BITS Pilani
Pilani Campus

Network Programming

K Hari Babu
Department of Computer Science & Information Systems

Outline

- System V Message Queues
- System V Semaphores
- System V Shared Memory

innovate achieve lead

BITS Pilani
Pilani Campus

Overview of System V IPC (R1: Ch45)

System V IPC

- It refers to three IPC mechanisms:
  - Message Queues
    - Used to pass messages between two processes.
    - Maintains boundaries between messages.
    - Messages can received out of order or in FIFO order.
  - Semaphores
    - Permit multiple processes to synchronize their actions.
    - An integer maintained by kernel.
    - Process indicates to its peers that it is performing some action by modifying the value of semaphore.
  - Shared memory
    - Enables multiple processes to share the same region of memory by mapping process address space.
    - Quickest method of IPC

API Overview

- All three mechanisms have similar API.

Interface	**Message queues**	**Semaphores**	**Shared memory**
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Associated data structure	*msqid_ds*	*semid_ds*	*shmid_ds*
Create/open object	*msgget()*	*semget()*	*shmget()* + *shmat()*
Close object	(none)	(none)	*shmdt()*
Control operations	*msgctl()*	*semctl()*	*shmctl()*
Performing IPC	*msgsnd()*—write message *msgrcv()*—read message	*semop()*—test/adjust semaphore	access memory in shared region

System V IPC API

- Three stages
  - Create/open IPC object
  - Perform operations
  - Close/remove the object
- Create/open IPC object
  - Just like open() for files, there is *msg/sem/shmget()* calls for creating objects.
  - get() call returns an identifier which is required for operations.
    - unlike fd, this id is globally visible. If we know id, we can directly operate on the object without even opening the object.
    - Example

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
    //Returns: message queue ID if OK, 1 on error
```

    - Same syntax for semget() and shmget().

IPC Keys

- Keys are integer values. IPC *get()* calls translate a key into the corresponding integer IPC identifier.
    - Unique mapping between keys and ids.
- How do we get unique key?
    - Random value included in header file shared by multiple programs.
    - IPC_PRIVATE as key: kernel creates unique id.
        - Useful parent-child communication.
    - Use *ftok()* to generate a likely unique key.

```
#include <sys/ipc.h>
key_t ftok(char * pathname , int  proj );
/*Returns integer key on success, or -1 on error*/
```

- Purpose of *proj* value is to generate multiple IPC objects using the same file.
- Lower 8 bits from proj, lower 8 bits from minor device number, lower 16 bits from inode of the file. Collision very less likely.

Create/open IPC object

```c
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1)
    errExit("msgget");
```

- Second argument: creation flags + permissions
- Creation flags:
  - IPC_CREAT: specified when a new object is to be created. If the object already exists, its id is returned.
  - IPC_CREAT | IPC_EXCL: if the object already exists, error EEXIST is returned.
- Permissions:
  - Execute permission is not valid. Owner can be changed.

```c
struct ipc_perm {
    key_t          __key;/* Key, as supplied to 'get' call */
    uid_t          uid;  /* Owner's user ID */
    gid_t          gid;  /* Owner's group ID */
    uid_t          cuid; /* Creator's user ID */
    gid_t          cgid; /* Creator's group ID */
    unsigned short mode; /* Permissions */
    unsigned short __seq;/* Sequence number */
};
```

Listing All IPC Objects

- *ipcs* and *ipcrm* commands can be used to list and remove IPC objects respectively.
- *ipcs*: displays objects that the user has read permission

```
$ ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x6d0731db 262147     mtk        600        8192        2
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x6107c0b8 0          cecilia    660        6
0x6107c0b6 32769      britta     660        1
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x71075958 229376     cecilia    620        12           2
```

- *ipcrm*

```
$ ipcrm -s 65538
```

  - Use *q, s, and m* for queues, semaphores, and share memory respectively.

IPC Limits

- Kernel imposes limits on resources for IPC objects.
- We can use *ipcs –l* option to list the limits.
- Limits can viewed/modified using /proc/sys/kernel files.

```
$ ipcs -l
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767

------ Messages Limits --------
max queues system wide = 3970
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

innovate achieve lead

BITS Pilani
Pilani Campus

System V Message Queues

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
  - Any process with adequate privileges can place the message into the queue and any process with adequate privileges can read from queue
- Differences with FIFO
  - There is no requirement that some process must be waiting to receive message before sending the message.
  - Maintains boundaries between messages.
    - Not possible to read a partial message.
    - Can't read multiple messages in a single call.
  - Messages can be retrieved in other than FIFO order.
  - Even if all processes referencing message queue terminate, still the message queue exists.

Message Queues

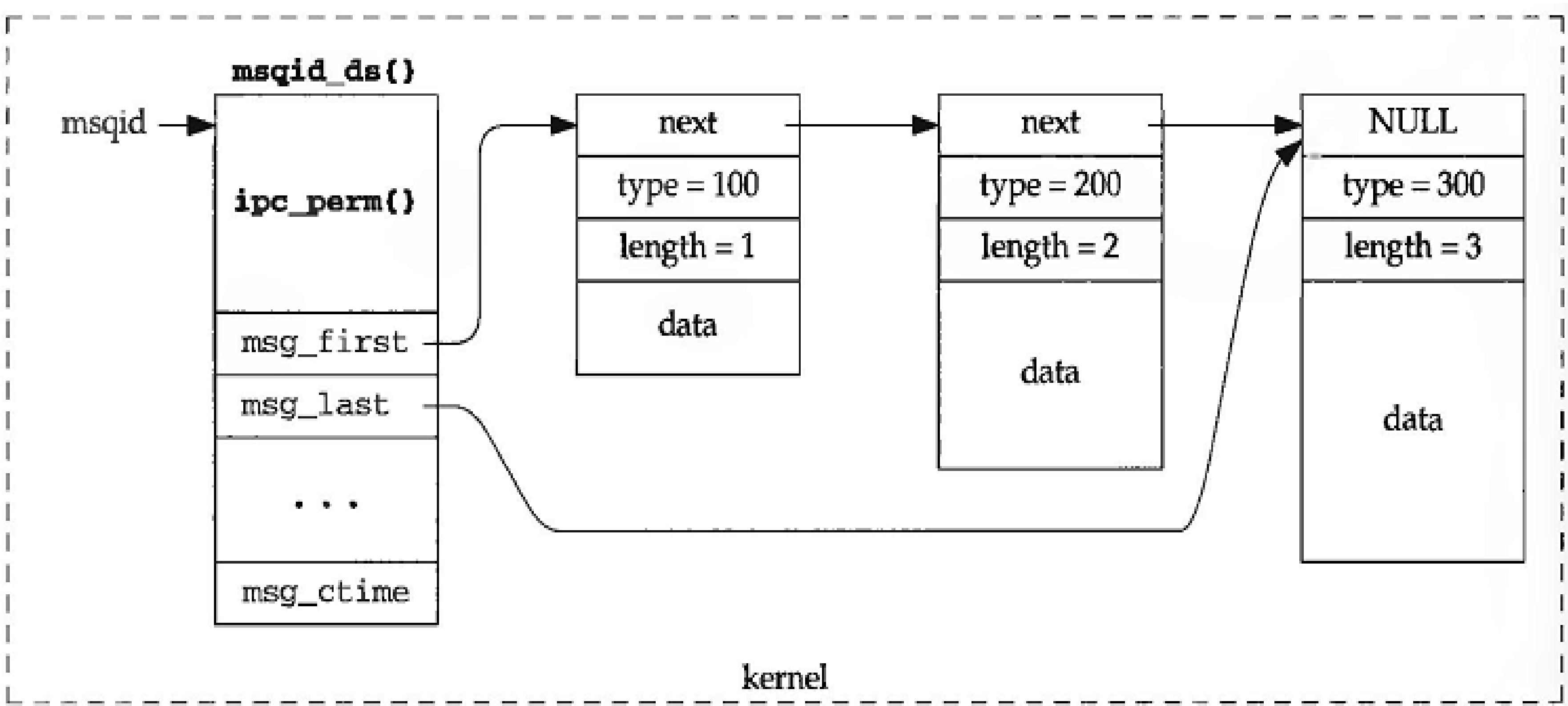
- Every message queue has following structure in kernel.

```
struct msqid_ds {
    struct ipc_perm msg_perm;          /* Ownership and permissions */
    time_t          msg_stime;         /* Time of last msgsnd() */
    time_t          msg_rtime;         /* Time of last msgrcv() */
    time_t          msg_ctime;         /* Time of last change */
    unsigned long   __msg_cbytes;      /* Number of bytes in queue */
    msgqnum_t       msg_qnum;          /* Number of messages in queue */
    msglen_t        msg_qbytes;        /* Maximum bytes in queue */
    pid_t           msg_lspid;         /* PID of last msgsnd() */
    pid_t           msg_lrpid;         /* PID of last msgrcv() */
};
```

