POSIX Inter-Process Communication (IPC)

- Message Queues
- Shared Memory
- Semaphores

- Reading: R&R, Ch 15

POSIX IPC: Overview

primitive	POSIX function	description
message queues	**msgget** **msgctl** **msgsnd/msgrcv**	create or access control send/receive message
semaphores	**semget** **semctl** **semop**	create or access control wait or post operation
shared memory	**shmget** **shmctl** **shmat/shmdt**	create and init or access control attach to / detach from process

xxGET: It's all about Naming!

- Condition variables, mutex locks:
  - Based on a **memory variable** concept.
  - Does not work across memory spaces!!
- Pipes
  - Uses **file descriptors**
  - Works across memory spaces.
  - Relies on inheritance of file descriptors -> does not work for unrelated processes.
- Named Pipes
  - Uses **file system as name space** for pipe.
  - Works for unrelated processes.
  - Carry the overhead of the file system.
- IPC Objects
  - Use system-global integer **keys** to refer to objects.

IPC Object Creation: Message Queues

Object key identifies object across processes. Can be assigned as follows:
-- Create some unknown key (`IPC_PRIVATE`)
-- Pass explicit key (beware of collisions!)
-- Use file system to consistently hash key (using `ftok`)

```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
/* create a message queue with given key and flags. */
```

Object id is similar to file descriptor.
-- It can be inherited across `fork()` calls.
-- Only useful in `fork()/exec()` settings.

Operations on Message Queues

```
#define PERMS (S_IRUSR | S_IWUSR)

int msqid;
if ((msqid = msgget(IPC_PRIVATE, PERMS)) == -1)
    perror(“msgget failed);
```

```
struct mymsg { /* user defined! */
    long msgtype;  /* first field must be a long identifier */
    char mtext[1];  /* placeholder for message content */
}
```

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

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

msgtyp	action
0	remove first message from queue
> 0	remove first message of type `msgtyp` from the queue
< 0	remove first message of lowest type that is less than or equal to absolute value of `msgtyp`

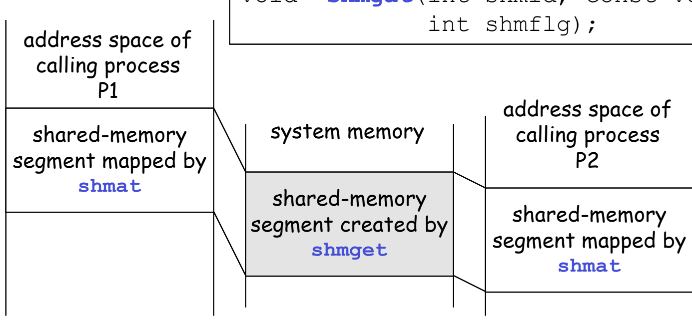
POSIX Shared Memory

```
#include <sys/shm.h>

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

Ok, we have created a shared-memory segment. Now what?

```
void *shmgat(int shmid, const void *shmaddr,
             int shmflg);
```



POSIX Semaphore Sets

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
/* Create semaphore set with nsems semaphores.
   If set exists, nsems can be zero. */
```

```
#include <sys/sem.h>
#define PERMS (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
#define SET_SIZE 2

int main (int argc, char * argv[]) {
  key_t mykey;
  int   semid;

  mykey = ftok(argv[1], atoi(argv[2]));
  semid = semget(mykey, SET_SIZE, PERMS | IPC_CREAT)
  return 0;
}
```

Semaphore Set Control

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, …);
```

command	description
SETVAL	set value of a specific semaphore element to `arg.val`
SETALL	set values of semaphore set from `arg.array`
GETVAL	return value of specific semaphore element
GETALL	return values of the semaphore set in `arg.array`
GETPID	return process id of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
etc.…	

Semaphore Set Operations

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
/* The operations are defined in the array pointed to
   by 'sops'. */
```

`struct sembuf` contains **approximately** the following members:

`short sem_num`	number of semaphore element
`short sem_op`	operation to be performed
`short sem_flg`	specific options for the operation

sem_op > 0	add the value to the semaphore element and awaken all processes that are waiting for element to increase
(sem_op == 0) && (semval != 0)	block calling process (waiting for 0) and increement count of processes waiting for 0.
sem_op < 0	add sem_op value to semaphore element value provided that result would not be negative. If result negative, block process on event that semaphore value increases. If result == 0, wake processes waiting for 0.

Semaphore Operations: Example

Mutually-Exclusive Access to Two Tapes:

```
/* pseudo code */
struct sembuf get_tapes[2];
struct sembuf release_tapes[2];

setsembuf(&(get_tapes[0]),      0, -1, 0);
setsembuf(&(get_tapes[1]),      1, -1, 0);
setsembuf(&(release_tapes[0]), 0, +1, 0);
setsembuf(&(release_tapes[1]), 1, +1, 0);

/* Process 1: */     semop(S, get_tapes, 1);
                     <use Tape 0>
                     semop(S, release_tapes, 1);

/* Process 2: */     semop(S, get_tapes, 2);
                     <use both tapes 0 and 1>
                     semop(S, release_tapes, 2);
```

Semaphore Operations: Another Example

Semaphore to control access to critical section:

```
int main(int argc, char * argv[]) {
  int semid;
  struct sembuf sem_signal[1];
  struct sembuf sem_wait[1];

  semid = semget(IPC_PRIVATE, 1, PERMS);
  setsembuf(sem_wait, 0, -1, 0);
  setsembuf(sem_signal, 0, 1, 0);
  init_element(semid, 0, 1); /* Set value of element 0 to 1 */

  for (int i = 1; i < n; i++) if fork() break;

  semop(semid, sem_wait, 1); /* enter critical section */
  /* in critical section */
  semop(semid, sem_signal, 1); /* leave critical section */
  /* in remainder section */
  return 0;
}
```