Interprocess Communication

IPC Techniques

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

Summary of Unix IPC

IPC type	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
pipes (half duplex)	●	●	●	●	●	●	●	●
FIFOs	●	●		●	●	●		
Stream pipes (full duplex)					●	●	●	●
named stream pipes					●	●	●	●
message queues		●		●	●	●		
semaphores		●		●	●	●		
shared memory		●		●	●	●		
sockets						●	●	●
streams					●	●		

Pipes

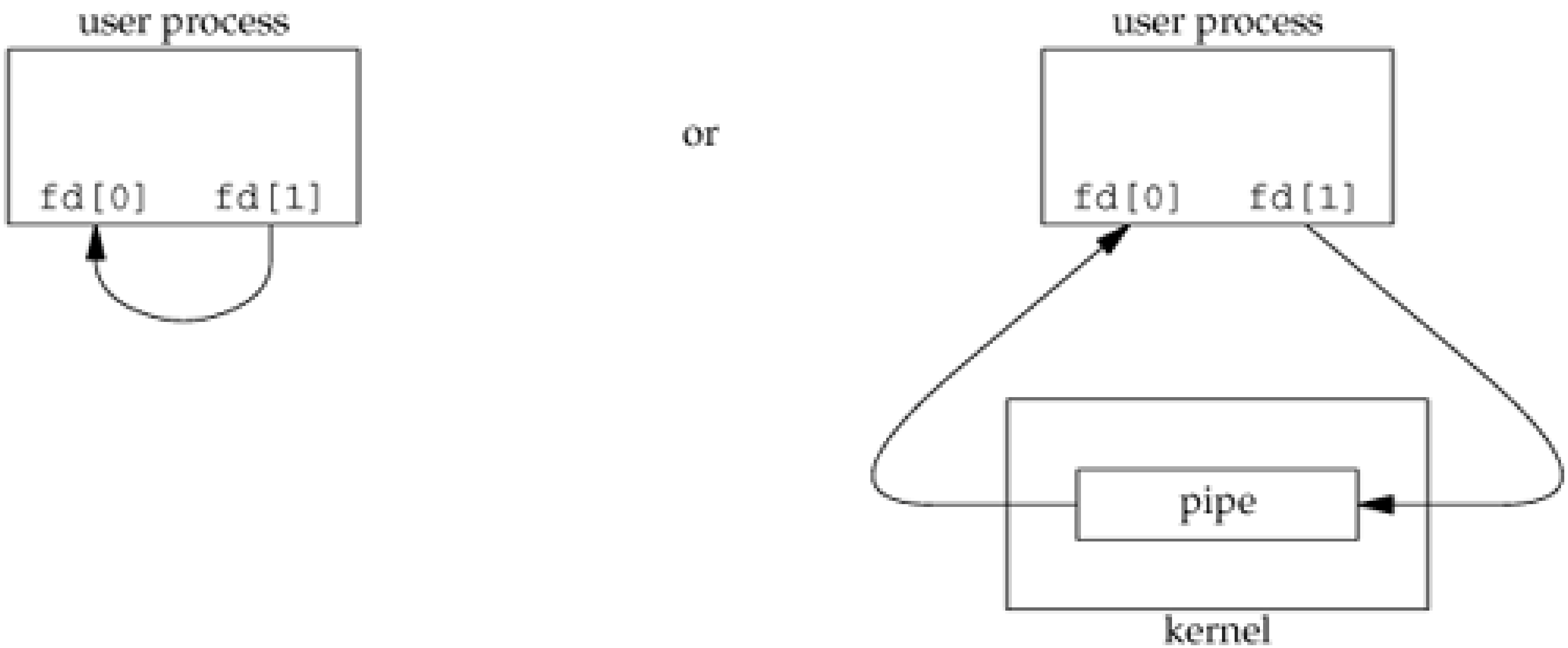
- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems
- Pipes have two limitations
  - They have been half duplex
  - Pipes can be used only between processes that have a common ancestor
    - Normally, a pipe is created by a process, that process calls fork and the pipe is used between the parent and the child
- Stream pipes (SOCK_STREAM) get around of first limitation
- FIFO gets around of 2nd limitation

Pipes

- int pipe(int filedes[2]);

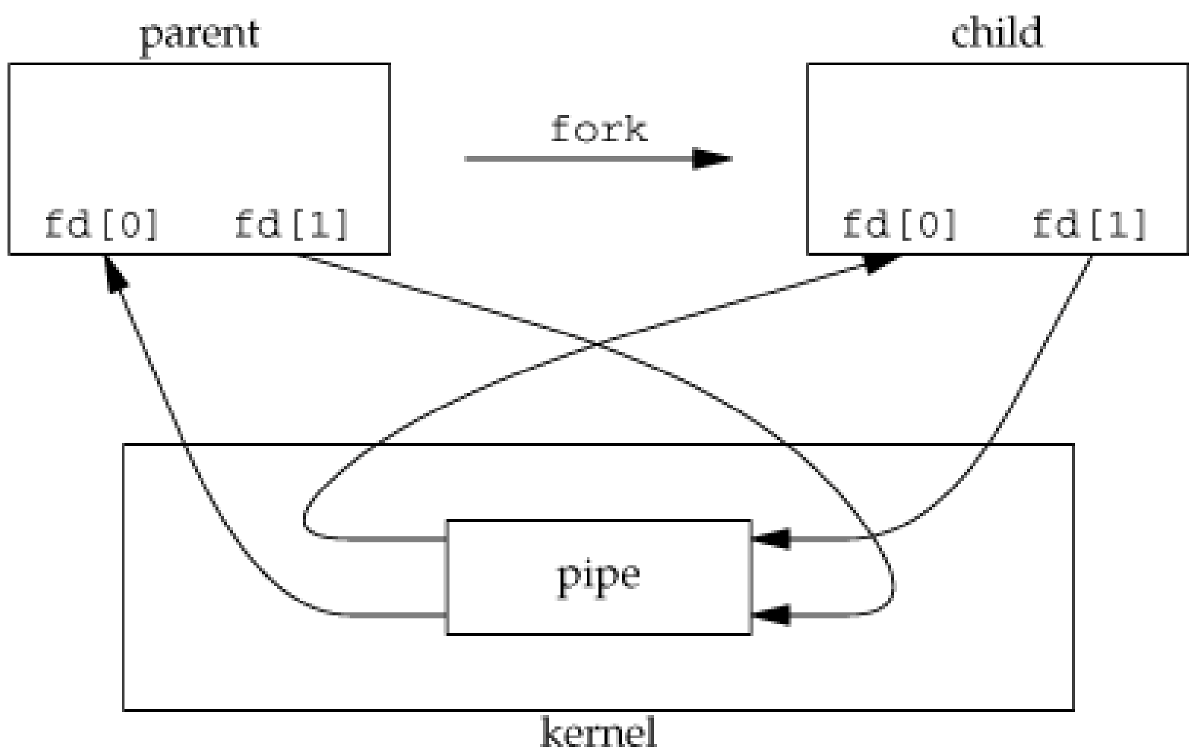
  Returns: 0 if OK, 1 on error
- filedes[0] – for reading
- filedes[1] – for writing
- fstat function returns a file type of FIFO for the file descriptor of either end of a pipe
- We can test for a pipe with S_ISFIFO macro