  - msg_perm and msg_qbytes are the only two fields user can set using *msgctl()* call with IPC_SET.
  - msg_ctime refers to the last time IPC_SET operation is performed.

Message Queues

- Each message has a *type* field. This type field is useful in
  - Retrieving in other than FIFO order.
  - Multiplexing message queue.



Create Message Queue

- First *msgget()* is used to either open an existing queue or create a new queue

```c
#include <sys/msg.h>
int msgget(key_t key, int flag);
    //Returns: message queue ID if OK, 1 on error
```

- Key value can be IPC_PRIVATE, key generated by ftok() or any key (long integer)
- Flag value must be
  - IPC_CREAT if a new queue has to be created
  - IPC_CREAT and IPC_EXCL if want to create a new a queue but don’t reference existing one .

Create Message Queue

- When a new queue is created, the following members of the msqid_ds structure are initialized.
  - The ipc_perm structure is initialized
  - 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.

Messages

- Most applications define their own message structure according to the needs of the application.

```c
struct mymsg {
    long mtype;                     /* Message type */
    char mtext[];                   /* Message body */
}
```

  - First field is the message type. It is must in all messages. Remaining can be programmer defined entries.

```c
#define MY_DATA   8

typedef struct my_msgbuf {
  long      mtype;      /* message type */
  int16_t   mshort;     /* start of message data */
  char      mchar[MY_DATA];
} Message;
```

- *msgsnd() and msgrcv()* calls perform I/O on message queues.

Sending Messages

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>
int msgsnd(int  msqid , const void * msgp , size_t  msgsz , int  msgflg );
//Returns 0 on success, or -1 on error
```

- *msqid* is the id returned by msgget sys call
- The *msgp* argument is a pointer to a message structure
- *msgsz* is the length of the message without mtype field.
- A flag value of 0 or IPC_NOWAIT can be specified.

- No partial writes. Whole message is written.
- mssnd() is blocked until one of the following occurs.
  - Room exists for the message.
  - Message queue is removed (EIDRM error is returned).
  - Interrupted by a signal (EINTR is returned).
  - If IPC_NOWAIT is specified and there is no space for new message, it returns with EAGAIN.

Receiving Messages

```c
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>
ssize_t msgrcv(int  msqid , void * msgp , size_t  maxmsgsz ,
long  msgtyp , int  msgflg );
/*Returns number of bytes copied into mtext field, or -1 on error*/
```

- *msgp* points to the message structure where message will be stored.
- *maxmsgsz* points to the size available on the message structure excluding size of (long) .
- *msgtyp* indicates the message desired on the message queue.
- Flag can be 0 or IPC_NOWAIT or MSG_NOERROR
  - If MSG_ERROR is specified, even if the incoming message is bigger than the size specified, it is still copied onto the buffer. Otherwise E2BIG error is retruned.

Receiving Messages

- The type argument lets us specify which message we want.
  - 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.
    - Multiples processes can read messages by specifying different type values. Generally pid as type.
  - type < 0, treat the waiting messages as a priority queue. The first message of the lowest mtype less than or equal to the absolute value of msgtyp is removed and returned to the calling process.

```
msgrcv(id, &msg, maxmsgsz, -300, 0);

retrieves messages in the order 2 (type 100),
5 (type 100), 3 (type 200), and 1 (type 300).
A further call would block, since the type of the
remaining message (400) exceeds 300.
```

queue position	Message type (*mtype*)	Message body (*mtext*)
1	300	...
2	100	...
3	200	...
4	400	...
5	100	...

Control Operations on Message Queues

```
#include <sys/types.h>          /* For portability */
#include <sys/msg.h>
int msgctl(int  msqid , int  cmd , struct msqid_ds * buf );
//Returns 0 on success, or -1 on error
```

- IPC_STAT: Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
- IPC_SET: Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.
- IPC_RMID: Remove the message queue from the system and any data still on the queue. This 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.
- Above two commands can be executed only
  - by a process whose effective user ID equals msg_perm.cuid
  - or msg_perm.uid or by a process with superuser privileges

System V Message Q Limits

- MSGMNI
  - System-wide limit on number of message q identifiers.
- MSGMAX
  - Maximum number of bytes in a single message.
- MSGMNB
  - Maximum number of bytes that can be held in a single message q at a time.

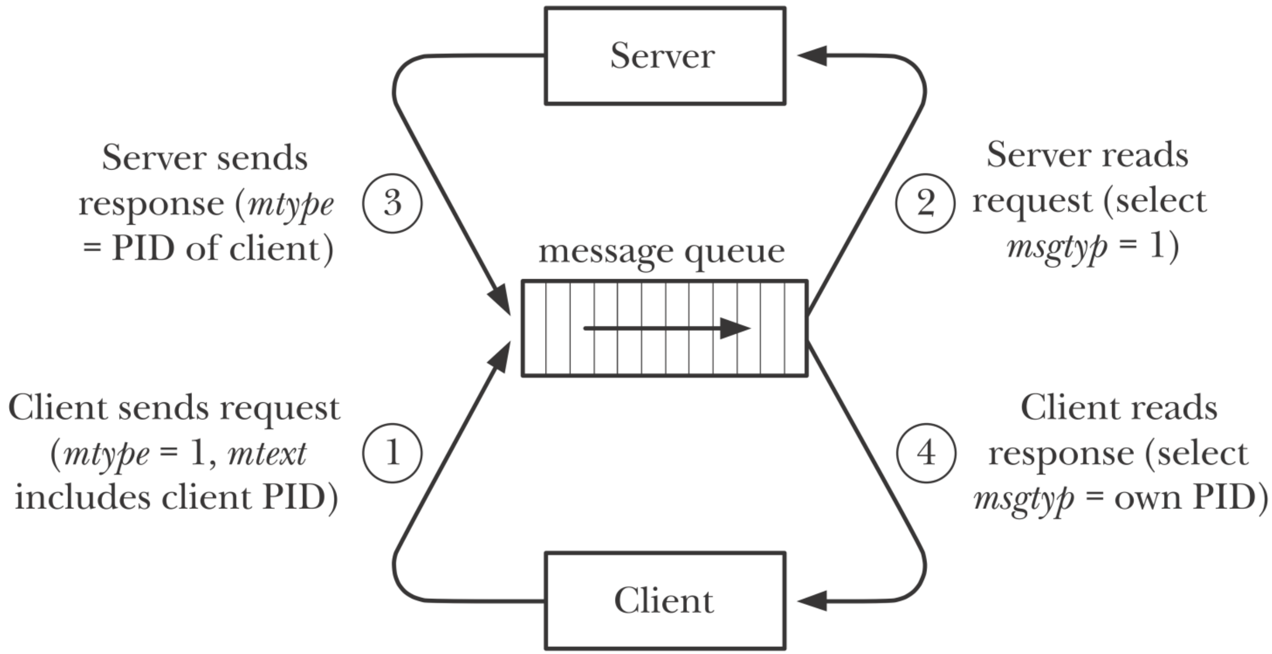
```
$ cd /proc/sys/kernel
$ cat msgmni
748
$ cat msgmax
8192
$ cat msgmnb
16384
```

Table 46-1: System V message queue limits

Limit	Ceiling value (x86-32)	Corresponding file in /proc/sys/kernel
MSGMNI	32768 (IPCMNI)	msgmni
MSGMAX	Depends on available memory	msgmax
MSGMNB	2147483647 (INT_MAX)	msgmnb

Client-Server Using MQ

- Example:
  - Client sends a string to server through a message Q
  - Server changes the case to upper and sends the string back to the client.
  - key.h includes path for ftok() fucntion.



