Inter-Process Communications (IPCs):
Message Queues
Shared Memory
Semaphores

May 2011

IPCs (System V)

- Three types of IPCs:
  - Message Queues
  - Shared Memory
  - Semaphores
- Each IPC structure is referred to by a non-negative integer identifier.
  - When an IPC is created, the program responsible for this creation provides a key of type *key_t*.
  - The Operating System converts this key into an IPC identifier.

Keys in the IPC Client-Server Paradigm

$\Rightarrow$ Keys can be created in three ways:

1. The "server" program creates a new structure by specifying a private key that is *IPC_PRIVATE*.
   - Client has to become explicitly aware of this private key.
   - This is often accomplished with the help of a file generated by the server and then looked-up by the client.
2. Server and client do agree on a key value (often defined and hard-coded in the header).
3. Server and client can agree on a pathname to an existing file in the file system *AND* a project-ID (0..255) and then call *ftok()* to convert these two values into a unique key!

Keys

- Keys help identify resources and offer access to the internal structures of the 3 IPC mechanisms (through systems calls):
  - *struct msqid_ds* // for message queues
  - *struct shmid_ds* // for shared segments
  - *struct semid_ds* // for semaphores
- Wrongly accessing resources returns -1
- Access rights for IPC mechanisms: read/write stored in *struct ipc_perm*
- Included header files:
  #include <sys/ipc.h>
  #include <sys/types.h>

The *ftok()* system call

- converts a pathname and a project identifier to a (System V) IPC-key
- *key_t ftok(const char *pathname, int proj_id)*
- ```
  if ( (thekey=ftok("/tmp/ad.tempfile", 23)) == -1)
      perror("Cannot create key from /tmp/ad.
          tempfile");
  ```
- The file */tmp/ad.tempfile* must be accessible by the invoking process.
  ```

Message Queues

- Message queues allow for the exchange of messages between processes.
- The dispatching process sends a specific type of message and the receiving process may request the specific type of message.
- Each message consists of its "type" and the "payload".
- Messages are pointers to stuctures:

```
struct message{
        long type;
        char messagetext[MESSAGESIZE];
};
```

- Header needed: *#include <sys/msg.h>*

The system call *msgget()* - creating/using a queue

int msgget(key_t key, int msgflg)

- returns (creates) a message queue identifier associated with the value of the *key* argument.
- A new message queue is created if *key* has the value *IPC_PRIVATE*.
- If key isn't *IPC_PRIVATE* and no message queue with the given *key* exists, the *msgflg* must be specified to *IPC_CREAT* (to create the queue).
- If a queue with key *key* exists and both *IPC_CREAT* and *IPC_EXCL* are specified in *msgflg*, then *msgget* fails with *errno* set to *EEXIST*.

Use-cases of *msgflg*

- Upon creation, the least significant bits of *msgflg* define the permissions of the message queue.
- These permission bits have the same format and semantics as the permissions specified for the mode argument of *open()*.
- The various use-cases of *msgflg* are:

	PERMS	*PERMS* | *IPC_CREAT*	*PERMS* | *IPC_CREAT* | *IPC_EXCL*
resource exists	use resource	use resource	error
resource does not exist	error	create and use new resource	create and use new resource

System call *msgsnd() - sending a message to a queue*

*int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)*

- send *msgp* (pointer to a record – see below) to message queue with id *msqid*.
- ```
  struct msgbuf {
        long mtype; // message type-must be > 0
        char mtext[MSGSZ]; // message data
        };
  ```
- sender must have write-access permission on the message queue to send a message.
  ```

System call *msgrcv()* – fetching a message from a queue

*ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);*

- receive a message *msgp* from a message queue with id *msqid*
- *msgtyp* is a strictly positive integer value.
- if *msgtyp* is zero, the first message is retrieved regardless its type.
- This value can be used by the receiving process for message selection.
- *mesgsz* specifies the size of the field *mtext*.
- By and large, *msgflg* is set to 0.

The role of *msgtyp* in *msgrcv()*

msgtyp specifies the type of message requested as follows:

- if *msgtyp=0* then the first message in the queue is read.
- if *msgtyp* $>$ *0* then the first message in the queue of type *msgtyp* is read.
- if *msgtyp* $<$ *0* then the first message in the queue with the lowest type value is read.
  - Assume a queue has 3 messages with *mtype* 1, 40, 554 and and *msgtyp* is set to -554; If *msgrcv* is called three times, the messages will be received in the following order: 1, 40, 554.

The *msgctl()* call - controling a queue

*int msgctl(int msqid, int cmd, struct msqid_ds *buf)*

- performs the control operation specified by *cmd* on the message queue with identifier *msqid*
- The *msqid_ds* structure is defined in <sys/msg.h> as:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t  msg_stime;     /* Time of last msgsnd(2) */
    time_t  msg_rtime;     /* Time of last msgrcv(2) */
    time_t  msg_ctime;     /* Time of last change */
    unsigned long  __msg_cbytes; /* Current number of bytes in
                                    queue (non-standard) */
    msgqnum_t msg_qnum;    /* Current number of messages
                                 in queue */
    msglen_t msg_qbytes;   /* Maximum number of bytes
                                allowed in queue */
    pid_t  msg_lspid;     /* PID of last msgsnd(2) */
    pid_t  msg_lrpid;     /* PID of last msgrcv(2) */
};
```

Operating with *msgctl()* on message queues

- *IPC_STAT*: Copy information from the kernel data structure associated with *msqid* into the *msqid_ds* structure pointed to by *buf*.
- *IPC_SET*: Write the values of some members of the *msqid_ds* structure pointed to by *buf* to the kernel data structure associated with this message queue, updating also its *msg_ctime* element.
- *IPC_RMID*: Immediately remove the message queue, awakening all waiting reader and writer processes (with an *error* return and *errno* set to *EIDRM*).

The server in a message-queue communication

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MSGSIZE 128
#define PERMS 0666
#define SERVER_MTYPE 27L
#define CLIENT_MTYPE 42L

struct message{
    long mtype;
    char mtext[MSGSIZE];
    };

main(){
    int qid;
    struct message sbuf, rbuf;
    key_t the_key;

    the_key = ftok("/home/ad/SysProMaterial/Set008/src/fileA", 226);

    if ( (qid = msgget(the_key, PERMS | IPC_CREAT)) < 0 ){
        perror("megget"); exit(1);
        }
    printf("Creating message queue with identifier %d \n",qid);
```