Two Ways to View Unix Pipes



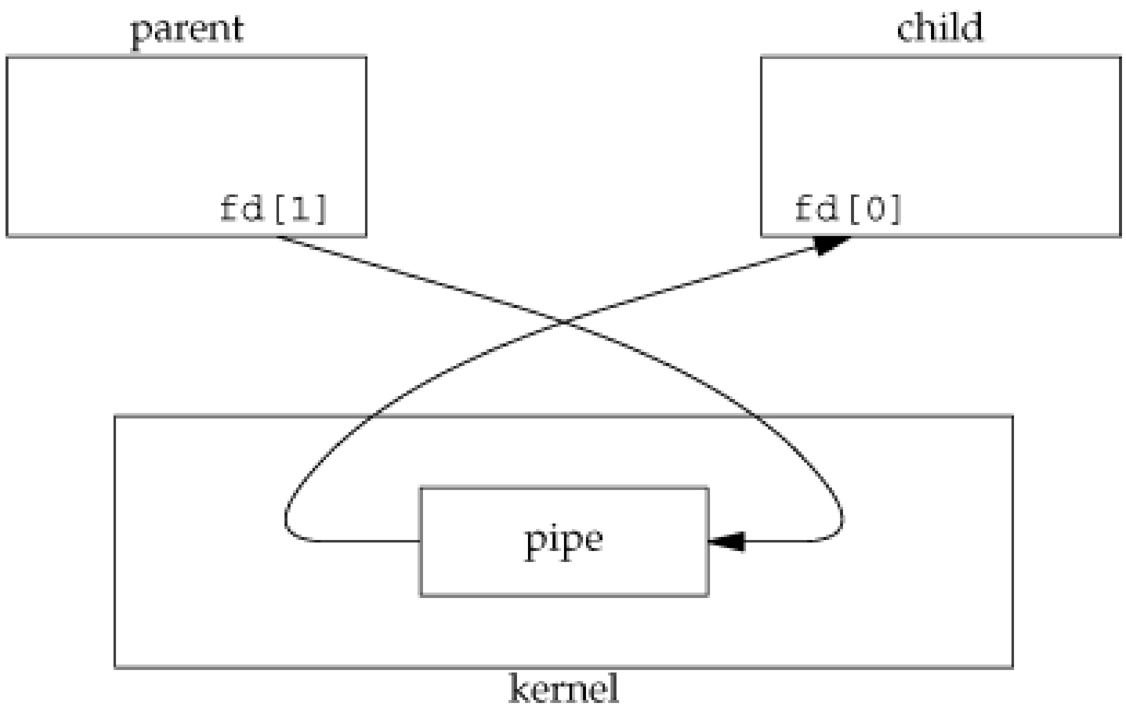
Pipe After a fork()

- A pipe in a single process is useless
- Process calls pipe, then calls fork to create IPC between parent and child



Pipe from Parent to Child

- Parent closes the read end fd[0]
- Child closes the write end fd[1]



Pipe - facts

- When one end of the pipe is closed, following rules apply
  - Read from a pipe whose write end has been closed, after all the data has been read, read returns 0 to indicate end of file
  - Write to a pipe whose read end has been closed, SIGPIPE is generated.
    - If we either ignore the signal or catch it and return from the signal handler, write returns an error with errno set to EPIPE
- PIPE_BUF, specifies kernel's pipe buffer size.

Send Data from Parent to Child over a Pipe

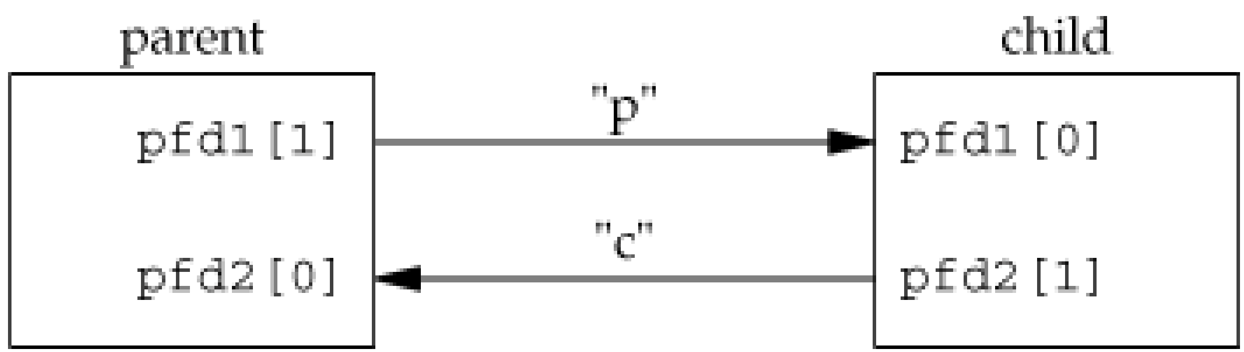
```
int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Example - “ls | wc -l

```
int main(void)
{  int pfds[2];

   pipe(pfds);

   if (!fork()) {
       close(1);         /* close normal stdout */
       dup(pfds[1]);     /* make stdout same as pfds[1] */
       close(pfds[0]); /* we don't need this */
       execlp("ls", "ls", NULL);
   } else {
       close(0);         /* close normal stdin */
       dup(pfds[0]);     /* make stdin same as pfds[0] */
       close(pfds[1]); /* we don't need this */
       execlp("wc", "wc", "-l", NULL);
   }

   return 0;
}
```

Routines to Let a Parent and Child Synchronize

- Parent writes ‘p’ on pipe1 when TELL_CHILD is called
- Child write ‘c’ on pipe2 when TELL_PARENT is called



Routines - implementation