```c
/*key.h*/
#define MSGQ_PATH "/home/user/desktop/msgq_server.c"
//put the appropriate path
```

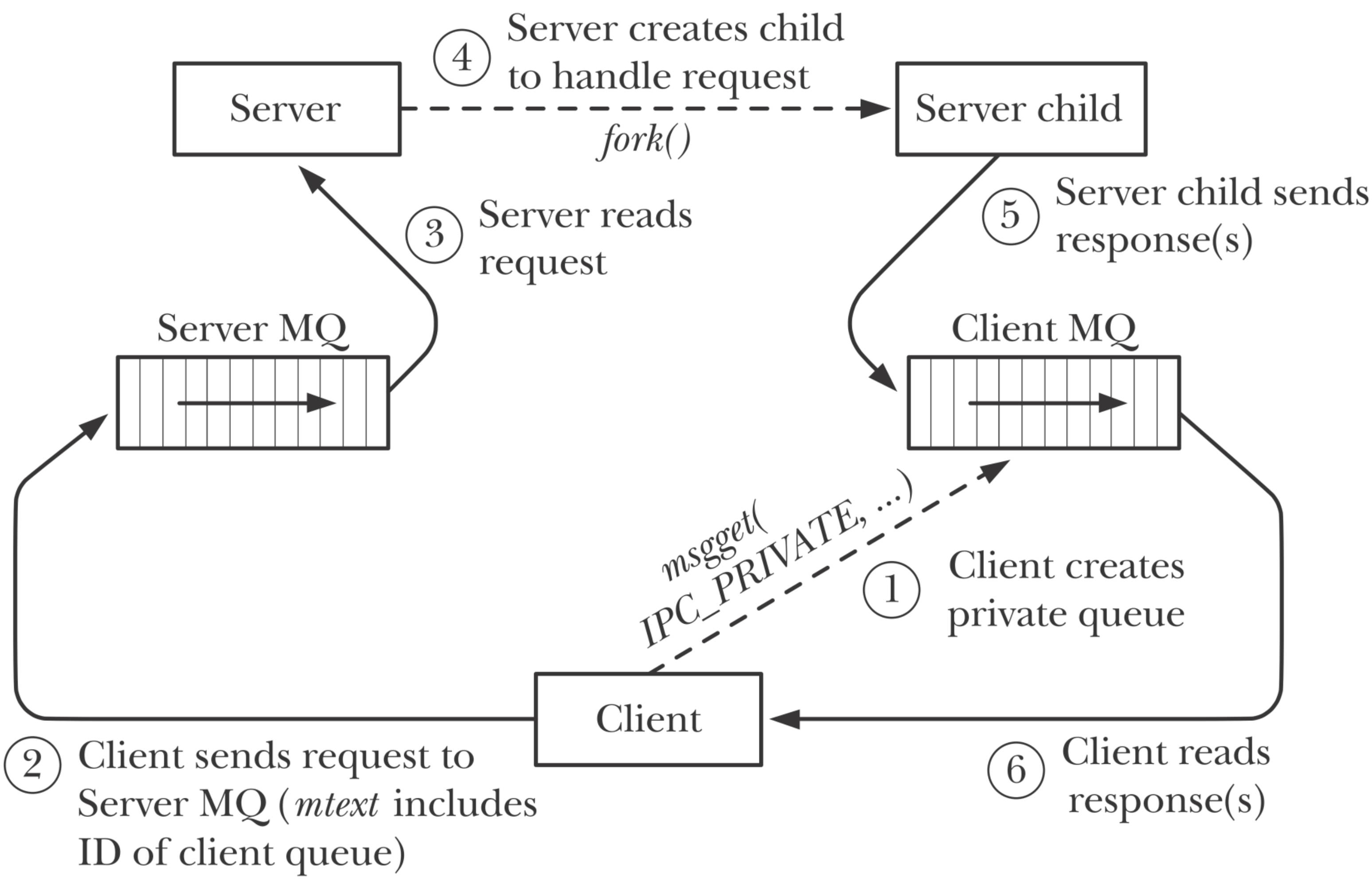
```c
/*client.c*/
#include "key.h"
struct my_msgbuf //message struct
{long mtype;
  int pid;
  char mtext[200];};
main (void)
{ struct my_msgbuf buf;
  int msqid;
  key_t key;
  if ((key = ftok (MSGQ_PATH, 'B')) == -1)//key gen
    { perror ("ftok"); exit (1);}
  if ((msqid = msgget (key, 0)) == -1)//open the q
    {perror ("msgget");exit (1);}
 printf ("Enter lines of text, ^D to quit:\n");
  buf.mtype = 1;//type for server
  buf.pid=getpid();//include own pid
  while (gets (buf.mtext), !feof (stdin))
    { if (msgsnd (msqid, &buf, sizeof (buf), 0) == -1)//send msg
        perror ("msgsnd");
      if (msgrcv (msqid, &buf, sizeof (buf),getpid(), 0) == -1)
        perror ("msgsnd");
    printf("Message received: %s\n",buf.mtext);
    }
  return 0;}
```