The server in a message-queue communication

```
    sbuf.mtype = SERVER_MTYPE;
    strcpy(sbuf.mtext,"A message from server");
    if (msgsnd(qid, &sbuf, strlen(sbuf.mtext)+1, 0) < 0){
        perror("msgsnd"); exit(1);
        }
    printf("Sent message: %s\n",sbuf.mtext);

    if ( msgrcv(qid, &rbuf, MSGSIZE, CLIENT_MTYPE, 0) < 0){
        perror("msgrcv"); exit(1);}
    printf("Received message: %s\n",rbuf.mtext);

    if ( msgrcv(qid, &rbuf, MSGSIZE, CLIENT_MTYPE, 0) < 0){
        perror("msgrcv"); exit(1);}
    printf("Received message: %s\n",rbuf.mtext);

    if (msgctl(qid, IPC_RMID, (struct msqid_ds *)0) < 0){
        perror("msgctl"); exit(1);}
    printf("Removed message queue with identifier %d\n",qid);

}
```

Client (1) in the message-queue communication

```
....
#define MSGSIZE 128
#define PERMS 0666
#define SERVER_MTYPE 27L
#define CLIENT_MTYPE 42L

struct message{
    long mtype;
    char mtext[MSGSIZE]; };

main(){
    int qid; struct message sbuf, rbuf; key_t the_key;

        the_key = ftok("/home/ad/SysProMaterial/Set008/src/fileA", 226);
    if ( (qid = msgget(the_key, PERMS)) < 0 ){
        perror("megget"); exit(1); }
    printf("Accessing message queue with identifier %d \n",qid);
    if ( msgrcv(qid, &rbuf, MSGSIZE, SERVER_MTYPE, 0) < 0){
        perror("msgrcv"); exit(1);}
    printf("Received message: %s\n",rbuf.mtext);
    sbuf.mtype = CLIENT_MTYPE;
    strcpy(sbuf.mtext,"A message from client 1");
    if (msgsnd(qid, &sbuf, strlen(sbuf.mtext)+1, 0) < 0){
        perror("msgsnd"); exit(1);
        }
    printf("Sent message: %s\n",sbuf.mtext);
}
```

Client (2) in the message-queue communication

```
......
#define MSGSIZE 128
#define PERMS 0666
#define SERVER_MTYPE 27L
#define CLIENT_MTYPE 42L

struct message{
    long mtype;
    char mtext[MSGSIZE]; };

main(){
    int qid; struct message sbuf, rbuf; key_t the_key;

        the_key = ftok("/home/ad/SysProMaterial/Set008/src/fileA", 226);
    if ( (qid = msgget(the_key, PERMS)) < 0 ){
        perror("megget"); exit(1); }
    printf("Accessing message queue with identifier %d \n",qid);
    sbuf.mtype = CLIENT_MTYPE;
    strcpy(sbuf.mtext,"A message from client 2");
    if (msgsnd(qid, &sbuf, strlen(sbuf.mtext)+1, 0) < 0){
        perror("msgsnd"); exit(1);
        }
    printf("Sent message: %s\n",sbuf.mtext);
}
```

Running the application

The server:

```
ad@ad-desktop:~/SysProMaterial/Set008/src$ ./msg-server
Creating message queue with identifier 65536
Sent message: A message from server
```

Client 1:

```
ad@ad-desktop:~/SysProMaterial/Set008/src$ ./msg-client1
Accessing message queue with identifier 65536
Received message: A message from server
Sent message: A message from client 1
ad@ad-desktop:~/SysProMaterial/Set008/src$
```

Server status:

```
ad@ad-desktop:~/SysProMaterial/Set008/src$ ./msg-server
Creating message queue with identifier 65536
Sent message: A message from server
Received message: A message from client 1
```

Running the application

Client 2:

```
ad@ad-desktop:~/SysProMaterial/Set008/src$ ./msg-client2
Accessing message queue with identifier 65536
Sent message: A message from client 2
ad@ad-desktop:~/SysProMaterial/Set008/src$
```

Server:

```
ad@ad-desktop:~/SysProMaterial/Set008/src$ ./msg-server
Creating message queue with identifier 65536
Sent message: A message from server
Received message: A message from client 1
Received message: A message from client 2
Removed message queue with identifier 65536
ad@ad-desktop:~/SysProMaterial/Set008/src$
```

Developing a Priority Queue

- Implement a Queue in which Jobs have Priorities
- A server gets the items from the queue and and in some way (pick one) “processes” these items.

"q.h"

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <errno.h>

#define QKEY          (key_t) 108
#define QPERM         0660
#define MAXOBN        50
#define MAXPRIOR      10

struct q_entry{
    long mtype;
    char mtext[MAXOBN+1];
    };
```

"init_queue.c"

```c
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

int warn(char *s){
    fprintf(stderr,"Warning: %s\n",s);
}

int init_queue(void){
    int queue_id;

    if ( (queue_id = msgget(QKEY, IPC_CREAT | QPERM)) == -1 )
        perror("msgget failed");
    return(queue_id);
}
```

"enter.c"

```c
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

int enter(char *objname, int priority){
    int len, s_qid;
    struct q_entry s_entry;

    if ( (len=strlen(objname)) > MAXOBN){
        warn("name too long\n"); exit(1);
        }
    if ( priority > MAXPRIOR || priority < 0 ){
        warn("invalid priority level"); return(-1);
        }
    if ( (s_qid = init_queue()) == -1 ) return(-1);

    s_entry.mtype= (long)priority;
    strncpy(s_entry.mtext, objname, MAXOBN);

    if (msgsnd(s_qid, &s_entry, len, 0) == -1 ){
        perror("msgsnd failed"); return(-1);}
    else    return(0);
}
```

"etest.c"

