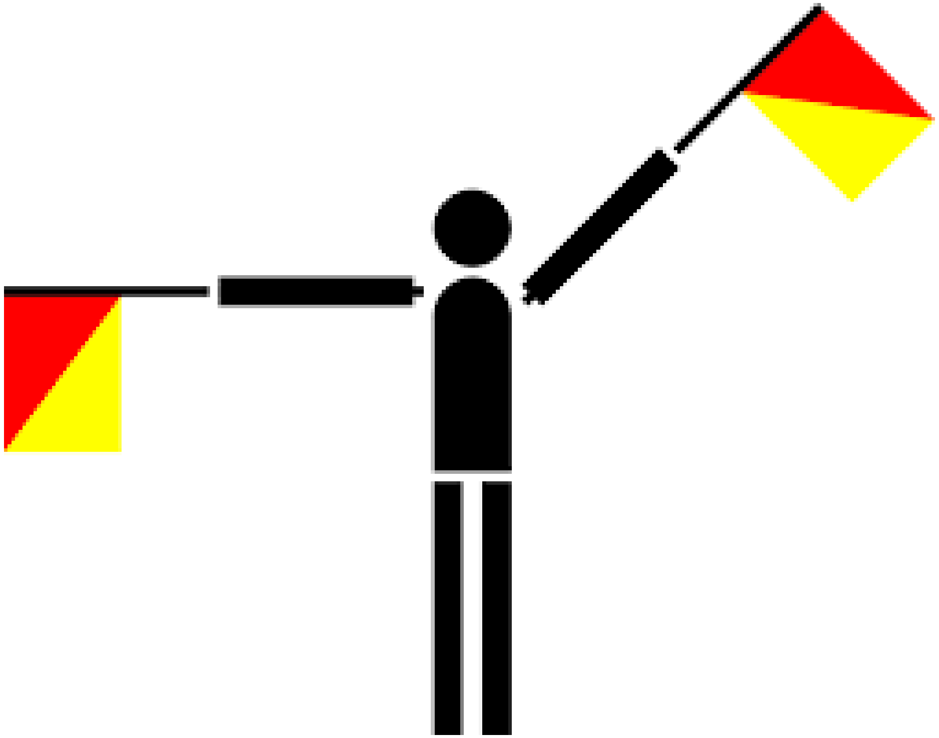
Inter-process communication: shared memory (6)



Clever student's question "*Uh-oh! If two processes can write to the same memory at the same time, then we have the same problem as two threads writing to the same memory at the same time!*"

Semaphores (1)

- **Answer:** “*True! And we will use a similar solution*”
- *semaphore:* non-negative integer synchronization variable.
  - sem_wait(s): [`while(s==0){pause();} s--;`]
  - sem_post(s): [`s++;`]
- OS guarantees that operations between brackets [ ] are executed indivisibly.
  - Only one sem_wait() or sem_post() operation at a time can modify s.
  - When `while` loop in sem_wait() terminates, only that sem_wait() can decrement s.
- Semaphore invariant: *(s >= 0)*



Semaphores (2)

- What to include:
  - #include <semaphore.h>
- Types and functions:
  - sem_t semaphore;
  - sem_init(sem_t* semPtr, int flag, int value)
    - Initialize pointed-to semaphore, with value
    - ***if flag == 1 then semaphore can be forked***
  - sem_destroy(sem_t* semPtr)
    - Destroy pointed-to semaphore. If it's negative then block.

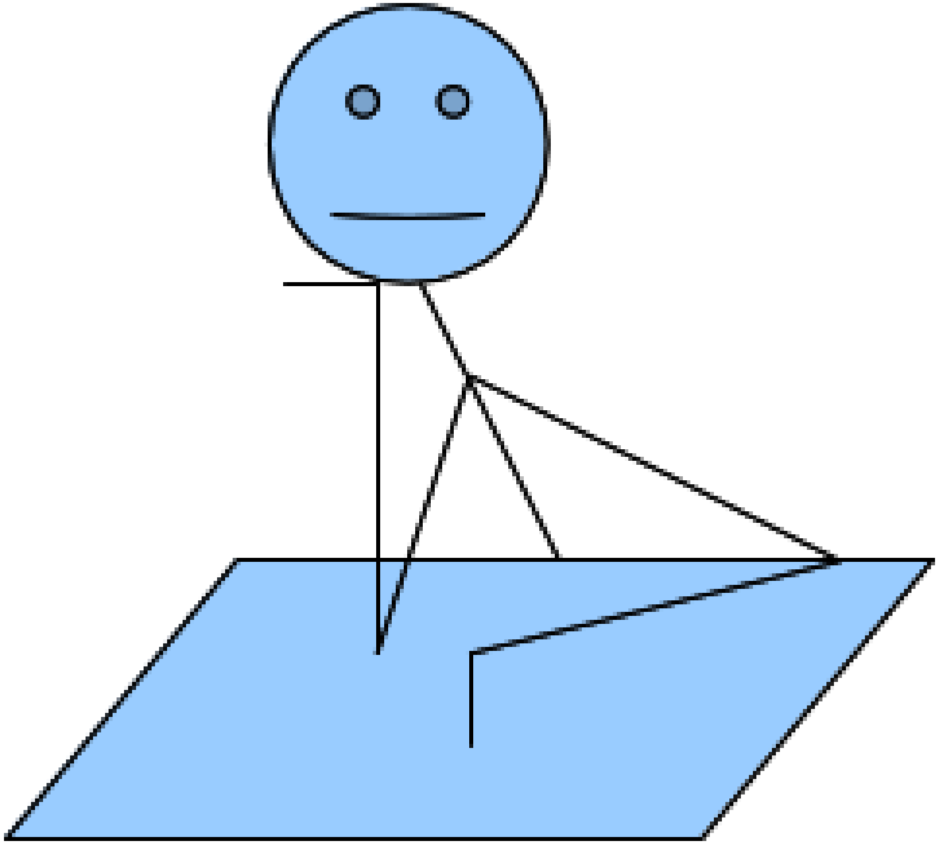
Semaphores (3)