```
static int  pfd1[2], pfd2[2];

voidTELL_WAIT(void)
{ if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void TELL PARENT(pid t pid)
{
   if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void WAIT_PARENT(void)
{
   char    c;

   if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

   if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}
```

Routines – implementation Contd…

```
void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

popen() and pclose()

- Two functions handling all the dirty works:
  - creation of a pipe
  - fork of a child
  - closing the unused ends of the pipe
  - executing a shell to execute the command
  - waiting for the command to terminate

popen()

- FILE *popen(char *cmdstring, char *type);
  Returns: file pointer if OK, NULL on error
- popen does a fork and exec to execute the cmdstring and return a standard I/O file pointer
  - *type* can be either “r” to read from child’s stdout or “w” to write to child’s stdin
  - FILE* returned is the created pipe

popen()

Result of `fp = popen(`*cmdstring*`, "r")`

FILE *rf = popen(“ls -l”, “r”);

while(fgets(buf,sizeof(buf),rf)!=NULL)

{ /* process lines */ }

Result of `fp = popen(`*cmdstring*`, "w")`

pclose function

- int pclose(FILE *fp);
  Returns: termination status of cmdstring, or -1 on error
- Closes the standard I/O stream, waits for the command to terminate and returns the termination status of the shell

Example

```
main()
{   int n;
    char line[MAXLINE];
    FILE *fp;
    fp=popen(“cat .cshrc”, “r”);
    \*read the lines in .cshrc from fp*\
    while ((fgets(line, MAXLINE, fp)) != NULL) {
            n=strlen(line);
            write(1, line, n);
    pclose(fp);
    }
}
```

Popen implementation

```
static pid_t    *childpid = NULL;
static int      maxfd;
FILE *popen(const char *cmdstring, const char *type)
{
  int       i;
  int       pfd[2];
  pid_t     pid;
  FILE      *fp;
   pipe(pfd);
   pid = fork();
   if (pid == 0) {
   if (*type == 'r') {
       close(pfd[0]);
       if (pfd[1] != STDOUT_FILENO) {
           dup2(pfd[1], STDOUT_FILENO);
           close(pfd[1]);
       }
   } else {
       close(pfd[1]);
       if (pfd[0] != STDIN_FILENO) {
           dup2(pfd[0], STDIN_FILENO);
           close(pfd[0]);
       }
   }
```

Popen implementation Contd …

```
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* parent continues... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}
```

Pclose function

```
pclose(FILE *fp)
{
    int      fd, stat;
    pid_t    pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

System V IPC functions

mechanism	function	meaning
message queues	msgget	create or access
	msgctl	control
	msgsnd	send message
	msgrcv	recieve message
semaphores	semget	create or access
	semctl	control
	semop	execute operation (wait or signal)
shared memory	shmget	create or access
	shmctl	control
	shmat	attach memory to process
	shmdt	detach memory to process

IPC Identifiers and Keys

- Each IPC structure in the kernel is referred to by a nonnegative integer identifier which is unique among all structures of that IPC (internal)
- *Identifier is obtained when the structure is created by a call to* XXXget (where XXX is one of msg, sem, or shm)
- *Identifier is used by all other IPC functions to reference this* structure

ftok()

- Processes need to share IPC structures and they cannot know the identifier ahead of time
- Whenever an IPC structure is created or accessed by XXXget(), a *key is specified as an argument*
- Type of *key is key_t* (a long integer) and is converted by the kernel into an *identifier*
- key_t ftok(const char **path*, int *id*);
  - Success, returns a key, failure returns -1
- ftok() return a key based on *path*name of an existing file and *id* which is used in subsequent calls to [*msgget()*](), [*semget()*](), and [*shmget()*]()

ftok () Example

- key = ftok("/home/beej/somefile", 'E');
- semid = semget(key, 10, 0666 | IPC_CREAT);

Various ways for Client-server to Specify Key

- Server creates a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier somewhere (a file) for the client to obtain
  - IPC_PRIVATE guarantees that the server creates a new IPC structure
  - Disadvantage: file system operations are required for the server to write the identifier to file and for the clients to retrieve this identifier later
- IPC_PRIVATE key is used in a parent-child relationship
  - Parent creates a new IPC structure specifying IPC_PRIVATE and the resulting identifier is then available to the child after the fork
  - Child can pass the identifier to a new program as an argument to one of the exec functions

Various ways for Client-server to Specify Key

- Client and the server agree on a key by defining the key in a common header, for example
  - Server creates a new IPC structure specifying this key
  - Problem: it's possible for the key to already be associated with an IPC structure,
    - This case, get function (msgget, semget, or shmget) returns an error
    - Server must handle this error, deleting the existing IPC structure and try to create it again
- Client and server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function ftok to convert these two values into a key

Comparison of different forms of IPC

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Message Queue

- Message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A new queue is created or an existing queue opened by msgget()
- New messages are added to end of a queue by msgsnd()
- Every message of msgsnd() has a positive long integer *type* field, a non-negative length and the actual data bytes
- Messages are fetched from a queue by msgrcv().
  - don't have to fetch the messages in a first-in, first-out order. Instead, can fetch messages based on *type* field

Message queue structure

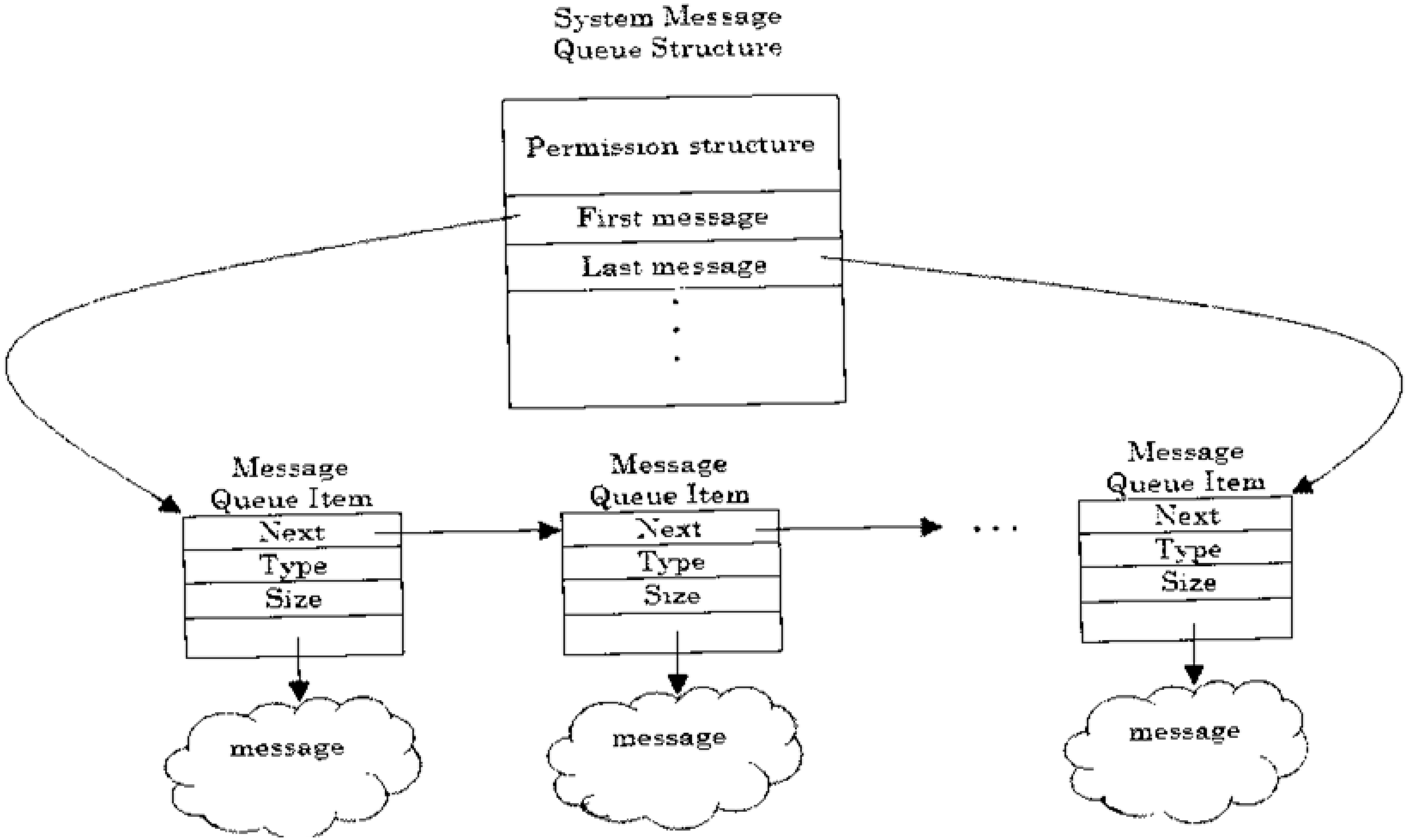
- Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see                  */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  .
  .
  .
};
```