```
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

main(int argc, char *argv[]){
    int priority;

    if ( argc!= 3){
        fprintf(stderr,"usage: %s objname priority\n",argv[0]);
        }
    if ((priority = atoi(argv[2])) <=0 || priority > MAXPRIOR){
        warn("invalid priority"); exit(2);
        }

    if ( enter(argv[1], priority) < 0 ){
        warn("enter failure"); exit(3);
        }
    exit(0);
}
```

→ *gcc enter.c init_queue.c etest.c -o etest*

"serve.c"

```
#include "q.h"

int serve(void){
    int mlen, r_qid;
    struct q_entry r_entry;

    if ( (r_qid=init_queue()) == -1) return(-1);

    for(;;){
        if ( (mlen=msgrcv(r_qid, &r_entry, MAXOBN,
            (-1 * MAXPRIOR), MSG_NOERROR) ) == -1 ){
            perror("mesgrcv failed"); return(-1);
            }
        else {
            r_entry.mtext[mlen]='\0';
            proc_obj(&r_entry);
            }
        }
}
```

"stest.c"

```c
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

extern void server();

main(){
    pid_t pid;

    switch (pid=fork()){
    case 0: // child
        serve();
        break;
    case -1:
        warn("fork to start the server failed");
        break;
    default:
        printf("server process pid is %d \n", pid);
    }
exit(pid != 1 ? 0 : 1);
}

int proc_obj(struct q_entry *msg){
    printf("\npriority: %ld name: %s\n", msg->mtype, msg->mtext);
    }
```

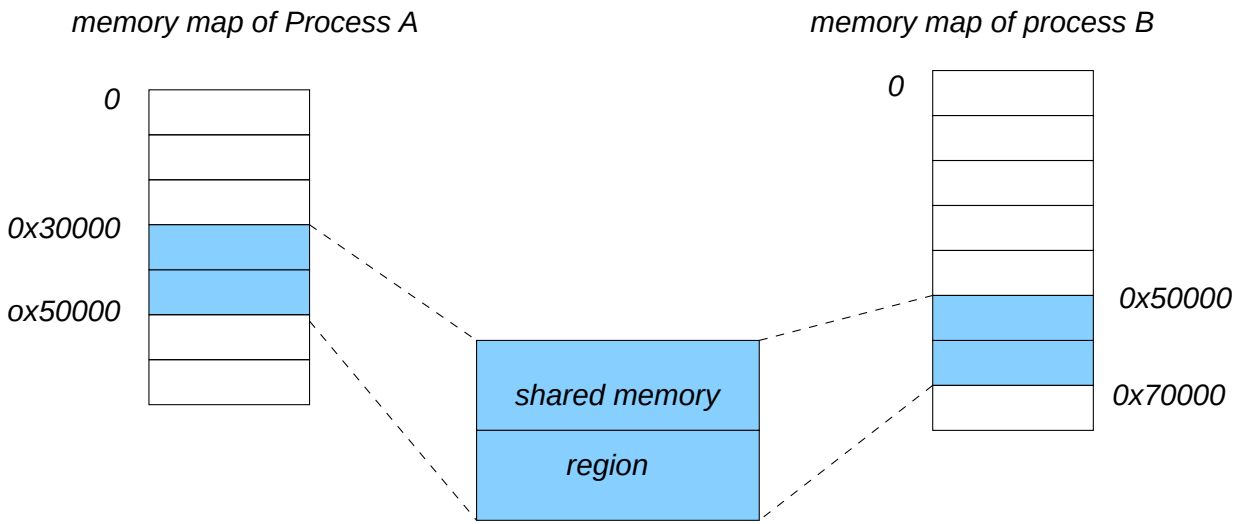
→ *gcc stest.c serve.c init_queue.c -o stest*

Running the "priority queue" program(s)

```
ad@ad-desktop:~/src/PriorityQueue$ ./etest object1 4
ad@ad-desktop:~/src/PriorityQueue$ ./etest object2 1
ad@ad-desktop:~/src/PriorityQueue$ ./etest object3 7
ad@ad-desktop:~/src/PriorityQueue$ ./etest object4 9
ad@ad-desktop:~/src/PriorityQueue$ ./stest
server process pid is 2213

priority: 1 name: object2

priority: 4 name: object1

priority: 7 name: object3

priority: 9 name: object4
ad@ad-desktop:~/src/PriorityQueue$
```

Shared Memory

- A shared memory region is a portion of physical memory that is shared by multiple processes.



- In this region, structures can be set up by processes and others may read/write on them.
- Synchronization (when it is required) is achieved with the help of semaphores.

Creating a shared segment with *shmget()*

- *#include <sys/ipc.h>*
  #include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg)

- returns the identifier of the shared memory segment associated with the value of the argument *key*.
- the returned size of the segment is equal to *size* rounded up to a multiple of *PAGE_SIZE*.
- *shmflg* helps designate the access rights for the segment (*IPC_CREAT* and *IPC_EXCL* are used in a way similar to that of message queues).
- If *shmflg* specifies *both IPC_CREAT* and *IPC_EXCL* and a shared memory segment already exists for *key*, then *shmget()* fails with *errno* set to *EEXIST*.

Attach- and Detach-ing a segment:*shmat()/shmdt()*

*void *shmat(int shmid, const void *shmaddr, int shmflg)*

- attaches the shared memory segment identified by *shmid* to the address space of the calling process.
- If *shmaddr* is NULL, the OS chooses a suitable (unused) address at which to attach the segment (frequent choice).
- Otherwise, *shmaddr* must be a page-aligned address at which the attach occurs.

*int shmdt(const void *shmaddr)*

- detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process.

The system call *shmctl()*

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

- performs the control operation specified by *cmd* on the shared memory segment whose identifier is given in *shmid*.
- The *buf* argument is a pointer to a *shmid_ds* structure:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

The system call *shmctl()*

Usual values for *cmd* are:

- *IPC_STAT*: copy information from the kernel data structure associated with *shmid* into the *shmid_ds* structure pointed to by *buf*.
- *IPC_SET*: write the values of some members of the *shmid_ds* structure pointed to by *buf* to the kernel data structure associated with this shared memory segment, updating also its *shm_ctime* member.
- *IPC_RMID*: mark the segment to be destroyed. The segment will be destroyed after the last process detaches it (i.e., *shm_nattch* is zero).

Use Cases of Calls

- Only one process creates the segment:

```
int id;
id = shmget(IPC_PRIVATE, 10, 0666);
if ( id == -1 ) perror("Creating");
```

- Every (interested) process attaches the segment:

```
int *mem;
mem = (int *) shmat (id, (void *)0, 0);
if ( (int)mem == -1 ) perror("Attachment");
```

- Every process detaches the segment:

```
int err;
err = shmdt((void *)mem);
if ( err == -1 ) perror("Detachment");
```

- Only one process has to remove the segment:

```
int err;
err = shmctl(id, IPC_RMID, 0);
if ( err == -1 ) perror("Removal");
```

Creating and accessing shared memory ("shareMem1.c")

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char **argv){
    int id=0, err=0;
    int *mem;

    id = shmget(IPC_PRIVATE,10,0666);  /* Make shared memory segment */
    if (id == -1) perror ("Creation");
    else printf("Allocated. %d\n",(int)id);

    mem = (int *) shmat(id, (void*)0, 0); /* Attach the segment */
    if ((int) mem == -1) perror("Attachment.");
    else printf("Attached. Mem contents %d\n",*mem);

    *mem=1; /* Give it initial value */
    printf("Start other process. >"); getchar();

    printf("mem is now %d\n", *mem); /* Print out new value */

    err = shmctl(id, IPC_RMID, 0); /* Remove segment */
    if (err == -1) perror ("Removal.");
    else printf("Removed. %d\n", (int)(err));
    return 0;
}
```

Creating and accessing shared memory ("shareMem2.c")

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char **argv) {
    int id, err;
    int *mem;

    if (argc <= 1) { printf("Need shared memory id. \n"); exit(1); }

    sscanf(argv[1], "%d", &id); /* Get id from command line. */
    printf("Id is %d\n", id);

    mem = (int *) shmat(id, (void*) 0,0); /* Attach the segment */
    if ((int) mem == -1) perror("Attachment.");
    else printf("Attached. Mem contents %d\n",*mem);

    *mem=2; /* Give it a different value */
    printf("Changed mem is now %d\n", *mem);

    err = shmdt((void *) mem); /* Detach segment */
    if (err == -1) perror ("Detachment.");
    else printf("Detachment %d\n", err);
    return 0;
}
```

Running the two programs:

- Starting off with executing "shareMem1":

```
ad@ad-desktop:~/Set008/src/SharedSegments$ ./shareMem1
Allocated. 1769489
Attached. Mem contents 0
Start other process. >
```

- Executing "shareMem2":

```
ad@ad-desktop:~/Set008/src/SharedSegments$ ./shareMem2 1769489
Id is 1769489
Attached. Mem contents 1
Changed mem is now 2
Detachment 0
ad@ad-desktop:~/Set008/src/SharedSegments$
```

- Providing the final input to "shareMem1":

```
Start other process. >s
mem is now 2
Removed. 0
ad@ad-desktop:~/Set008/src/SharedSegments$
```

Semaphores

- Fundamental mechanism that facilitates the synchronization of accessing resources placed in shared memory.
- A semaphore is an integer whose value is never allowed to fall below zero.
- *Two operations* can be performed on a semaphore:
  - increment the semaphore value by one (*UP* or *V()* ala Dijkstra).
  - decrement a semaphore value by one (*DOWN* or *P()* ala Dijkstra). If the value of semaphore is currently zero, then the invoking process will block until the value becomes greater than zero.

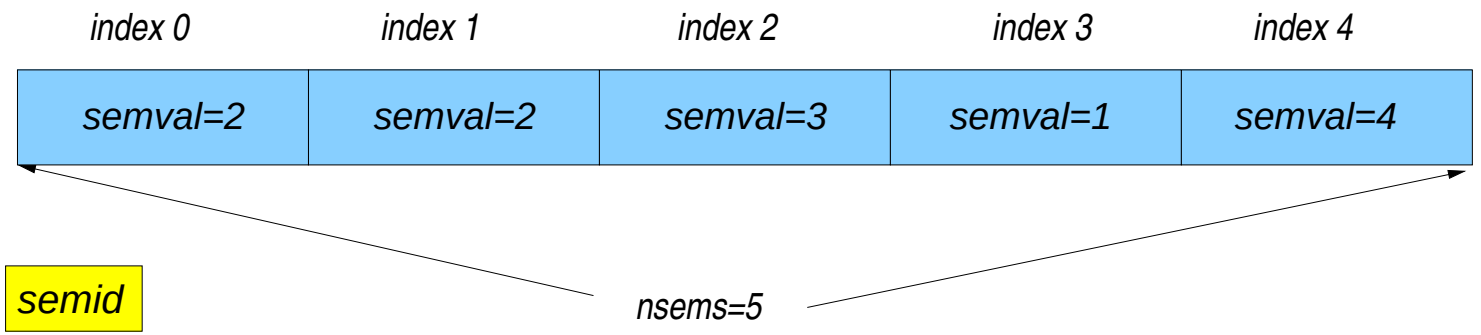
System-V Semaphores

- In general, (System-V) system calls create sets of semaphores:
  - The kernel warrants atomic operations on these sets.
  - Should we have more than one resources to protect, we can "lock" all of them simultaneously.

Creating a set of Semaphores

- included files: *<sys/types.h>* *<sys/ipc.h>* *<sys/sem.h>*
- *int semget(key_t key, int nsems, int semflg)*
- returns the semaphore set identifier associated with the argument *key*.
- A new set of *nsems* semaphores is created if *key* has the value *IPC_PRIVATE* **OR** if no existing semaphore set is associated with *key* and *IPC_CREAT* is specified in *semflg*.
- *semflg* helps set the access right for the semaphore set.
- If *semflg* specifies both *IPC_CREAT* and *IPC_EXCL* and a semaphore set already exists for *key*, then *semget()* fails with *errno* set to *EEXIST*.

Structure of a Semaphore Set



Associated with each (single) semaphore in the set are the following values:

- *semval*: the semaphore value, always a positive number.
- *sempid*: the *pid* of the process that last "acted" on the set (of semaphores).

Operating on a Set of Semaphores

- *int semop(int semid, struct sembuf *sops, unsigned nsops)*
- performs operations on *selected* semaphores in the set indicated by *semid*.
- each of the *nsops* elements in the array pointed to by *sops* specifies an operation to be performed on a single semaphore on the set.

Operating on a Set of Semaphores

- The elements of the *struct sembuf* have as follows:

```
struct sembuf{
    unsigned short  sem_num;   /* semaphore number */
    short           sem_op;    /* semaphore operation */
    short           sem_flg;   /* operation flags */
    };
```

- In the above:
  - *sem_num* identifies the ID of the specific semaphore on the set (0..*nsemd*-1) on which *sem_op* operates.
  - The value of *sem_op* is set to:
    - < 0 for locking
    - > 0 for unlocking
  - *sem_flg* often set to 0.

The *semctl()* system call

- *int semctl(int semid, int semnum, int cmd, [union semun arg])*
- performs the control operation specified by *cmd* on the *semnum*-th semaphore of the set identified by *semid*.
- The forth parameter (if it exists) has the following layout:

```
union semun {
    int               val;    /* Value for SETVAL */
    struct semid_ds  *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short   *array;  /* Array for GETALL, SETALL */
    struct seminfo   *__buf;  /* Buffer for IPC_INFO (Linux-specific) */
};
```

The *semid_ds* structure

- The semaphore data structure *semid_ds*, is as follows:

```
struct semid_ds {
     struct ipc_perm sem_perm;  /* Ownership and permissions */
     time_t          sem_otime; /* Last semop time */
     time_t          sem_ctime; /* Last change time */
     unsigned short  sem_nsems; /* No. of semaphores in set */
     };
```

semctl()

Values for the *cmd* parameter:

- *IPC_STAT*: copy information from the kernel data structure associated with *semid* into the *semid_ds* structure pointed to by *arg.buf*.
- *IPC_SET*: write the values of some members of the *semid_ds* structure pointed to by *arg.buf* to the kernel data structure associated with this semaphore set updating also its *sem_ctime* member.
- *IPC_SETALL*: Set *semval* for all semaphores of the set using *arg.array*, updating also the *sem_ctime* member of the *semid_ds* structure associated with the set.
- *IPC_GETALL*: Return to *semval* the current values of all semaphores of the set *arg.array*.
- *IPC_RMID*: remove the semaphore set while awakening all processes blocked by the respective *semop()*.

A server program using Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SHMKEY (key_t)4321
#define SEMKEY (key_t)9876
#define SHMSIZE 256
#define PERMS 0600

union semnum{
    int val;
    struct semid_ds *buff;
    unsigned short *array; };

main(){
    int shmid, semid; char line[128], *shmem;
    struct sembuf oper[1]={0,1,0};
    union semnum arg;

    if ((shmid = shmget (SHMKEY, SHMSIZE, PERMS | IPC_CREAT)) < 0) {
        perror("shmget"); exit(1); }
    printf("Creating shared memory with ID: %d\n",shmid);
    /* create a semaphore */
    if ((semid = semget(SEMKEY, 1, PERMS| IPC_CREAT)) <0) {
        perror("semget"); exit(1); }
    printf("Creating a semaphore with ID: %d \n",semid);
    arg.val=0;
```

A server program using Semaphores (continued)

```
/* initialize semaphore for locking */
if (semctl(semid, 0, SETVAL, arg) <0) {
    perror("semctl");
    exit(1);
    }
printf("Initializing semaphore to lock\n");

if ( (shmem = shmat(shmid, (char *)0, 0)) == (char *) -1) {
    perror("shmem");
    exit(1);
    }
printf("Attaching shared memory segment \nEnter a string: ");
fgets(line, sizeof(line), stdin);
line[strlen(line)-1]='\0';

/* Write message in shared memory */
strcpy(shmem, line);

printf("Writing to shared memory region: %s\n", line);

/* Make shared memory available for reading */
if ( semop(semid, &oper[0], 1) < 0 ) {
    perror("semop");
    exit(1);
    }
shmdt(shmem);
printf("Releasing shared memory region\n");
}
```

A client program using semaphore

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SHMKEY (key_t)4321
#define SEMKEY (key_t)9876
#define SHMSIZE 256
#define PERMS 0600

main(){
    int shmid, semid;
    char *shmem;
    struct sembuf oper[1]={0,-1,0};

    if ((shmid = shmget (SHMKEY, SHMSIZE, PERMS )) < 0) {
        perror("shmget"); exit(1); }
    printf("Accessing shared memory with ID: %d\n",shmid);

    /* accessing a semaphore */
    if ((semid = semget(SEMKEY, 1, PERMS )) <0) {
        perror("semget"); exit(1); }
    printf("Accessing semaphore with ID: %d \n",semid);
```

A client program using semaphore (continued)

```
if ( (shmem = shmat(shmid, (char *) 0, 0)) == (char *) -1 ) {
    perror("shmat"); exit(1); }
printf("Attaching shared memory segment\n");

printf("Asking for access to shared memory region \n");
if (semop(semid, &oper[0], 1) <0)  {
    perror("semop"); exit(1); }
printf("Reading from shared memory region: %s\n", shmem);

/* detach shared memeory */
shmdt(shmem);

/* destroy shared memory */
if (shmctl(shmid, IPC_RMID, (struct shmid_ds *)0 ) <0) {
    perror("semctl"); exit(1); }
printf("Releasing shared segment with identifier %d\n", shmid);

