Operating Systems

CSE 231

Instructor: Sambuddho Chakravarty

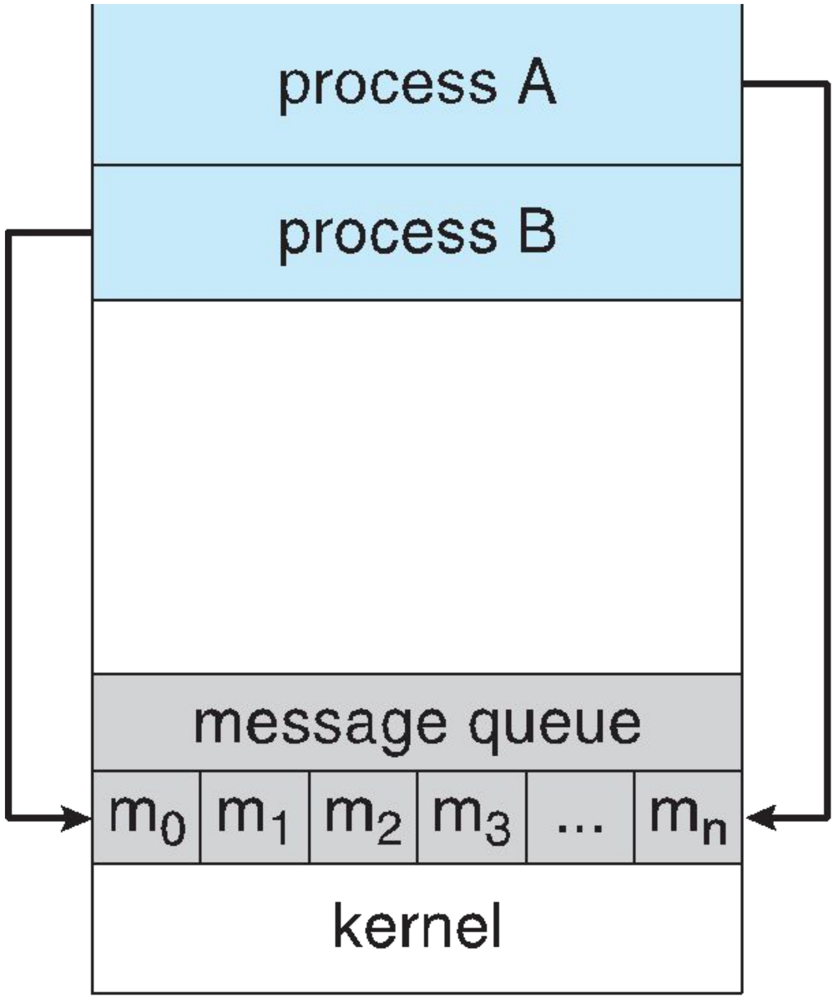
(Semester: Winter 2018)