```c
/*server.c*/
#include "key.h"
struct my_msgbuf
{long mtype;
  int pid;
  char mtext[200];};
main (void)
{ struct my_msgbuf buf;
  int msqid;
  key_t key;int i;
  if ((key = ftok (MSGQ_PATH, 'B')) == -1)
    {perror ("ftok"); exit (1);}
  if ((msqid = msgget (key, IPC_CREAT | 0644)) == -1)
    {perror ("msgget"); exit (1);}
  printf ("server: ready to receive messages\n");
  for (;;){
      if (msgrcv (msqid, &buf, sizeof (buf), 1, 0) == -1){
      perror ("msgrcv"); exit (1);}
    for(i=0;i<strlen(buf.mtext);i++)
        buf.mtext[i]=toupper(buf.mtext[i]);
    buf.mtype=buf.pid;
      if (msgsnd (msqid, &buf, sizeof (buf), 0) == -1){
      perror ("msgsnd"); exit (1);}
    }
    return 0;}
```

Using Single Message Q for C/S

- Message Qs have limited capacity.
- Two problems:
  - Possibility of deadlock: Clients have written the messages into the queue up to the limit. Server is waiting to write a message to the Q. So both clients and server waiting for each other to remove a message.
    - Using separate Qs for client-server and server-client communication would solve this problem.
  - Poorly behaved client: A client may fail to read messages from the Q. Over a period of time this will lead to clogging of the Q.

Using Multiple Message Qs for C/S



Disadvantages of System V Msg Qs

- Advantages:
  - Ability to attach a type with a message
    - Reading processes may select messages by type
    - Priority queue strategy (type<0). Lower type means higher priority.
- Disadvantages:
  - Message q ids can’t be used in select()/poll() .
  - Use of keys rather than filenames like FIFO.
  - MQs are connection less. Kernel doesn’t maintain count of processes referring to the Q.
    - When is it safe to delete a Q?

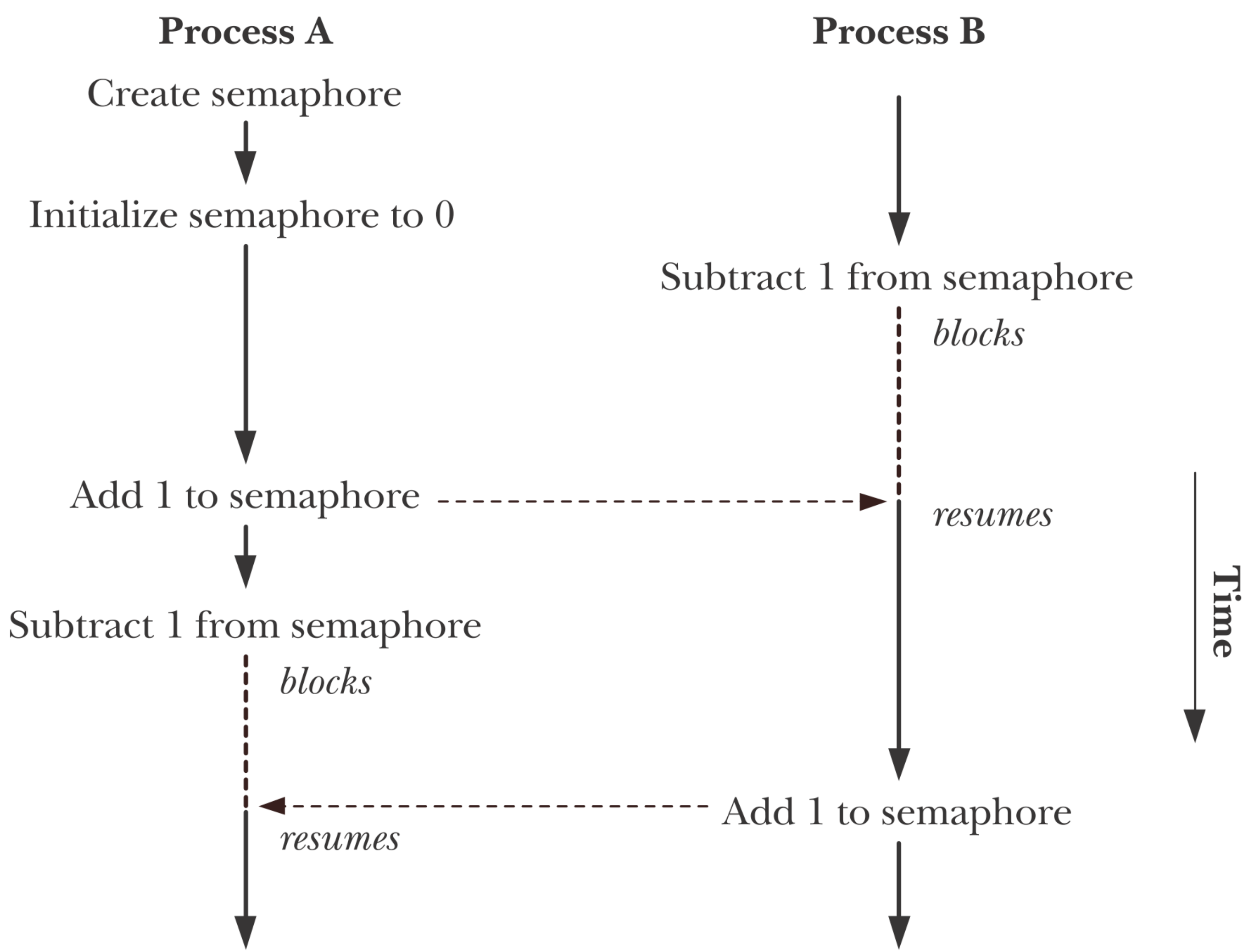
BITS Pilani
Pilani Campus

System V Semaphores

Semaphores

- A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0.
- Various operations (i.e., system calls) can be performed on a semaphore, including the following:
  - setting the semaphore to an absolute value;
  - adding a number to the current value of the semaphore;
  - subtracting a number from the current value of the semaphore (can block)
  - waiting for the semaphore value to be equal to 0 (can block).
- Semaphore has no meaning in and of itself. Meaning is associated with it by the processes using it.
  - E.g. May refer to number of buffers
- Binary Semaphores and Counting Semaphores

Semaphore: Sync between two processes



Example

- Given a shared resource, design a solution that makes the accesses in the following way.

P1↓ P2→	Read	Write
Read	√	×
Write	×	×

- How can we use semaphores to coordinate the access?

General Steps

- Create or open a semaphore set using *semget()*.
- Initialize the semaphores in the set using the *semctl()* SETVAL or SETALL operation.
  - Only one process should do this.
- Perform operations on semaphore values using *semop*().
  - The processes using the semaphore use these operations to indicate acquisition and release of a shared resource.
- When all processes have finished using the semaphore set, remove the set using the *semctl*() IPC_RMID operation.
  - Only one process should do this.

System V Semaphore

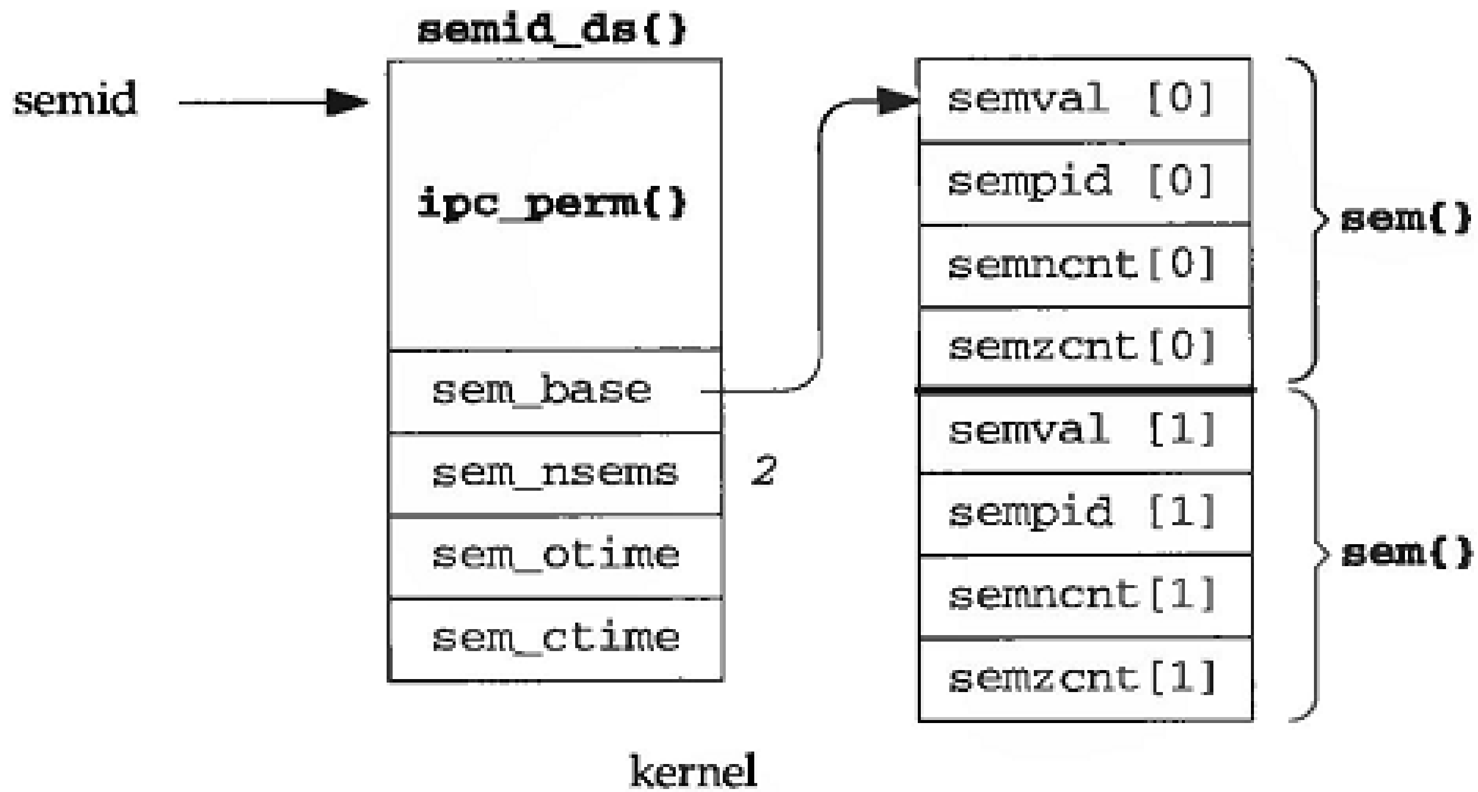
- Every semaphore set has following structure in kernel.