- sem_wait(sem_t* semPtr)
  - Decrement pointed-to semaphore. If it's negative then block.
- sem_post(sem_t* semPtr)
  - Increment pointed-to semaphore. Wake one blocked process if any.
- sem_getvalue(sem_t* semPtr, int* valuePtr)
  - Get value of pointed to semaphore.

Semaphores (4)

Question "*Where shall we put the semaphore so both processes can see it?*"

Answer "*Hey, didn't we just make memory that both processes can see?*"



Semaphores in Shared Memory (1)

```
//  semaEx.h

#define  SHARED_MEM_PATH \
"/TestSMem"
#define         QUIT_CMD        \
"QUIT\n"

const int       TEXT_LEN        = 63;

struct SharedSegment
{
  sem_t         sem_;
  char          isDataAvailable_;
  char          text_[TEXT_LEN];
};
```

```
//      Compile with
//      $ g++ shareSemaProd.cpp -o shareSemaProd -lpthread -lrt

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>
#include <errno.h>

#include "semaEx.h"

void            writeIntoSegment(SharedSegment* segPtr)
{
  char  quit[sizeof(QUIT_CMD)];
  char* cPtr;

  strcpy(quit,QUIT_CMD);
  cPtr = strchr(quit,'\n');

  if  (cPtr != NULL)
    *cPtr = '\0';

  printf("Enter text (\"%s\" to quit): ",quit);
  fgets(segPtr->text_,TEXT_LEN,stdin);
  segPtr->isDataAvailable_      = 1;
}
```

Semaphores in Shared Memory (2)

```
void            waitForMyTurn   (SharedSegment* segPtr)
{
  int   isItMyTurn      = 0;

  while  (1)
  {
    sem_wait(&segPtr->sem_);
    isItMyTurn  = (segPtr->isDataAvailable_ == 0);
    sem_post(&segPtr->sem_);

    if  (isItMyTurn)
      break;
    else
      sleep(1);
  }
}


int             main            ()
{

  SharedSegment* segmentPtr;

  shm_unlink(SHARED_MEM_PATH);

  int   fd = shm_open
              (SHARED_MEM_PATH,
               O_CREAT|O_EXCL|O_RDWR,
               S_IREAD|S_IWRITE| S_IRGRP|S_IWGRP
              );

  ftruncate(fd, sizeof(SharedSegment));
```

```
  segmentPtr = (SharedSegment*)mmap(NULL,
sizeof(SharedSegment), PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);

  sem_init(&segmentPtr->sem_,1,1);
  segmentPtr->isDataAvailable_  = 0;

  do
  {
    writeIntoSegment(segmentPtr);
    waitForMyTurn(segmentPtr);
    printf("%s\n\n",segmentPtr->text_);
  }
  while  ( strncmp(segmentPtr→text_,
QUIT_CMD,sizeof(QUIT_CMD)-1) != 0);

  shm_unlink(SHARED_MEM_PATH);
  return(EXIT_SUCCESS);
}
```