Week 5: Jan 29 – Feb 2

Interprocess Communication

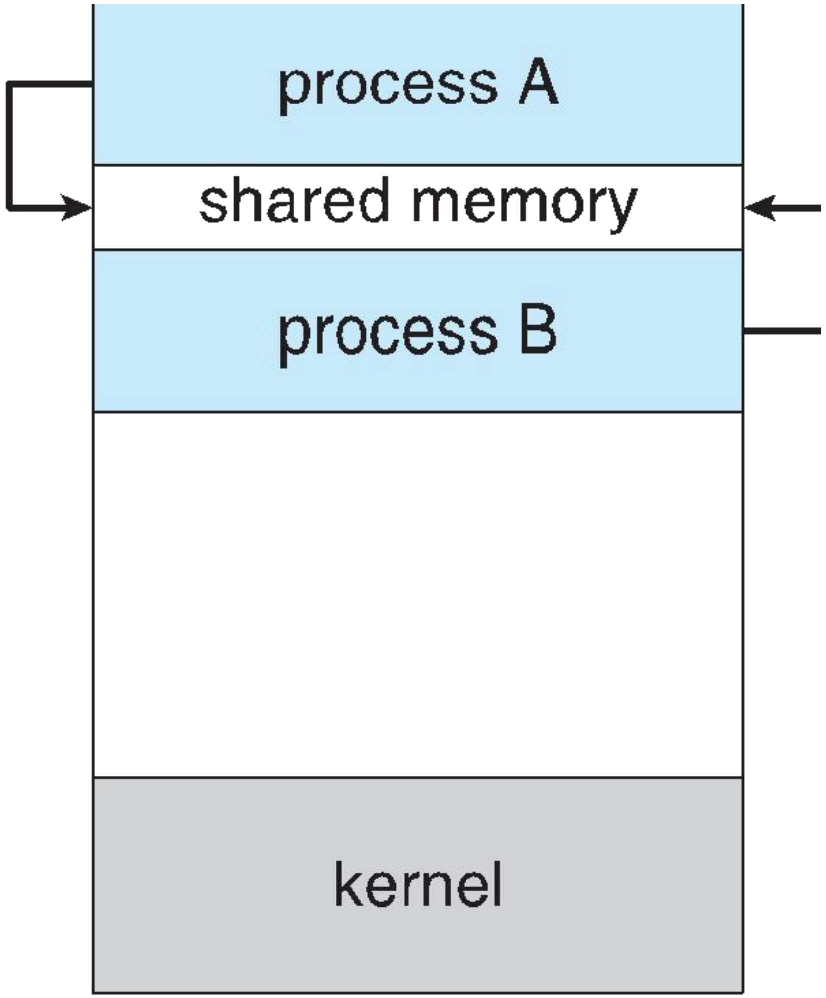
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models



(a)

Message Passing



(b)

Shared Memory

Message Passing v.s. Shared Memory

- Message passing
  - Why good? Simpler. All sharing is explicit
  - Why bad? Overhead. Data copying, cross protection domains

- Shared Memory
  - Why good? Performance. Set up shared memory once, then access w/o crossing protection domains
  - Why bad? Synchronization

Signals

What is signal?

- A signal is an event generated by the UNIX system in response to some condition, upon which a process may in turntake some action.
- Signals are generated by some error conditions, such as memory segment violations, floating point processor errors or il-legal instructions. They are generated by the shell and terminal handlers to cause interrupts.

Signals

Generally, signals can be generated, caught and acted upon, or ignored.

- Signal names are defined by including the header file < signal.h > as followed:
- SIGABORT: Process abort;
- SIGALRM: Alarm clock;
- SIGFPE: Floating point exception
- SIGHUP: Hangup
- SIGILL: illegal instruction
- SIGINT: Terminal interrupt.
- SIGKILL: kill (can’t be caught or ignored)
- SIGPIPE: write on a pipe with no reader
- SIGQUIT: Termianl quit
- SIGSEGV: Invalid memory segment access
- SIGTERM: Termination
- SIGUSR1: User-defined signal 1
- SIGUSR2: User-defined signal 2

Signal Concepts

- Signals are defined in <signal.h>
- man 7 signal for complete list of signals and their numeric values
- kill –l for full list of signals on a system
  - 64 signals. The first 32 are traditional signals, the rest are for real time applications

Signals

- Programming interface of signal handling

```
#include <signal.h>
void (*signal(int sig, void(*func)(int)));
```

It says that signal is a function that takes two parameters – sig and func

POSIX Signal Handling

C90 standard

- Defines **`signal()`** and **`raise()`** functions
  - Work across all systems (UNIX, LINUX, Windows), but…
  - Work **differently** across some systems!!!
    - On some systems, signals are blocked during execution of handler for that type of signal -- but not so on other (older) systems
    - On some (older) systems, handler installation for signals of type x is cancelled after first signal of type x is received; must reinstall the handler -- but not so on other systems
- Does not provide mechanism to block signals in general

POSIX Signal Handling

POSIX standard

- Defines `kill()`, `sigprocmask()`, and `sigaction()` functions
  - Work the same across all POSIX-compliant UNIX systems (Linux, Solaris, etc.), but…
  - Do not work on non-UNIX systems (e.g. Windows)
- Provides mechanism to block signals in general