```
struct semid_ds {
     struct ipc_perm sem_perm;        /* permissions .. see ipc.h */
     time_t          sem_otime;       /* last semop time */
     time_t          sem_ctime;       /* last change time */
     struct sem      *sem_base;       /* ptr to first semaphore in array */
     struct wait_queue *eventn;
     struct wait_queue *eventz;
     struct sem_undo  *undo;          /* undo requests on this array */
     ushort          sem_nsems;       /* no. of semaphores in array */
};
```

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
     short   sempid;         /* pid of last operation */
     ushort  semval;         /* current value */
     ushort  semncnt;        /* num procs awaiting increase in semval */
     ushort  semzcnt;        /* num procs awaiting semval = 0 */
};
```

System V Semaphores

- Kernel structure for a semaphore set having 2 counting semaphores



Creating or Opening Semaphore Set

- Similar to creating or opening message queue.

```
#include <sys/types.h>          /* For portability */
#include <sys/sem.h>
int semget(key_t  key , int  nsems , int  semflg );
//Returns semaphore set identifier on success, or -1 on error
```

  - The number of semaphores in the set is nsems. If a new set is being created, we must specify *nsems*. If we are referencing an existing set, we can specify *nsems* as 0.
  - When a new set is created, the following members of the semid_ds structure are initialized.
    - sem_ctime is set to the current time.
    - sem_otime is set to 0.
    - The ipc_perm structure
    - sem_nsems is set to nsems.

```
struct semid_ds {
    struct ipc_perm sem_perm;        /* permissions .. see ipc.h */
    time_t          sem_otime;       /* last semop time */
    time_t          sem_ctime;       /* last change time */
    struct sem      *sem_base;       /* ptr to first semaphore in array */
    struct wait_queue *eventn;
    struct wait_queue *eventz;
    struct sem_undo  *undo;          /* undo requests on this array */
    ushort          sem_nsems;       /* no. of semaphores in array */
};
```

Semaphore Control Operations

- The semctl() system call performs a variety of control operations on a semaphore set or on an individual semaphore within a set.

```c
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>
int semctl(int  semid , int  semnum , int  cmd , ... /* union semun  arg  */);
//Returns nonnegative integer on success; returns -1 on error
```

  - Semnum specifies which semaphore (0,1,2 …)
  - Semun union is used for some commands
  - cmd: IPC_RMID, IPC_STAT etc

```c
union semun {  /* Used in calls to semctl() */
    int                   val;
    struct semid_ds *     buf;
    unsigned short *      array;
#if defined(__linux__)
    struct seminfo *      __buf;
#endif
};
```

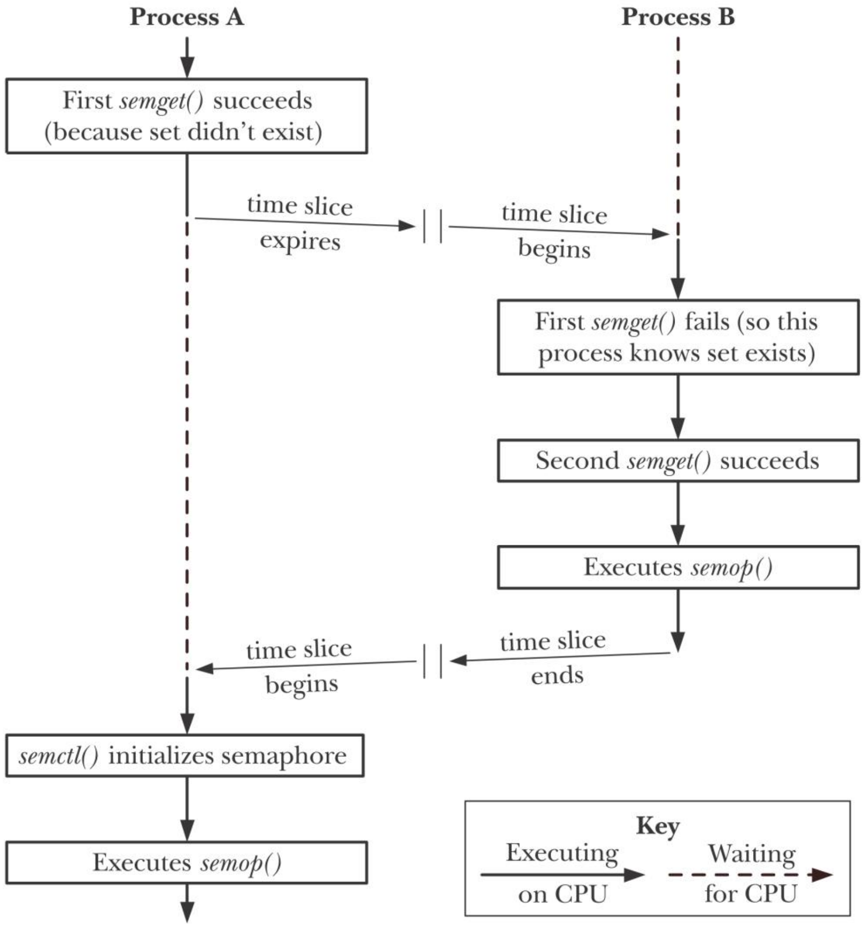
This union desn’t appear in any header file, it should be declared in your program