Permission Structure

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user id */
  gid_t  gid;  /* owner's effective group id */
  uid_t  cuid; /* creator's effective user id */
  gid_t  cgid; /* creator's effective group id */
  mode_t mode; /* access modes */
   .
   .
   .
};
```

Message Queue Structure



System Limits that Affect Message Queues

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Size in bytes of largest message we can send	16,384	8,192	notsup	2,048
The maximum size in bytes of a particular queue (i.e., the sum of all the messages on the queue)	2,048	16,384	notsup	4,096
The maximum number of messages queues, systemwide	40	16	notsup	50
The maximum number of messages, systemwide	40	derived	notsup	40

msgget()

- int msgget(key_t key, int flag);
  Returns: message queue ID if OK, -1 on error
- A new MQ is created if key is IPC_PRIVATE or for a non existing MQ key and the IPC_CREAT bit of flag is specified
- To open an existing MQ, key must equal the key that was specified when the MQ was created and IPC_CREAT must not be specified
- To make sure that a new MQ is created, specify a flag with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the MQ already exists
- When a new MQ is created
  - The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag
  - msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
  - msg_ctime is set to the current time
  - msg_qbytes is set to the system limit

msgctl()

- int msgctl(int msqid, int cmd, struct msqid_ds *buf);
  Returns: 0 if OK, -1 on error
- The cmd specifies the command to be performed by msqid.
  - IPC_STAT
    - Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf
  - IPC_SET
    - Copy msg_perm.uid, msg_perm.gid, msg_perm.mode and msg_qbytes fields from the structure pointed by the msqid_ds structure to buf
    - can be executed only by a process whose effective user ID equals msg_perm.cuid or msg_perm.uid or by a process with super user privileges
    - Only the super user can increase the value of msg_qbytes
  - IPC_RMID
    - Remove the message queue from system and any data still on the queue
    - removal is immediate
    - Any other process still using the message queue will get an error of EIDRM on its next attempted operation on the queue
    - can be executed only by a process whose effective user ID equals msg_perm.cuid or msg_perm.uid or by a process with super user privileges.

msgsnd()

- int msgsnd(int msqid, void *ptr, size_t nbytes, int flag);

  Returns: 0 if OK, -1 on error
- ptr argument points to a long integer that contains the positive integer message type and it is immediately followed by the message data
  - struct mymesg {

    long mtype; /* positive message type */

    char mtext[512]; /* message data, of length nbytes */ };
- Flag value of IPC_NOWAIT can be specified for nonblocking, if MQ is full

msgrcv()

- ssize_t msgrcv(int msqid, void *ptr, size_t nbytes , long type, int flag);
  Returns: size of data portion of message if OK, -1 on error
- type argument specifies which message to retrieve
  - type == 0 : the first message on the queue is returned
  - type > 0: The first message on the queue whose message type equals type is returned
  - type < 0: The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned
- Flag of IPC_NOWAIT to be nonblocking, causing msgrcv to return -1 with errno set to ENOMSG if a message of the specified type is not available.
- If IPC_NOWAIT is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system, or a signal is caught and the signal handler returns

Creating and Sending to a Message Queue

```
#define MSGSZ 128
type def struct msgbuf
{ long mtype;
  char mtext[MSGSZ]; }  message_buf;
main()
{
      int msqid;
      int msgflg = IPC_CREAT | 0666;
      key_t key; message_buf sbuf;
      size_t buf_length;
      key = 1234;
      msqid = msgget(key, msgflg );
      /* * We'll send message type 1 */
      sbuf.mtype = 1;
      strcpy(sbuf.mtext, "Did you get this?");
      buf_length = strlen(sbuf.mtext) + 1 ;
      msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) ;
}
```

Receiving the Above Message