Blocking Signals in General

Each process has a signal mask in the kernel

- OS uses the mask to decide which signals to deliver
- User program can modify mask with **`sigprocmask()`**

`sigprocmask()`

```
int sigprocmask(int iHow,
                const sigset_t *psSet,
                sigset_t *psOldSet);
```

- **`psSet`**: Pointer to a signal set
- **`psOldSet`**: (Irrelevant for our purposes)
- **`iHow`**: How to modify the signal mask
    - **`SIG_BLOCK:`** Add **`psSet`** to the current mask
    - **`SIG_UNBLOCK:`** Remove **`psSet`** from the current mask
    - **`SIG_SETMASK:`** Install **`psSet`** as the signal mask
- Returns 0 iff successful

Functions for constructing signal sets

- **`sigemptyset(), sigaddset(),`** …

Blocking Signals Example

```
sigset_t sSet;
int main(void) {
   int iRet;
   sigemptyset(&sSet);
   sigaddset(&sSet, SIGINT);
   sigprocmask(SIG_BLOCK, &sSet, NULL);
   …
   sigprocmask(SIG_UNBLOCK, &sSet, NULL);
   assert(iRet == 0);
   …
}
```

Installing a Signal Handler

`sigaction()`

```
int sigaction(int iSig,
              const struct sigaction *psAction,
              struct sigaction *psOldAction);
```

- **`iSig`**: The type of signal to be affected
- **`psAction`**: Pointer to a structure containing instructions on how to handle signals of type **`iSig`**, including signal handler name and which signal types should be blocked
- **`psOldAction`**: (Irrelevant for our purposes)
- Installs an appropriate handler
- Automatically blocks signals of type **`iSig`**
- Returns 0 iff successful

Note: More powerful than C90 **`signal()`**

Installing a Handler Example

Program testsigaction.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Installing a Handler Example (cont.)

Program testsigaction.c (cont.):

```
…
int main(void) {
   int iRet;
   struct sigaction sAction;
   sAction.sa_flags = 0;
   sAction.sa_handler = myHandler;
   sigemptyset(&sAction.sa_mask);
   sigaction(SIGINT, &sAction, NULL);

   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0;
}
```

Alarm Example 2

Program testalarmtimeout.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
   printf("\nSorry.  You took too long.\n");
   exit(EXIT_FAILURE);
}
```

Interval Timers

`setitimer()`

```
int setitimer(int iWhich,
              const struct itimerval *psValue,
              struct itimerval *psOldValue);
```

- Sends SIGALRM
- Timing is specified by **`psValue`**
- **`psOldValue`** is irrelevant for our purposes
- Uses **virtual time**, alias **CPU time**
  - Time spent executing other processes does not count
  - Time spent waiting for user input does not count
- Returns 0 iff successful

Interval Timer Example

Program testitimer.c:

```
##include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/time.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
…
int main(void)
{
   int iRet;
   void (*pfRet)(int);
   struct itimerval sTimer;