semctl() commands

- IPC_STAT, IPC_SET, IPC_RMID same as in message queues
- 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. These values are stored in the array pointed to by arg.array.
- SETALL: Set all the semaphore values in the set to the values pointed to by arg.array

Semaphore Initialization: race conditions

- The fact that semaphore creation and initialization are two separate steps, can lead to race conditions.



Semaphore Initialization: Solution

- Depends on the value of *sem_otime* field
- When process P1 creates semaphore *sem_otime* is set to zero.
- When P1 calls *semctl*() to initialize and then *semop()*, *sem_otime* is set to current time.
- When process P2 checks *sem_otime* is non zero it understands that semaphore has been initialized.

```
union semun arg;
struct semid_ds ds;
arg.buf = &ds;
for (j = 0; j < MAX_TRIES; j++) {
    if (semctl(semid, 0, IPC_STAT, arg) == -1)
        errExit("semctl");
    if (ds.sem_otime != 0)      /* semop() performed? */
        break;                  /* Yes, quit loop */
    sleep(1);                   /* If not, wait and retry */
}
if (ds.sem_otime == 0)          /* Loop ran to completion! */
    fatal("Existing semaphore not initialized");
```

Semaphore Operations

- The semop() system call performs one or more operations on the semaphores in the semaphore set identified by semid.

```c
#include <sys/types.h>          /* For portability */
#include <sys/sem.h>
int semop(int  semid , struct sembuf * sops , unsigned int  nsops );
//Returns 0 on success, or –1 on error
```

  - *sops* points to an array of *sembuf* structures. Each sembuf structure is one operation.
  - *nsops* is the no of operations in *sops* array.
  - Semop gurantees that either all these operations are done or none are done.

```c
struct sembuf {
    unsigned short sem_num; /* Semaphore number */
    short          sem_op; /* Operation to be performed */
    short          sem_flg; /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

Semaphore operations

- The operation on each member of the set is specified by the corresponding *sem_op* value. This value can be negative, 0, or positive.
- If sem_op > 0:
  - returning of resources by the process.
  - semval += sem_op
  - If the SEM_UNDO flag is specified, semadj -=sem_op
    - subtracted from the semaphore's adjustment value for this process.

```
struct sembuf {
    unsigned short sem_num; /* Semaphore number */
    short          sem_op;  /* Operation to be performed */
    short          sem_flg; /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

Semaphore Operations

- If sem_op <0
  - obtain resources that the semaphore controls.
  - If semval>= |sem_op|
    - the resources are available
    - semva -= |sem_op|
  - If the SEM_UNDO flag is specified,
    - semadj += sem_op
    - added to the semaphore's adjustment value for this process.

```
struct sembuf {
    unsigned short sem_num; /* Semaphore number */
    short          sem_op;  /* Operation to be performed */
    short          sem_flg; /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

Semaphore Operations

- If sem_op <0
  - If semval < |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 (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
      – Semval>=|sem_op| i.e. some other process has released some resources. Semncnt--
      – 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. and the function returns an error of EINTR. semncnt--

Semaphore Operations

- If sem_op = 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, the following conditions apply.
  - If IPC_NOWAIT is specified, return is made with an error of EAGAIN.
  - If IPC_NOWAIT is not specified, semzcnt++, and the calling process is suspended until one of the following occurs.
    - The semaphore's value becomes 0. semzcnt--
    - 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. the function returns an error of EINTR. Semzcnt--

Example

- Add a resource and wait until it becomes zero.

```
struct sembuf sops[3];
sops[0].sem_num = 0;      /* Add 1 to semaphore 0 */
sops[0].sem_op =  1;
sops[0].sem_flg = 0;
sops[1].sem_num = 0;      /* Wait till sem becomes 0 */
sops[1].sem_op = 0;
sops[1].sem_flg = IPC_NOWAIT;

if (semop(semid, sops, 2) == -1) {
   if (errno == EAGAIN)  /* Semaphore 0 would have blocked */
       printf("Operation would have blocked\n");
   else
       errExit("semop");           /* Some other error */
}
```

Semval Adjustment on Process Termination

- What if a process terminates while it has resources allocated through a semaphore.
- Kernel maintains a per-process integer: *semadj*
- SEM_UNDO flag tells kernel to do necessary changes to *semadj* value.
- If we set the value of a semaphore using *semctl()*, with either the SETVAL or SETALL commands, the *semadj* semaphore in all processes is set to 0.
- When a process terminates without releasing resources, kernel simply subtracts *semadj* value from currents sem value.

Limits on Sys V Semaphores

- SEMAEM
  - Max value in semadj
- SEMMNI
  - Limit on no of sem ids
- SEMSL
  - No of semaphores in a set
- SEMMNS
  - No of semaphores in whole system
- SEMOPM
  - Max no of ops in semop()
- SEMVMX
  - Max value for a semaphore

```
$ cd /proc/sys/kernel
$ cat sem
250      32000    32       128
//SEMMSL, SEMMNS, SEMOPM, SEMMNI
```

Table 47-1: System V semaphore limits

Limit	Ceiling value (x86-32)
SEMMNI	32768 (IPCMNI)
SEMMSL	65536
SEMMNS	2147483647 (INT_MAX)
SEMOPM	See text

Example

- Consumer consumes from the stock 1 item at a time;
- Producer produces 10 at a time when the stock reaches 0.