```
#define MSGSZ 128
type def struct msgbuf
{ long mtype;
  char mtext[MSGSZ]; }  message_buf;
main()
{
     int msqid;
     key_t key = 1234;
     message_buf rbuf;
     msqid = msgget(key, 0666);
     /* * Receive an answer of message type 1. */
     msgrcv(msqid, &rbuf, MSGSZ, 1, 0);
     printf("%s\n", rbuf.mtext);
}
```

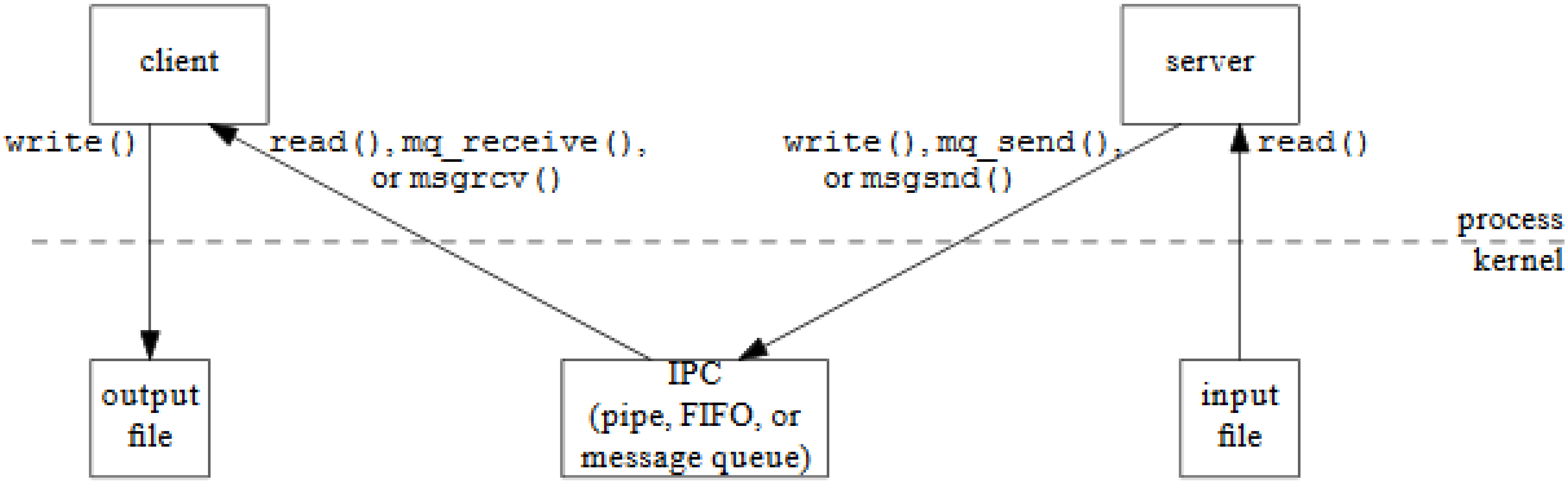
Client–Server Using Single Message Queue

- A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient
- For example, the server receives only the messages with a type field of 1 and the clients receive only the messages with a type field equal to their process IDs
- Clients send their requests with a type field of 1. Request must include the client's process ID
- Server then sends the response with the type field set to the client's process ID

Client–Server using Multiple Message Queue

- Individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE
- Server also has its own queue, with a key or identifier known to all clients
- Client sends its first request to the server's well-known queue and this request must contain the message queue ID of the client's queue
- Server sends its responses to the client's queue
- Problems with this technique :
  - Each client-specific queue usually has only a single message on it. This seems wasteful of a limited system wide resource
  - Server has to read messages from multiple queues. Neither select nor poll works with message queues

Client–server with pipe, FIFO or MQ



Client–server with pipe, FIFO or MQ

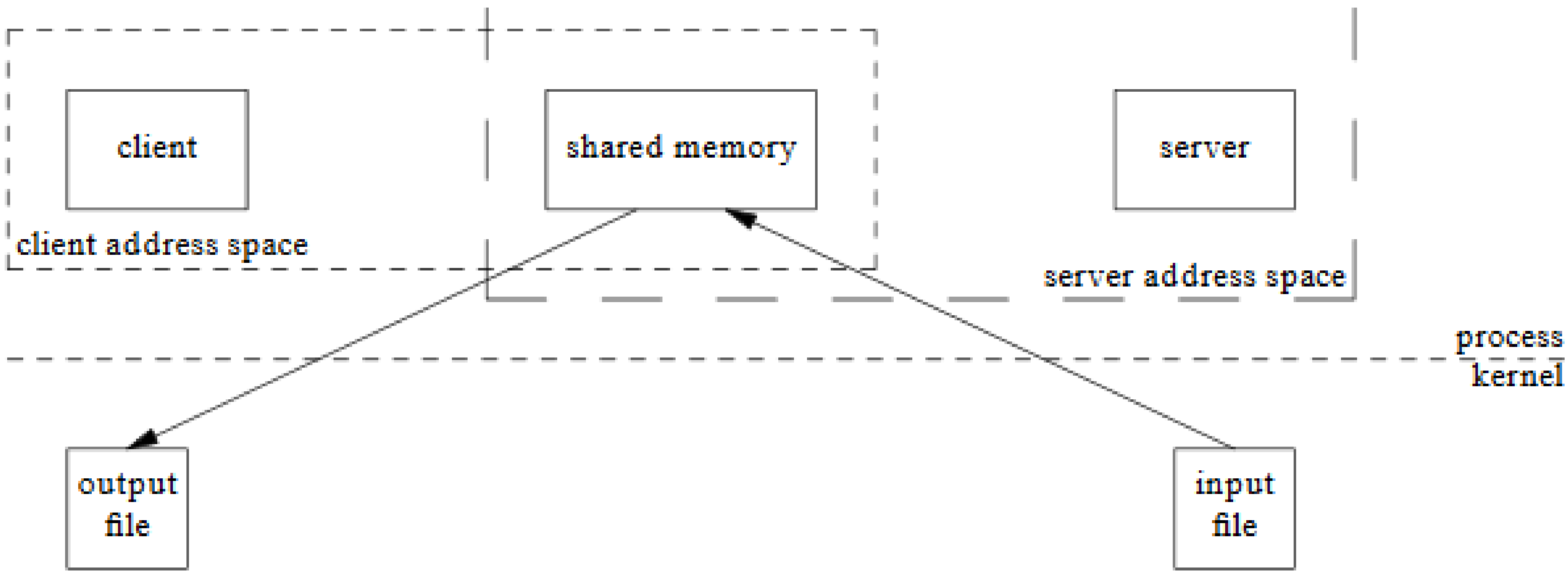
- Server reads from input file, data is read by kernel into its memory and then copied to process
- Server writes this data in a message, using a pipe, FIFO, or message queue
- Client reads data from IPC channel, requiring data to be copied from kernel to process
- Finally, data is copied from client’s buffer (second argument to the write function), to output file
- A total of four copies of data are done between kernel and a process

Shared Memory

- Two processes to exchange information using Pipes, FIFOs and message queues, information has to go through the kernel
- Shared memory provides a way around this by allowing two or more processes to share a given region of memory
- If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done
- Synchronizing access to a given region among multiple processes is done using semaphores

Client and Server with Shared Memory

- Shared memory appears in the address space of both the client and the server



Client and Server with Shared Memory

- The fastest form of IPC, because the data does not need to be copied between client and server
- Server gets access to a shared memory using a semaphore
- Server reads from input file into shared memory
  - Second argument to the read (address of the data buffer) points into the shared memory
- When read is completed, server notifies client using a semaphore
- Client writes data from shared memory to output file
- Data is copied only twice — from the input file into shared memory and from shared memory to the output file

shmid_ds

- Kernel maintains a structure with at least following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

System Limits that Affect Shared Memory