   pfRet = signal(SIGPROF, myHandler);
   assert(pfRet != SIG_ERR);
   …
```

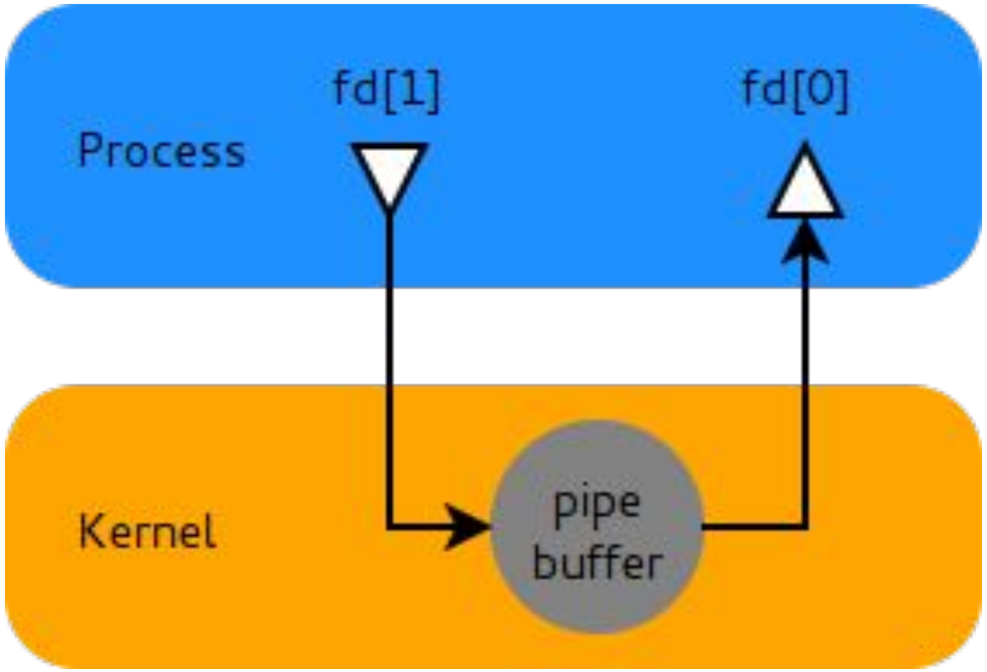
Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
   /* Send first signal in 1 second, 0 microseconds.
*/
   sTimer.it_value.tv_sec = 1;
   sTimer.it_value.tv_usec = 0;

   /* Send subsequent signals in 1 second,
      0 microseconds intervals. */
   sTimer.it_interval.tv_sec = 1;
   sTimer.it_interval.tv_usec = 0;

   iRet = setitimer(ITIMER_PROF, &sTimer, NULL);
   assert(iRet != -1);

   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0;
}
```