/* destroy semaphore set */
if (semctl(semid, 0, IPC_RMID, 0) <0 ) {
    perror("semctl"); exit(1); }
printf("Releasing semaphore with identifier %d\n", semid);
}
```

Running the server and the client

The server:

```
ad@ad-desktop:~/SysProMaterial/Set008/src/V-Sems$ ./sem-server
Creating shared memory with ID: 22511641
Creating a semaphore with ID: 327688
Initializing semaphore to lock
Attaching shared memory segment
Enter a string:
```

The client:

```
ad@ad-desktop:~/SysProMaterial/Set008/src/V-Sems$ ./sem-client
Accessing shared memory with ID: 22511641
Accessing semaphore with ID: 327688
Attaching shared memory segment
Asking for access to shared memory region
```

Running the programs

- Server:

```
ad@ad-desktop:~/src/V-Sems$ ./sem-server
Creating shared memory with ID: 22511641
Creating a semaphore with ID: 327688
Initializing semaphore to lock
Attaching shared memory segment
Enter a string: THIS IS A TEST ONLY A TEST
Writing to shared memory region: THIS IS A TEST ONLY A TEST
Releasing shared memory region
ad@ad-desktop:~/src/V-Sems$
```

- Client:

```
ad@ad-desktop:~/src/V-Sems$ ./sem-client
Accessing shared memory with ID: 22511641
Accessing semaphore with ID: 327688
Attaching shared memory segment
Asking for access to shared memory region
Reading from shared memory region: THIS IS A TEST ONLY A TEST
Releasing shared segment with identifier 22511641
Releasing semaphore with identifier 327688
ad@ad-desktop:~/src/V-Sems$
```

Access to Critical Section

```c
#include <stdio.h> /* Example code using semaphores and shared memory */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/ipc.h>

/* Union semun */
union semun {
   int val;                    /* value for SETVAL */
   struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
   unsigned short *array;      /* array for GETALL, SETALL */
};

void free_resources(int shm_id, int sem_id) {
   /* Delete the shared memory segment */
   shmctl(shm_id,IPC_RMID,NULL);
   /* Delete the semaphore */
   semctl(sem_id,0,IPC_RMID,0);
}

int sem_P(int sem_id) {      /* Semaphore P - down operation, using semop */
   struct sembuf sem_d;

   sem_d.sem_num = 0;
   sem_d.sem_op = -1;
   sem_d.sem_flg = 0;
   if (semop(sem_id,&sem_d,1) == -1) {
       perror("# Semaphore down (P) operation "); return -1; }
   return 0;
}
```

Access to Critical Section

```c
/* Semaphore V - up operation, using semop */
int sem_V(int sem_id) {
   struct sembuf sem_d;

   sem_d.sem_num = 0;
   sem_d.sem_op = 1;
   sem_d.sem_flg = 0;
   if (semop(sem_id,&sem_d,1) == -1) {
       perror("# Semaphore up (V) operation "); return -1; }
   return 0;
}

/* Semaphore Init - set a semaphore's value to val */
int sem_Init(int sem_id, int val) {
   union semun arg;

   arg.val = val;
   if (semctl(sem_id,0,SETVAL,arg) == -1) {
       perror("# Semaphore setting value "); return -1; }
   return 0;
}
```

Access to Critical Section

```
int main () {
   int shm_id; int sem_id; int t = 0; int *sh; int pid;

   /* Create a new shared memory segment */
   shm_id = shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT | 0660);
   if (shm_id == -1) {
       perror("Shared memory creation"); exit(EXIT_FAILURE); }

   /* Create a new semaphore id */
   sem_id = semget(IPC_PRIVATE,1,IPC_CREAT | 0660);
   if (sem_id == -1) {
       perror("Semaphore creation ");
       shmctl(shm_id,IPC_RMID,(struct shmid_ds *)NULL);
       exit(EXIT_FAILURE);
   }

   /* Set the value of the semaphore to 1 */
   if (sem_Init(sem_id, 1) == -1) {
       free_resources(shm_id,sem_id);
       exit(EXIT_FAILURE);
   }

   sh = (int *)shmat(shm_id,NULL,0);  /* Attach the shared memory segment */
   if (sh == NULL) {
       perror("Shared memory attach ");
       free_resources(shm_id,sem_id);
       exit(EXIT_FAILURE);
   }
   /* Setting shared memory to 0 */
   *sh = 0;
```

Access to Critical Section

```
/* New process */
if ((pid = fork()) == -1) {
    perror("fork");
    free_resources(shm_id,sem_id);
    exit(EXIT_FAILURE);
}

if (pid == 0) {
   /* Child process */
   printf("# I am the child process with process id: %d\n", getpid());
} else {
   /* Parent process */
   printf("# I am the parent process with process id: %d\n", getpid());
   sleep(2);
}

printf("(%d): trying to access the critical section\n", getpid());
sem_P(sem_id);
printf("(%d): accessed the critical section\n", getpid());

(*sh)++;
printf("(%d): value of shared memory is now: %d\n", getpid(), *sh);

printf("(%d): getting out of the critical section\n", getpid());
sem_V(sem_id);

printf("(%d): got out of the critical section\n", getpid());
```

Access to Critical Section

```
    /* Child process */
    if (!pid)
      exit(EXIT_SUCCESS);

    /* Wait for child process */
    wait(NULL);

    /* Clear recourses */
    free_resources(shm_id,sem_id);
    return 0;
}
```

→ outcome of execution:

```
ad@ad-desktop:~/src/V-Sems$ ./access-criticalsection
# I am the parent process with process id: 9256
# I am the child process with process id: 9257
(9257): trying to access the critical section
(9257): accessed the critical section
(9257): value of shared memory is now: 1
(9257): getting out of the critical section
(9257): got out of the critical section
(9256): trying to access the critical section
(9256): accessed the critical section
(9256): value of shared memory is now: 2
(9256): getting out of the critical section
(9256): got out of the critical section
ad@ad-desktop:~/src/V-Sems$
```

POSIX Semaphores

- *#include <semaphore.h>*
- *sem_init, sem_destroy, sem_post, sem_wait, sem_trywait*
- *int sem_init(sem_t *sem, int pshared, unsigned int value);*
  - The above initializes a semaphore.
  - Compile either with -lrt or -lpthread
  - *pshared* indicates whether this semaphore is to be shared between the threads of a process, or between processes:
    - zero: semaphore is shared between the **threads of a process**; should be located at an address visible to **all threads**.
    - non-zero: semaphore is shared **among processes** and should be located in a region of shared memory.

POSIX Semaphore Operations

- *sem_wait()*, *sem_trywait()*
  - *int sem_wait(sem_t *sem);*
  - *int sem_trywait(sem_t *sem);*
  - Perform $\mathrm{P}(\mathrm{s})$ operation.
  - *sem_wait* blocks; *sem_trywait* will fail rather than block.
- *sem_post()*
  - *int sem_post(sem_t *sem);*
  - Perform $\mathrm{V}(\mathrm{s})$ operation.
- *sem_destroy()*
  - *int sem_destroy(sem_t *sem);*
  - Destroys a semaphore.

Creating and using a POSIX Semaphore

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>

extern int errno;

int main(int argc, char **argv)
{
    sem_t sp; int retval;

    /* Initialize the semaphore. */
    retval = sem_init(&sp,1,2);
    if (retval != 0) {
        perror("Couldn't initialize."); exit(3); }

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    sem_destroy(&sp);
    return 0;
}
```

Executing the Program

```
ad@ad-desktop:~/src/PosixSems$ ./semtest
Did trywait. Returned 0 >

Did trywait. Returned 0 >

Did trywait. Returned -1 >

ad@ad-desktop:~/src/PosixSems$
```

Initialize and Open a **named** *Semaphore*

- *sem_t *sem_open(const char *name, int oflag);*
  *sem_t *sem_open(const char *name, int oflag,*
  mode_t mode, unsigned int value);
- creates a new POSIX semaphore OR opens an existing semaphore whose name is *name*.
- *oflag* specifies flags that control the operation of the call
  - *O_CREAT* creates the semaphore;
  - provided that both *O_CREAT* and *O_EXCL* are specified, an *error* is returned if a semaphore with *name* already exists.
- if *oflag* is *O_CREAT* then **two more arguments** have to be used:
  - *mode* specifies the permissions to be placed on the new semaphore.
  - *value* specifies the initial value for the new semaphore.

Named POSIX Semaphore

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <assert.h>
#include    <errno.h>
#include    <unistd.h>
#include    <sys/file.h>
#include    <sys/stat.h>
#include    <semaphore.h>

int main(int argc, char *argv[]){

const  char *semname = "mysemafile";
int op = 0; int val=0;

sem_t *sem=sem_open(semname, O_CREAT|O_EXCL, S_IRUSR|S_IWUSR, 0);

if (sem!= SEM_FAILED)
    printf("created new semaphore!\n");
else if (errno== EEXIST ) {
    printf("semaphore appears to exist already!\n");
    sem = sem_open(semname, 0);
    }
else ;

assert(sem != SEM_FAILED);
```

Named Posix Semaphore

```
if (argc>1)
    op=atoi(argv[1]);

if ( op == 1 ) {
    printf("incrementing semaphore\n");
    sem_post(sem);
    }
else if ( op == -1 ) {
    printf("decrementing semaphore\n");
    sem_wait(sem);
    }
else    printf("not defined operation! \n");

sem_getvalue(sem, &val);
printf("semaphore's current value is %d\n",val);
return(0);
}
```

Execution Outcome

```
ad@ad-desktop:~/src/PosixSems$ ./namedSem 1
created new semaphore!
incrementing semaphore
semaphore's current value is 1
ad@ad-desktop:~/src/PosixSems$ ./namedSem 1
semaphore appears to exist already!
incrementing semaphore
semaphore's current value is 2
ad@ad-desktop:~/src/PosixSems$ ./namedSem -1
semaphore appears to exist already!
decrementing semaphore
semaphore's current value is 1
ad@ad-desktop:~/src/PosixSems$ ./namedSem -1
semaphore appears to exist already!
decrementing semaphore
semaphore's current value is 0
ad@ad-desktop:~/src/PosixSems$ ./namedSem -1
semaphore appears to exist already!
decrementing semaphore
^C
ad@ad-desktop:~/SysProMaterial/Set008/src/PosixSems$
```

Locking a file

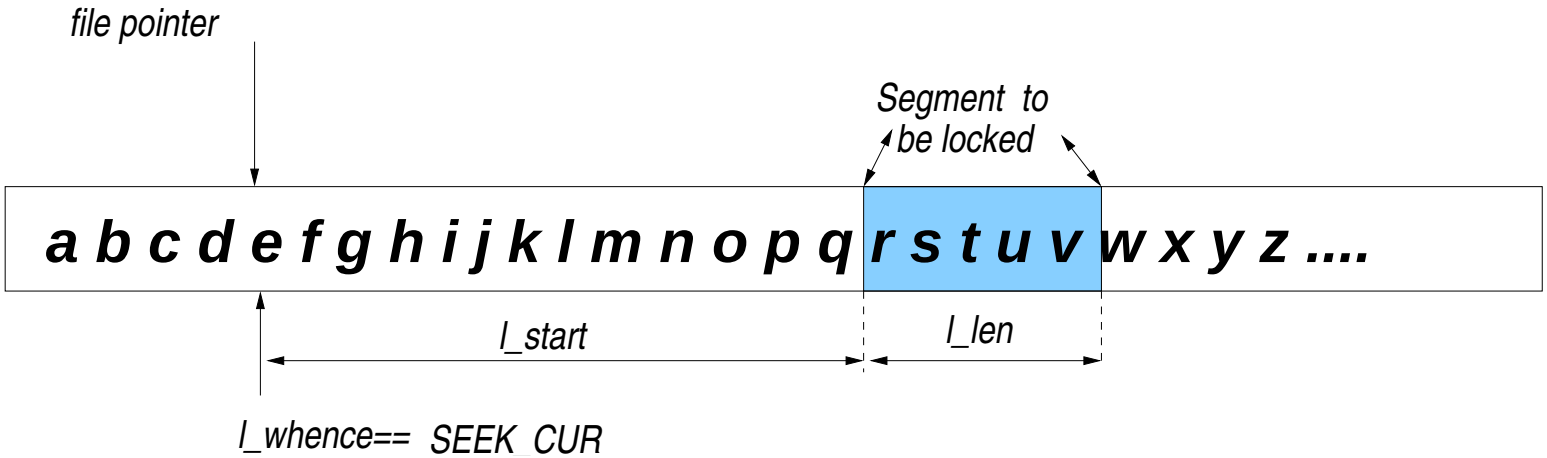
- Imposing read/write locks on files (or sections of files) is essential at times.
- *#include <fnctl.h>*
  *int fnctl(int filedes, int cmd, struct flock *ldata)*
- File *filedes* must be opened with *O_RDONLY* or *O_WRONLY*.
- The *cmd* can be one of the three:
  - *F_GETLK*: get lock from data returned from *ldata*
  - *F_SETLK*: apply lock to a file; return immediately if this is not feasible.
  - *F_SETLKW*: apply lock to a file. Sleep if lock blocked by a previous lock owned by another process.

The *flock* structure

- The *flock* structure is defined in <*fnctl.h*> and includes:

```
struct flock {
    ...
    short  l_type;    /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
    short  l_whence;  /* How to interpret l_start:
                         SEEK_SET, SEEK_CUR, SEEK_END */
    off_t  l_start;   /* Starting offset for lock */
    off_t  l_len;     /* Number of bytes to lock */
    pid_t  l_pid;     /* PID of process blocking our lock (F_GETLK only) */
    ...
};
```

Locking a file



- *l_whence*: can be *SEEK_SET*, *SEEK_CUR* or *SEEK_END*.
  l_start: start position of the segment.
  l_len: segment in bytes.
- The *l_type* (lock type) can be:
  - *F_RDLCL*: lock to be applied is *read*
  - *F_WRLCL*: lock to be applied is *write*
  - *F_UNLCL*: lock on specified segment to be removed.

Locking a file

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main( ){
    int fd;
    struct flock my_lock;

    my_lock.l_type = F_WRLCK;
    my_lock.l_whence =  SEEK_SET;
    my_lock.l_start = 0 ;
    my_lock.l_len= 10;

    fd=open("locktest", O_RDWR);

    // lock first 10 bytes
    if ( fcntl(fd, F_SETLKW , &my_lock) == -1 ){
        perror("parent: locking");
            exit(1);
            }

    printf("parent: locked record \n");
```

Locking a file

```
    switch(fork()){
        case -1:
            perror("fork"); exit(1);
        case 0:
            printf("child: trying to lock file \n");
            my_lock.l_len = 5 ;
            if ( (fcntl(fd, F_SETLKW, &my_lock)) == -1 ){
                perror("child: problem in locking");
                exit(1);
                }
            printf("child: locked \n"); sleep(1);
            printf("child: exiting \n");
            fflush(stdout); fflush(stderr); exit(1);
        default:
            printf("parent: just about unlocking now \n");
            sleep(5);
            my_lock.l_type = F_UNLCK;
            printf("parent: unlocking -now- \n");
            if ( fcntl(fd, F_SETLK, &my_lock) == -1 ){
                perror("parent: problem in unlocking! \n");
                exit(1); }
            printf("parent: has unlocked and is now exiting \n");
            fflush(stdout); fflush(stderr); wait(NULL);
    }
    sleep(2);
}
```

Execution Outcome

```
ad@ad-desktop:~/Filelocking$ ./lockit
parent: locked record
child: trying to lock file
parent: just about unlocking now
parent: unlocking -now-
parent: has unlocked and is now exiting
child: locked
child: exiting
ad@ad-desktop:~/Filelocking$
```

Possible Deadlock

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main( ){
    int fd;
    struct flock first_lock;
    struct flock second_lock;

    first_lock.l_type = F_WRLCK;
    first_lock.l_whence =  SEEK_SET;
    first_lock.l_start = 0 ;
    first_lock.l_len= 10;

    second_lock.l_type = F_WRLCK;
    second_lock.l_whence =  SEEK_SET;
    second_lock.l_start = 10;
    second_lock.l_len= 5;


    fd=open("locktest", O_RDWR);

    if ( fcntl(fd, F_SETLKW , &first_lock) == -1 )
        perror("-A:");
    printf("A: lock obtained by processs %d \n",getpid());

    switch(fork()) {
        case -1:
            perror("error on fork");
            exit(1);
```

Possible Deadlock

```
        case 0: /* child */
            if ( fcntl(fd, F_SETLKW, &second_lock) == -1 )
                perror("-B:");
            printf("B: lock obtained by process %d\n",getpid());

            if ( fcntl(fd, F_SETLKW, &first_lock) == -1 ){
                perror("-C:");
                printf("Process %d terminating\n",getpid());
                     exit(1);
                }
            else printf("C: lock obtained by process %d\n",getpid());
            printf("Process %d successfully acquired BOTH locks \n",getpid());
            exit(0);
        default: /* parent */
            printf("Parent process %d sleeping \n",getpid());
            sleep(10);
            if ( fcntl(fd, F_SETLK, &second_lock) == -1 ){
                perror("--D:");
                printf("Process %d about to terminate\n",getpid());
                }
            else printf("D: lock obtained by process %d\n",getpid());
            sleep(1);
            printf("Process %d on its way out of here \n",getpid());
    }
}
```

Execution Outcome

```
ad@ad-desktop:~/K24/MolC/Filelocking$ ./deadlock
A: lock obtained by processs 10822
Parent process 10822 sleeping
B: lock obtained by process 10823
--D:: Resource temporarily unavailable
Process 10822 about to terminate
Process 10822 on its way out of here
C: lock obtained by process 10823
Process 10823 successfully acquired BOTH locks
ad@ad-desktop:~/K24/MolC/Filelocking$
```