```
/*CONSUMER: consumes 1 item at a time*/
id = semget (KEY, 1, 0666);
operations[0].sem_num = 0;
operations[0].sem_op = -1;
operations[0].sem_flg = 0;

for (;;)
{
  retval = semop (id, operations, 1);
  if (retval == 0)
{
printf ("Consumer: Getting one object from shelf.\n");
  setval.array=val;
semctl (id, 0, GETALL, setval);
printf("Sem Value: %d\n", setval.array[0]);
}
}
```

```c
/*PRODUCER: waits until stock becomes 0 and add 10*/
unsigned short val[1];
id = semget (KEY, 1, IPC_CREAT | 0666);
setval.val = 2;
semctl (id, 0, SETVAL, setval);

operations[0].sem_num = 0;
operations[0].sem_op = 0;
operations[0].sem_flg = 0;

operations[1].sem_num = 0;
operations[1].sem_op = 10;
operations[1].sem_flg = 0;
for (;;)
{
  retval = semop (id, operations, 2);
  if (retval == 0)
{
printf ("Producer: Adding 10  objects\n");
getval.array = val;
  semctl (id, 0, GETALL, getval);
printf ("Sem Val: %d\n", getval.array[0]);
}
}
```

Example

- Given a shared resource, design a solution that makes the accesses in the following way.

P1↓ P2→	Read	Write
Read	√	×
Write	×	×

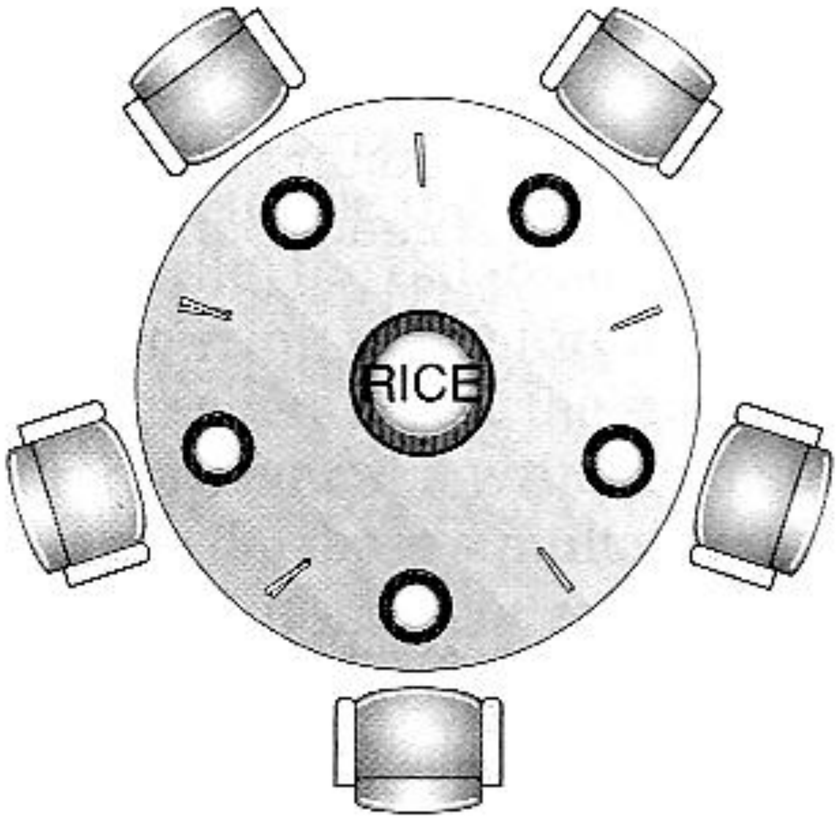
Example[1]

- There will be two semaphores.
  - Sem 0, For read access
  - Sem 1, For write access
- For reading do the following array of operations
  - {1,0,0} //wait until writers become zero
  - {0,1,0} //increase readers by 1
  - //read
  - {0,-1,0} //decrease readers by 1

```
struct sembuf {
    unsigned short sem_num;
    short          sem_op;
    short          sem_flg;
};
```

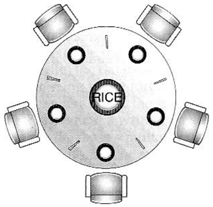
- For writing do the following array of operations
  - {1,0,0} //wait until writers become zero
  - {0,0,0} // wait until readers become zero
  - {1,1,0} //increase no of writers by 1
  - //write
  - {1,1,0} //decrease no of writers by 1

Dining Philosophers Problem



Dining Philosophers Problem

- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.



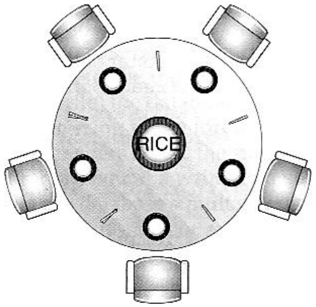
Semaphore Solution

- Chopsticks are the shared objects.
- For each chopstick, let there be a semaphore.
- N=5 semaphores. Each init to 1.
- **For each philosopher i:**
- do{

```
    semop((i+1)%N, -1, 0);
    semop((i+N-1)%N, -1, 0);
    //eat
    semop((i+1)%N, 1, 0);
    semop((i+N-1)%N, 1, 0);
    //think
}
```

```
struct sembuf {
    unsigned short sem_num;
    short          sem_op;
    short          sem_flg;
};
```

- This can create deadlock.

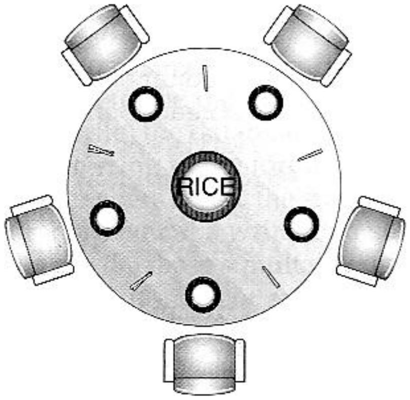


Semaphore Solution Improvement

- Improvement: A philosopher may move into eating state only if no other neighbor is eating.
- **`For each philosopher i:`**
- **`do{`**

```
Semop(binsem, -1,0)
semop((i+1)%N, -1, 0);
semop((i+N-1)%N, -1, 0);
//eat
semop((i+1)%N, 1, 0);
semop((i+N-1)%N, 1, 0);
semop(binsem, 1,0);
//think
```

- **`}`**



- But only one philosopher can eat at a time. There can be two eating at the same time.

Semaphore Solution Improvement

- A philosopher may move only into eating state if none of the neighbor s (LEFT and RIGHT) is eating .
- Requires that state of philosopher to be maintained.
- **`For each philosopher i:`**
- **`do{`**

```
 semop(binsem, -1,0)
 If(state[i]==THINKING && state[i+N-1%N]!=EATING &&
    state[i+1%N]!=EATING)
      state[i]=EATING;
 semop(binsem, 1,0);