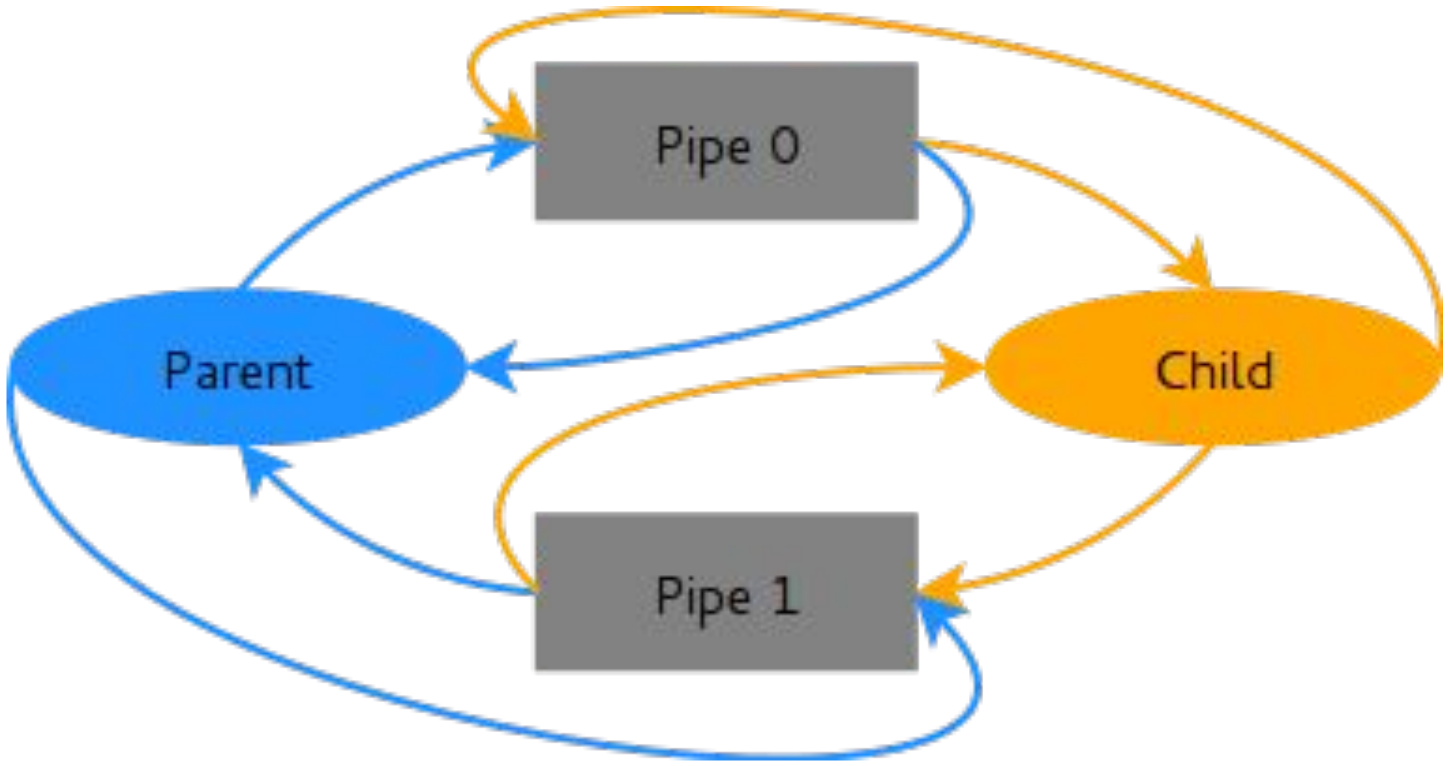
IPC using Pipes

- Communication channel between two processes. Like what happens when you type – cat foo | grep bar. The communication between the *stdout* of foo to the *stdin* of grep



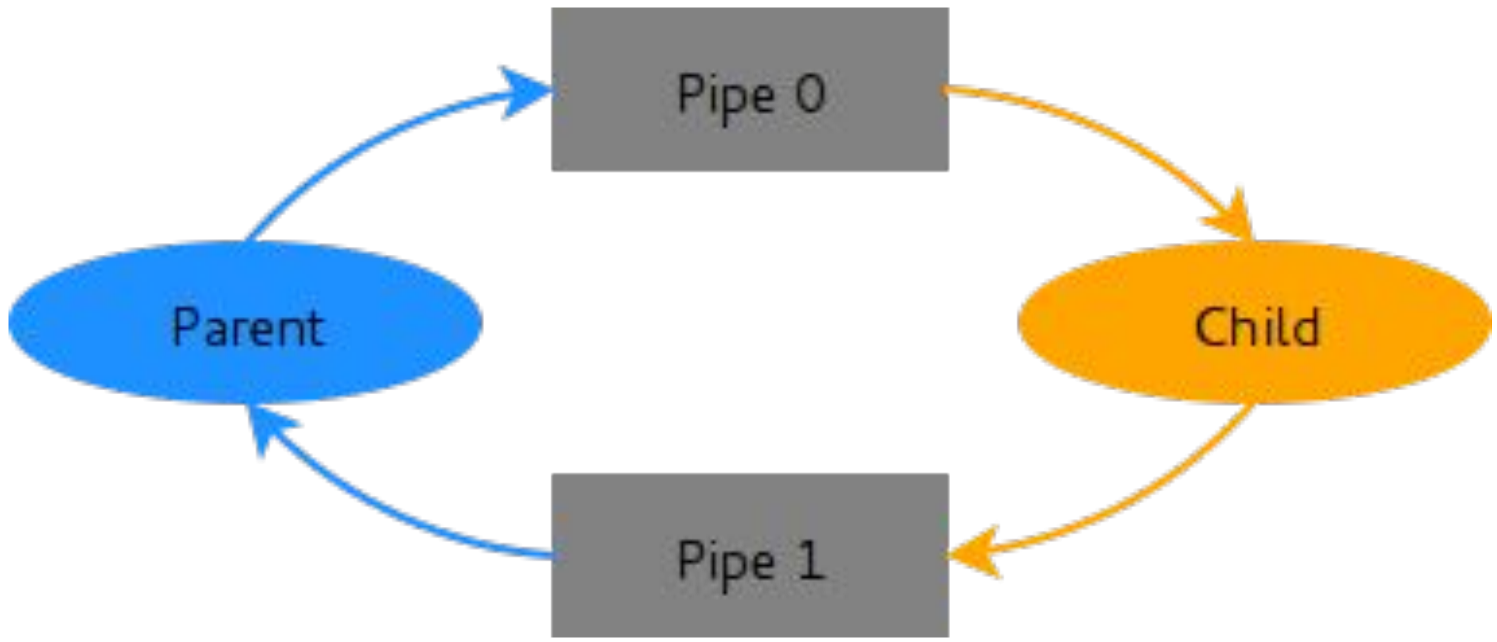
IPC using Pipes

- What happens when a process fork()s while still having pipes.