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum size in bytes of a shared memory segment	33,554,432	33,554,432	4,194,304	8,388,608
The minimum size in bytes of a shared memory segment	1	1	1	1
The maximum number of shared memory segments, systemwide	192	4,096	32	100
The maximum number of shared memory segments, per process	128	4,096	8	6

shmget

- int shmget(key_t key, size_t size, int flag); Returns: shared memory ID if OK, –1 on error
- A new segment is created or an existing segment is referenced
- When a new segment is created, the following members of the shmid_ds structure are initialized.
  - ipc_perm structure is initialized
    - mode member of this structure is set to the corresponding permission bits of flag
  - shm_lpid, shm_nattach, shm_atime, and shm_dtime are all set to 0.
  - shm_ctime is set to the current time.
  - shm_segsz is set to the size requested.

shmctl

- int shmctl(int shmid, int cmd, struct shmid_ds *buf);

  Returns: 0 if OK, –1 on error
- cmd argument specifies one of the following five commands:
  - IPC_STAT: Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.
  - IPC_SET: Set shm_perm.uid, shm_perm.gid, and shm_perm.mode fields from the structure pointed to by buf
    - Can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.
  - IPC_RMID: Remove the shared memory segment set from the system
    - Since an attachment count is maintained for shared memory, the segment is not removed until the last process using the segment terminates or detaches it
    - Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment
    - command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges
  - SHM_LOCK :Lock the shared memory segment in memory
    - Command can be executed only by the superuser.
  - SHM_UNLOCK Unlock the shared memory segment
    - Command can be executed only by the superuser.

shmat

- void *shmat(int shmid, const void *addr, int flag); Returns: pointer to shared memory segment if OK, –1 on error
- A process attaches a shared memory to its address space
- Depends on the addr argument and the SHM_RND ("round“) bit in flag
  - If addr is 0, the segment is attached at the first available address selected by the kernel.
    - This is the recommended technique
  - If addr is nonzero and SHM_RND is not specified, the segment is attached at the address given by addr
  - If addr is nonzero and SHM_RND is specified, the segment is attached at the address given by (addr - (addr modulus SHMLBA))
  - SHMLBA stands for "low boundary address multiple" and is always a power of 2
  - What the arithmetic does is round the address down to the next multiple of SHMLBA

shmat()

- If the SHM_RDONLY bit is specified in flag, the segment is attached read-only. Otherwise, the segment is attached read–write.
- If shmat succeeds, the kernel will increment the shm_nattch counter in the shmid_ds structure associated with the shared memory segment

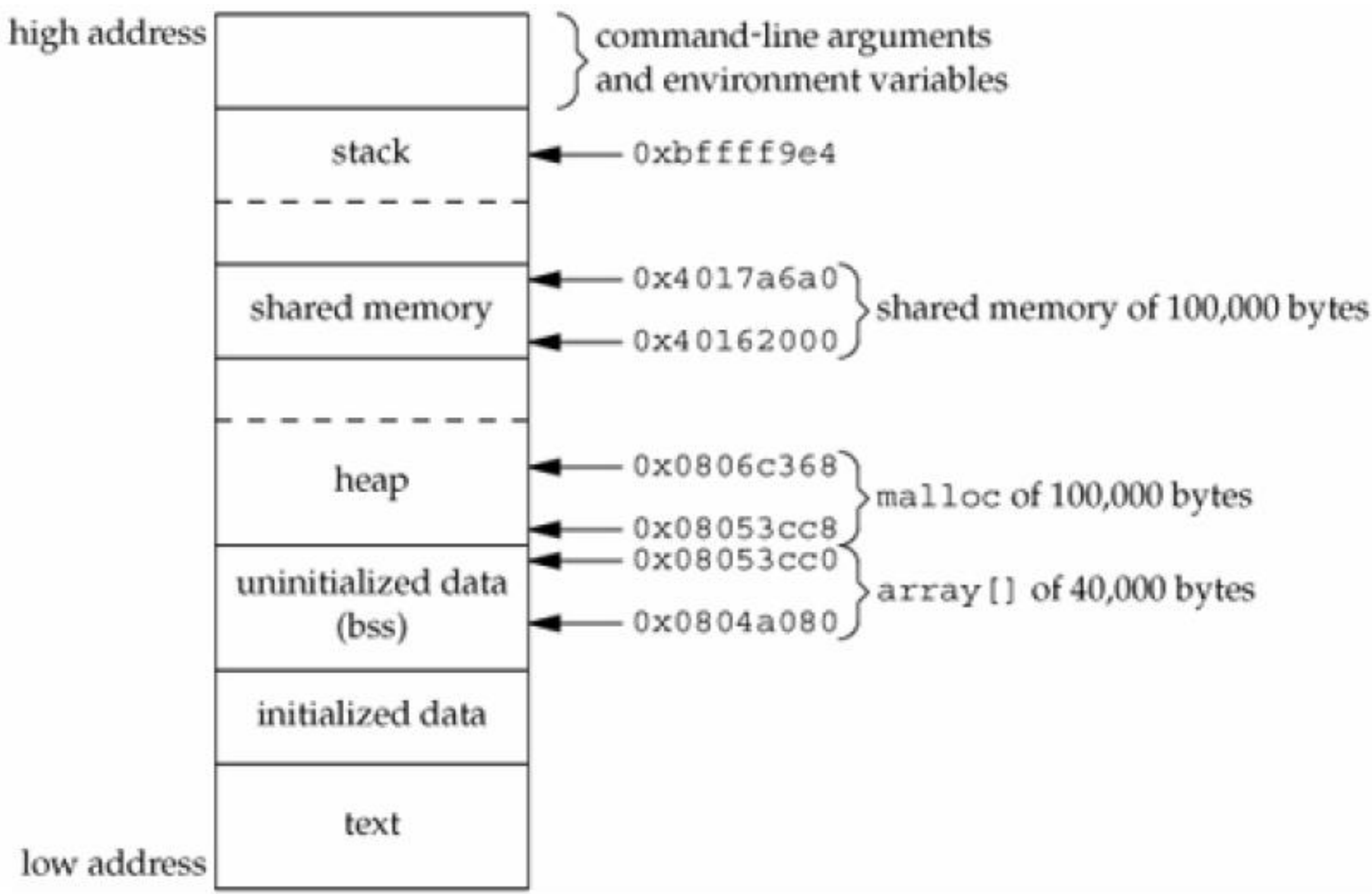
shmat and shmdt

- int shmdt(void *addr); Returns: 0 if OK, –1 on error
- Call shmdt to detach it
- Does not remove the identifier and its associated data structure from the system
- If successful, shmdt will decrement the shm_nattch counter in the associated shmid_ds structure

Print where various types of data are stored