 semop((i+1)%N, -1, 0);
 semop((i+N-1)%N, -1, 0);
 //eat
 semop((i+1)%N, 1, 0);
 semop((i+N-1)%N, 1, 0);
 //think
```

- **`}`**

innovate achieve lead

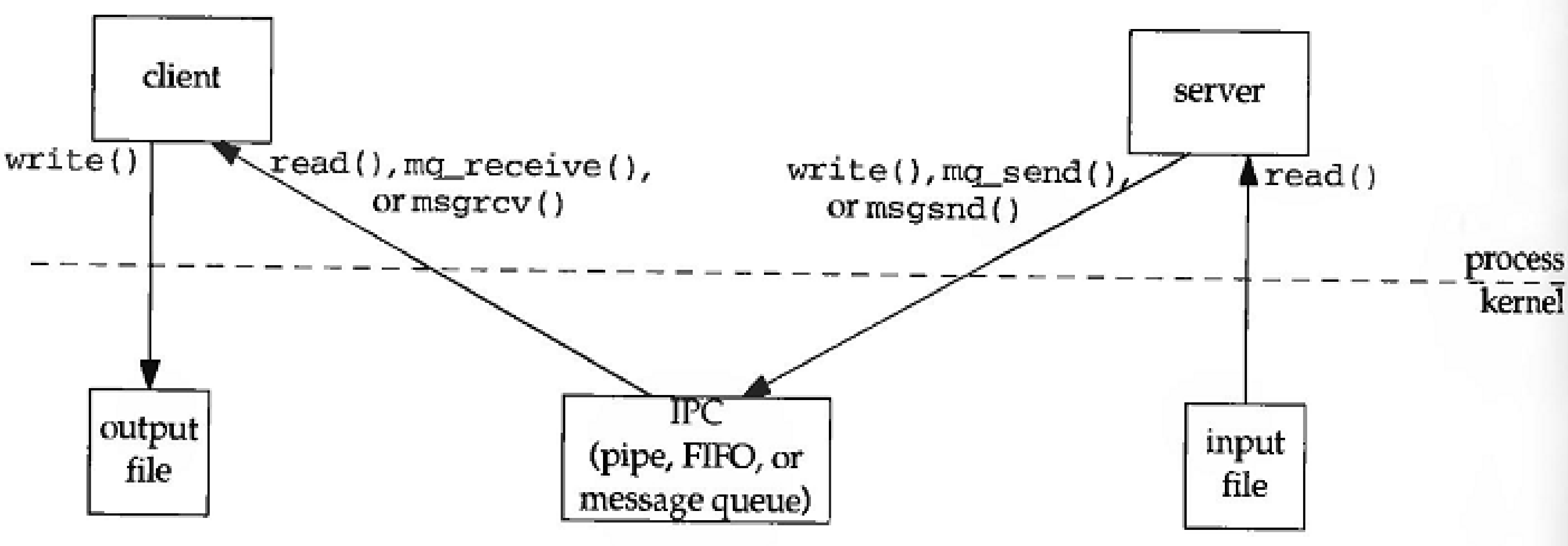
BITS Pilani
Pilani Campus

System V Shared Memory

Shared Memory

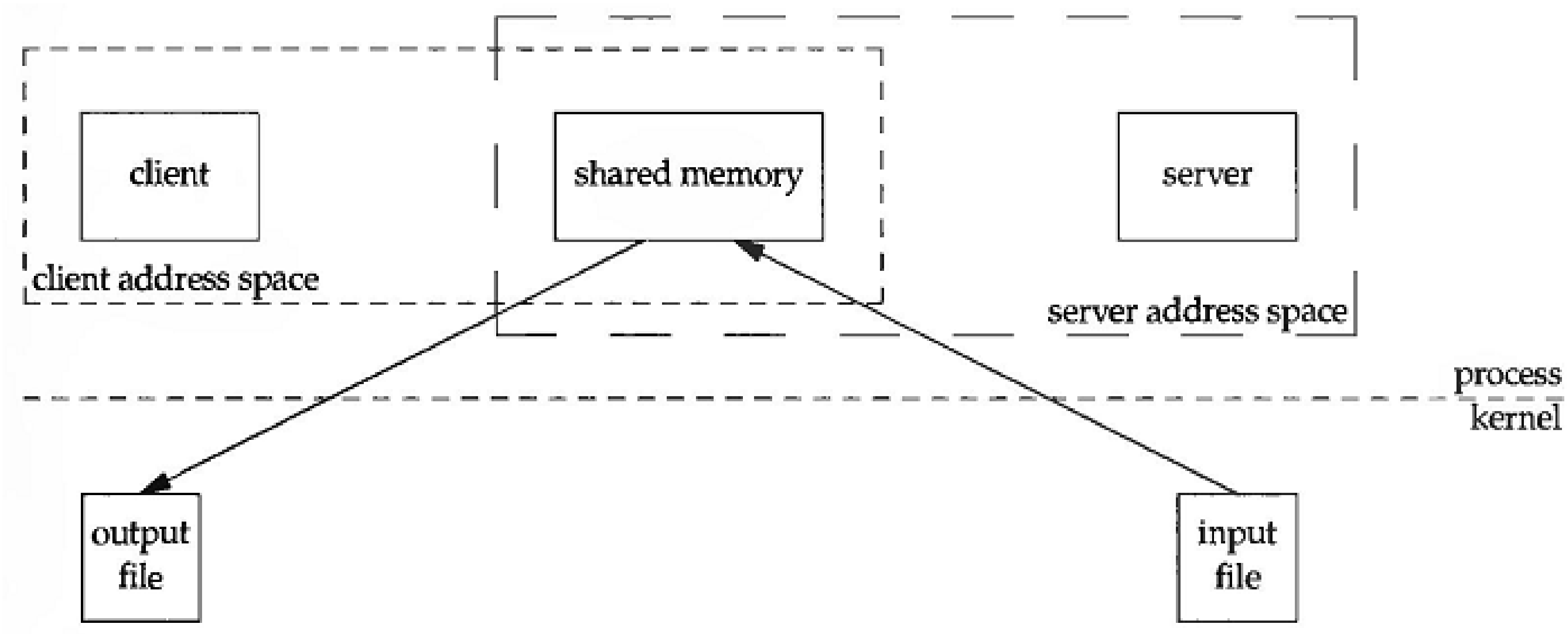
- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server

Message Passing



- Takes 4 copies to transfer data between two processes

Shared Memory



- Takes only two steps
- Kernel is not involved in transferring data but it is involved in creating shared memory

System V Shared Memory

- For every shared memory segment kernel maintains the following structure.

```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* Ownership and permissions */
    size_t   shm_segsz;         /* Size of segment in bytes */
    time_t   shm_atime;         /* Time of last shmat() */
    time_t   shm_dtime;         /* Time of last shmdt() */
    time_t   shm_ctime;         /* Time of last change */
    pid_t    shm_cpid;          /* PID of creator */
    pid_t    shm_lpid;          /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch;        /* Number of currently attached processes */
};
```

System V Shared Memory

- Creating or opening shared memory

```c
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>
int shmget(key_t  key , size_t  size , int  shmflg );
//Returns shared memory segment identifier on success, or -1 on error
```

  - Size is given in bytes.
  - Size is given as zero if we are referencing existing shared memory segment.
  - When a new segment is created, the contents of the segment are initialized with zeros.
  - Flags: IPC_CREAT, IPC_EXCL

Attaching Shared Memory to a Process

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```c
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>
void *shmat(int  shmid , const void * shmaddr , int  shmflg );
//Returns address at which shared memory is attached on success,
//or (void *) -1 on error
```

  - The address in the calling process at which the segment is attached depends on the addr argument.
  - If addr is 0, the segment is attached at the first available address selected by the kernel.
    - This is the recommended technique.

Table 48-1: *shmflg* bit-mask values for *shmat()*

Value	Description
SHM_RDONLY	Attach segment read-only
SHM_REMAP	Replace any existing mapping at *shmaddr*
SHM_RND	Round *shmaddr* down to multiple of SHMLBA bytes

Detaching Shared Memory from a Process

```c
#include <sys/types.h>  /* For portability */
#include <sys/shm.h>
int shmdt(const void * shmaddr );
//Returns 0 on success, or -1 on error
```

- this does not remove the identifier and its associated data structure from the system.
- the identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

shmctl

```c
#include <sys/types.h>          /* For portability */
#include <sys/shm.h>
int shmctl(int  shmid , int  cmd , struct shmid_ds * buf );
//Returns 0 on success, or -1 on error
```

- IPC_STAT, IPC_SET same as other System V IPC or XSI IPC.
- IPC_RMID:
  - Remove the shared memory segment set from the system. The segment is not removed until the last process using the segment terminates or detaches it.

Q&A

Next Time

- Please read through R1: chapters 47-48

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You