Semaphores in Shared Memory (3)

```
//	Compile with:
//	$ g++ shareSemaCons.cpp -o shareSemaCons -lpthread -lrt


#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>

#include "semaEx.h"

void		writeIntoSegment(SharedSegment* segPtr)
{
  printf("Received: %s",segPtr->text_);

  for  (char* cPtr = segPtr->text_;  *cPtr != '\0';  cPtr++)
    *cPtr = toupper(*cPtr);

  printf("Sending: %s",segPtr->text_);
}
```

```
void		waitForMyTurn   (SharedSegment* segPtr)
{
  int   isItMyTurn      = 0;

  while  (1)
  {
    sem_wait(&segPtr->sem_);
    isItMyTurn  = (segPtr->isDataAvailable_);
    sem_post(&segPtr->sem_);

    if  (isItMyTurn)
      break;
    else
    {
      printf("Waiting\n");
      sleep(1);
    }
  }
}


int main()
{

  errno			= 0;
  int		fd	= shm_open
			  (SHARED_MEM_PATH,
			   O_RDWR,
			   S_IRUSR |S_IWUSR|S_IRGRP|S_IWGRP
			  );
```

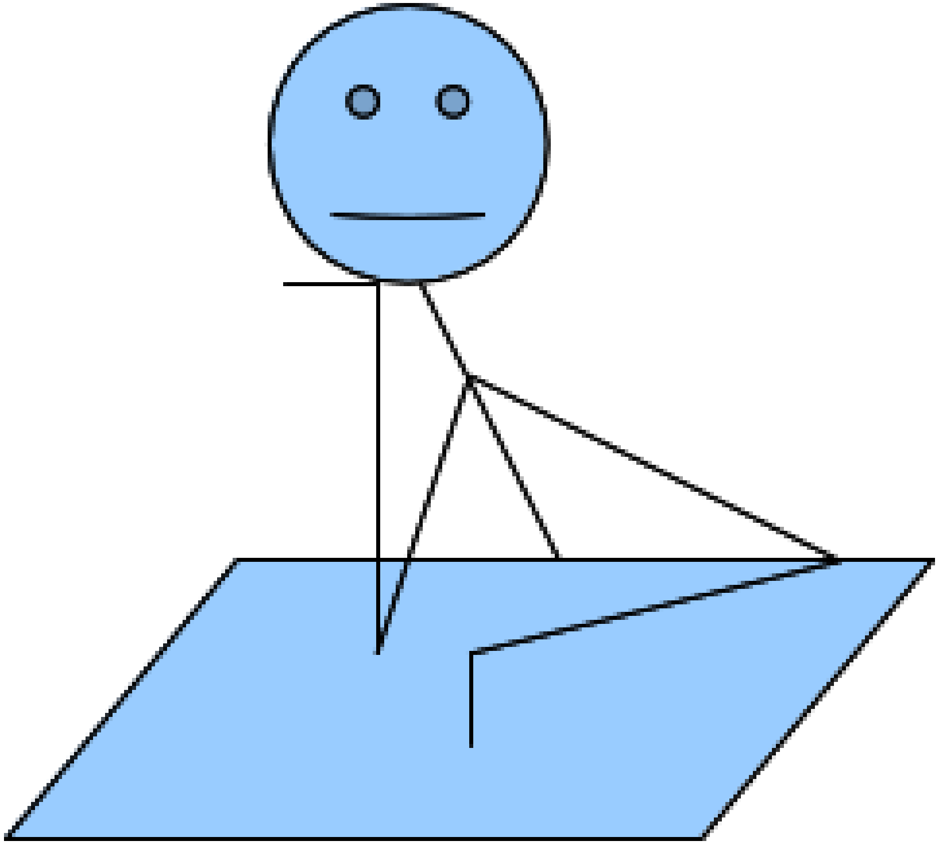
Semaphores in Shared Memory (4)

```
  SharedSegment*segPtr  = (SharedSegment*)
                          mmap(NULL,
                               sizeof(SharedSegment),
                               PROT_READ|PROT_WRITE,
                               MAP_SHARED, fd, 0);


  int   shouldQuit;

  do
  {
    waitForMyTurn(segPtr);
    writeIntoSegment(segPtr);
    shouldQuit  = (strncmp(segPtr-
>text_,QUIT_CMD,sizeof(QUIT_CMD)-1) == 0);
    segPtr->isDataAvailable_    = 0;
  }
  while (!shouldQuit);

  return(EXIT_SUCCESS);
}
```

Your turn!

Revise the last 2 programs to send math problems (like the message passing problem), but to send the answer back (like the socket pair problem).

Decisions! Decisions! Which *shall* I use?



- Remember! These are only applicable for multi-process
  - Moot for multi-thread
  - They see the same memory
- One way?
  - Pipes!
- Two way?
  - Socket pairs!
  - ***Bonus!*** Straight-forward to generalize to ordinary multi-machine sockets
- LOTS OF DATA!
  - Shared memory
- What about messages?
  - Personally, I don’t see that much of an advantage over pipes
  - If use pipes, revising to sockets is straight-forward

References:

- M. Tim Jones "*GNU/Linux Application Programming, 2nd Ed*" Course Technology, Cengage Learning. 2008
- ibm.com
- User "slezica" at stackoverflow.com