```
#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 /* user read/write */
char array[ARRAY_SIZE]; /* uninitialized data = bss */
Int main(void)
{
    int shmid;
    char *ptr, *shmptr;
    printf("array[] from %lx to %lx\n", (unsigned long)&array[0], (unsigned
    long)&array[ARRAY_SIZE]);
    ptr = malloc(MALLOC_SIZE);
    printf("malloced from %lx to %lx\n", (unsigned long)ptr, (unsigned
    long)ptr+MALLOC_SIZE);
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE);
    shmptr = shmat(shmid, 0, 0);
    printf("shared memory attached from %lx to %lx\n", (unsigned long)shmptr, (unsigned
    long)shmptr +SHM_SIZE);
    shmctl(shmid, IPC_RMID, 0);
    exit(0);
}
```

Memory layout on an Intel-based Linux system



shm_server.c

```
#define SHMSZ     27
main()
{
key_t key 5678;;
   int shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
   char *shm = shmat(shmid, NULL, 0);
   /* Now put some things into the memory for the  other process to read.      */
  char *s = shm;

   for (char c = 'a'; c <= 'z'; c++)
      *s++ = c;
   *s = NULL;
   /*wait until the client process  changes the first character of our memory to '*',
     indicating that it has read what  we put there.   */
   while (*shm != '*')
      sleep(1);

   exit(0);
}
```

shm_client.c

```
#define SHMSZ     27
main()
{
   int shmid;
   key_t key = 5678;
   char *shm, *s;
  shmid = shmget(key, SHMSZ, 0666);
  shm = shmat(shmid, NULL, 0)

/ * Now read what the server put in the memory.      */
   for (s = shm; *s != NULL; s++)
       putchar(*s);
   putchar('\n');

   /* Finally, change the first character of the segment to '*', indicating we have read the segment.      */
   *shm = '*';

   exit(0);
}
```

Shared Memory Features

- Advantages
  - Random Access
    - can update a small piece in the middle of a data structure, rather than the entire structure
  - Efficiency
    - unlike message queues and pipes, which copy data from the process *into* memory within the kernel, shared memory is directly accessed
    - Shared memory resides in the user process memory and is then shared among other processes
- Disadvantages
  - No automatic synchronization as in pipes or message queues
  - Have to provide synchronization with *semaphores* or signals
  - Must remember that pointers are only valid within a given process. Thus, pointer offsets cannot be assumed to be valid across inter-process boundaries. This complicates the sharing of linked lists or binary trees

Semaphores

- Semaphore is a counter used to provide access to a shared resource between multiple processes
- To obtain a shared resource, a process
  - Test the semaphore that controls the resource
  - If the value of the semaphore is positive, the process can use the resource
    - Process decrements the semaphore value by 1, indicating that it has used one unit of the resource
  - Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0.
    - When the process wakes up, it returns to step 1.
- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.
- If any other processes are asleep, waiting for the semaphore, they are awakened

Semaphore (contd)

- To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel
- A common form of semaphore is binary semaphore. It controls a single resource and its value is initialized to 1

XSI Semaphores

- XSI semaphores are more complicated by the features:
  - Defines a semaphore as a set of one or more semaphore values. When a semaphore is created, need to specify the number of values in the set
  - Creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
  - Since all forms of XSI IPC remain in existence even when no process is using them, a program may terminate without releasing the semaphores it has been allocated

semid_ds Structure

```
struct semid_ds {
   struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
   unsigned short   sem_nsems; /* # of semaphores in set */
   time_t           sem_otime; /* last-semop() time */
   time_t           sem_ctime; /* last-change time */
   .
   .
   .
};
```

Sem Structure

- Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct  sem{
    ushort semval; /* semaphore value, always >= 0 */
    pid_t sempid; /* pid for last operation */
    ushort semncnt; /* # processes awaiting semval>curval */
    ushort semzcnt; /* # processes awaiting semval==0 */
};
```

System Limits that Affect Semaphores

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum value of any semaphore	32,767	32,767	32,767	32,767
The maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	16,384
The maximum number of semaphore sets, systemwide	10	128	87,381	10
The maximum number of semaphores, systemwide	60	32,000	87,381	60
The maximum number of semaphores per semaphore set	60	250	87,381	25
The maximum number of undo structures, systemwide	30	32,000	87,381	30
The maximum number of undo entries per undo structures	10	32	10	10
The maximum number of operations per `semop` call	100	32	100	10

semget

- int semget(key_t key, int nsems, int flag);
  Returns: semaphore ID if OK, -1 on error
- Number of semaphores in the set is nsems.
  - Must specify nsems if a new set is being created
  - Can specify nsems as 0 if referencing an existing set
- When a new set is created, following members of the semid_ds structure are initialized
  - ipc_perm structure is initialized
    - mode member is set to permission bits of flag
  - sem_otime is set to 0
  - sem_ctime is set to the current time
  - sem_nsems is set to nsems