IPC using Pipes

- What you really want is this:



IPC Example: Unix pipe

- int pipe(int fd[2]);
    - Returns two file descriptors in fd[0] and fd[1];
    - Writes to fd[1] will be read on fd[0]
    - When last copy of fd[1] closed, fd[0] will return EOF
    - Return 0 on success, -1 on error

- Operations on pipes:
    - read/write/close --- as with files
    - When fd[1] closed, read(fd[0]) returns 0 bytes
    - When fd[0] closed, write(fd[1]):
        - Kills process with SIGPIPE, or if blocked
        - Failes with EPIPE

IPC Example: Unix pipe (cont.)

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

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; }
```

Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Named Pipes – aka FIFOs

- System primitives used for implementing IPC when no two processes are related.
- Pipe()s work when the parents are directly related to one another – parent/child.
- What if the two processes are not related to one another.

Named Pipes – aka FIFOs

- Creating a FIFO

mknod("myfifo", S_IFIFO | 0644 , 0);

``myfifo'': name of the FIFO

S_FIFO: Create a FIFO

S_FIFO|0644: add permissions : rw, r, r

Named Pipes – aka FIFOs

```
#define FIFO_NAME “fifo_1"
int main(void)
 { char s[300];
int num, fd;
mknod(FIFO_NAME, S_IFIFO | 0666, 0);
printf("waiting for readers...\n");
 fd = open(FIFO_NAME, O_WRONLY);
printf("got a reader--type some stuff\n");
while (gets(s), !feof(stdin))
{
if ((num = write(fd, s, strlen(s))) == -1)
 perror("write");
else printf("speak: wrote %d bytes\n", num);
}
return 0;
}
```

```
#define FIFO_NAME “fifo_1"
int main(void)
 { char s[300];
int num, fd;
mknod(FIFO_NAME, S_IFIFO | 0666, 0);
printf("waiting for writers...\n");
 fd = open(FIFO_NAME, O_WRONLY);
printf("got a reader--type some stuff\n");
do
{
if ((num = read(fd, s, 300)) == -1)
  perror("read");
  else
    { s[num] = '\0'; printf("read %d bytes: \"%s\"\n", num, s);
 }
}  while (num>0);
return 0;
}
```

Shared Memory – Memory section that can be shared between different processes

- Create a shared memory ID, much like a file descriptor.

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


key_t key;
int shmid;
key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Shared Memory – Attach to an address

- Attach to memory location or let the kernel decide one for you
- void *shmat(int shmid, void *shmaddr, int shmflg);

```
key_t key;
int shmid;
char *data;
key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

You may read or write to data like you do with any memory location.

Once you are done with the shared memory you may need to detach it.

```
int shmdt(void *shmaddr);
```

But is not yet destroyed. To actually destroy it you need to actually delete it

```
shmctl(shmid, IPC_RMID, NULL);
```

Shared Memory – with multiple processes

```
#define SHMSIZE 27 int main()
 {
 int shmid;
char *shm;
 if(fork() == 0)
  { shmid = shmget(2009, SHMSIZE, 0);
    shm = shmat(shmid, 0, 0);
    char *s = (char *) shm; *s = '\0'; int i;
    for(i=0; i<5; i++)
     { int n;
       printf("Enter number<%i>: ", i);     scanf("%d", &n);
sprintf(s, "%s%d", s, n);
} strcat(s, "\n");
printf ("Child wrote <%s>\n",shm); shmdt(shm);
}
else {
shmid = shmget(2009, SHMSIZE, 0666 | IPC_CREAT);
shm = shmat(shmid, 0, 0);
wait(NULL);
printf ("Parent reads <%s>\n",shm) ;
shmdt(shm);
shmctl(shmid, IPC_RMID, NULL); }
return 0; }
```