Semaphore Example (1)

```
#define MAX_RETRIES 10
int main(void)
{
    key_t key = ftok("semdemo.c", 'J');
    int semid = semget(key, 1, IPC_CREAT | IPC_EXCL | 0666);
```

semctl

- int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
  Returns: (see following)
- Fourth argument is optional, depending on the command requested
- union semun {

```
int val; /* for SETVAL */
struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
unsigned short *array; /* for GETALL and SETALL */
}arg;
```

semctl (contd)

- cmd argument:
  - IPC_STAT - Fetch the semid_ds structure for this set, storing it in the arg.buf.
  - IPC_SET - Set the sem_perm.uid, sem_perm.gid and sem_perm.mode fields from the arg.buf in the semid_ds
    - command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or with superuser privileges
  - IPC_RMID - Remove the semaphore set from the system.
    - Removal is immediate
    - Any other process still using the semaphore will get an error of EIDRM on its next attempted operation on the semaphore
    - Command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or with superuser privileges

semctl (contd)

- cmd argument (Contd):
  - GETVAL - Return the value of semval for the member semnum
  - SETVAL - Set the value of semval for the member semnum
    - The value is specified by arg.val
  - GETPID - Return the value of sempid for the member semnum
  - GETNCNT - Return the value of semncnt for the member semnum
  - GETZCNT - Return the value of semzcnt for the member semnum
  - GETALL - Fetch all the semaphore values in the set and are stored in the arg.array.
  - SETALL - Set all the semaphore values in the set to the values pointed to by arg.array.

Semaphore Example (2)

```
#define MAX_RETRIES 10
int main(void)
{
   key_t key = ftok("semdemo.c", 'J');
   int semid = semget(key, 1, IPC_CREAT | IPC_EXCL | 0666);

//Initialize the semahore
   if (semid >= 0)
   {    union semun {
            int val;
            struct semid_ds *buf;
            ushort *array;
      } arg;
     arg.val = 0; //Initializing semval to 0
     semctl(id, 0, SETVAL, arg);
```

semop

- int semop(int semid, struct sembuf semoparray[], size_t nops);
  Returns: 0 if OK, -1 on error
- struct sembuf {
  ushort sem_num; /* member # in set (0, 1, ..., nsems-1) */
  short sem_op; /* operation (negative, 0 or positive) */
  short sem_flg; /* IPC_NOWAIT */

};

- Semop() atomically performs an array of operations on a semaphore set
- Operation is specified by sem_op value. This value can be negative, 0, or positive.

semop (contd)

sem_op	**What happens**
Negative	Allocate resources
Positive	Release resources
Zero	Process will wait until the semaphore reaches 0.

semop (contd)

- When sem_op is positive, corresponds to the returning of resources by the process
- Value of sem_op is added to the semaphore's value

semop (contd)

- If sem_op is negative, want to obtain resources that the semaphore controls
  - If the semaphore's value is less than the absolute value of sem_op (the resources are not available):
    - If IPC_NOWAIT is specified, semop returns with an error of EAGAIN.
    - If IPC_NOWAIT is not specified, the semncnt value for this semaphore is incremented and the calling process is suspended until:
      - The semaphore's value becomes greater than or equal to the absolute value of sem_op (i.e., some other process has released some resources). The value of semncnt for this semaphore is decremented and the absolute value of sem_op is subtracted from the semaphore's value
      - Semaphore is removed from the system and the function returns an error of EIDRM.
      - A signal is caught by the process, and the signal handler returns, the value of semncnt for this semaphore is decremented and the function returns an error of EINTR

semop (contd)

- If sem_op is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.
  - If the semaphore's value is currently 0, the function returns immediately
  - If the semaphore's value is nonzero:
    - If IPC_NOWAIT is specified, return with an error of EAGAIN.
    - If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented and the calling process is suspended until:
      - The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented
      - The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
      - A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented and the function returns an error of EINTR.

Semaphore Example (3)

```
#define MAX_RETRIES 10
int main(void)
{
   key_t key = ftok("semdemo.c", 'J');
   int semid = semget(key, 1, IPC_CREAT | IPC_EXCL | 0666);
   if (semid >= 0)
   {    union semun {
            int val;
            struct semid_ds *buf;
            ushort *array;
      } arg;
     arg.val = 0;//Initializing semval to 0
     semctl(id, 0, SETVAL, arg);

     // to make semaphore ready  by allocating 1 resource
     struct sembuf sb;
     sb.sem_num = 0; sb.sem_op = 1; sb.sem_flg = 0;
     semop(semid, &sb, 1); /* this sets the sem_otime field */
   }
```

Process A

```
key_t key = ftok("semdemo.c", 'J');
int semid = semget(key, 1, 0);
 printf("Trying to lock...\n");
 struct sembuf sb;
 sb.sem_num = 0;
 sb.sem_op = -1;  /* set to allocate resource */
 sb.sem_flg = SEM_UNDO;
if (semop(semid, &sb, 1) == -1)
    { perror("semop");     exit(1);   }
else
     printf("Locked.\n");
```

Process B

```
key_t key = ftok("semdemo.c", 'J');
int semid = semget(key, 1, 0);
printf("Trying to unlock...\n");

struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = 1; /* free resource */

if (semop(semid, &sb, 1) == -1)
    { perror("semop");       exit(1); }
else
   printf("Unlocked\n");
```

```
//removing the semaphore when you're done:
 key_t key;
 int semid;
union semun arg;
key = ftok("semdemo.c", 'J');
semid = semget(key, 1, 0);
semctl(semid, 0, IPC_RMID, arg);
```

Persistence of IPC

- lifetime of the interprocess communication mechanism – 3 types
  - **Process-persistence**: The mechanism lasts until all the processes that have opened the mechanism close it, exit or crash
  - **Kernel-persistence**: The mechanism exists until the kernel of the operating system reboots or the mechanism is explicitly deleted
  - **Filesystem-persistence**: The mechanism exists until the mechanism is explicitly deleted

Persistence of IPC

Mechanism	Persistence
Shared memory	Kernel
Process-shared semaphore	Process
Message queue	Kernel
files, FIFOs	Filesystem
pipes, sockets	Process
Memory mapped file	Filesystem

Other IPC Mechanisms

- Pipe, FIFO, Shared Memory, Message Queue, Semaphore – within same machine
- Sockets – used also for network communication
- Memory mapped files - to read and write to and from files so that the information is shared between processes
  - map a section of the file to memory, and get a pointer to it by using the **mmap()** system call