Lecture 08: Inter-process Communication in Shared Memory

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

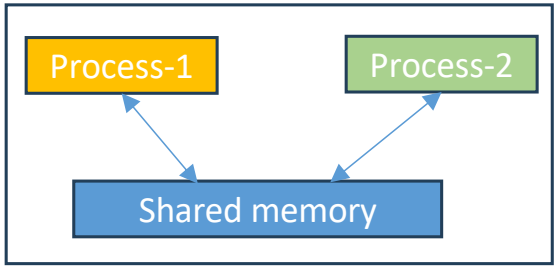
vivekk@iiitd.ac.in

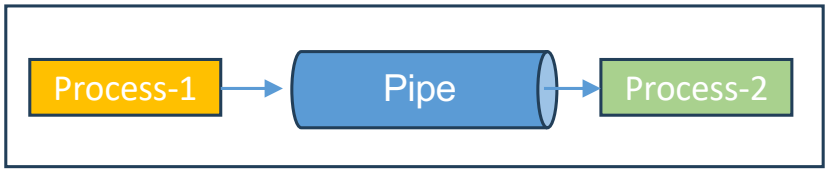
Last Lecture

Today's Class

- Inter-process communication
  - Signals
  - Pipes
  - Shared memory

Inter-Process Communication





- Recall, forked process calls an exec after which it has its own address space that is not shared with the parent
- IPC mechanisms to share information between processes

Points to Ponder

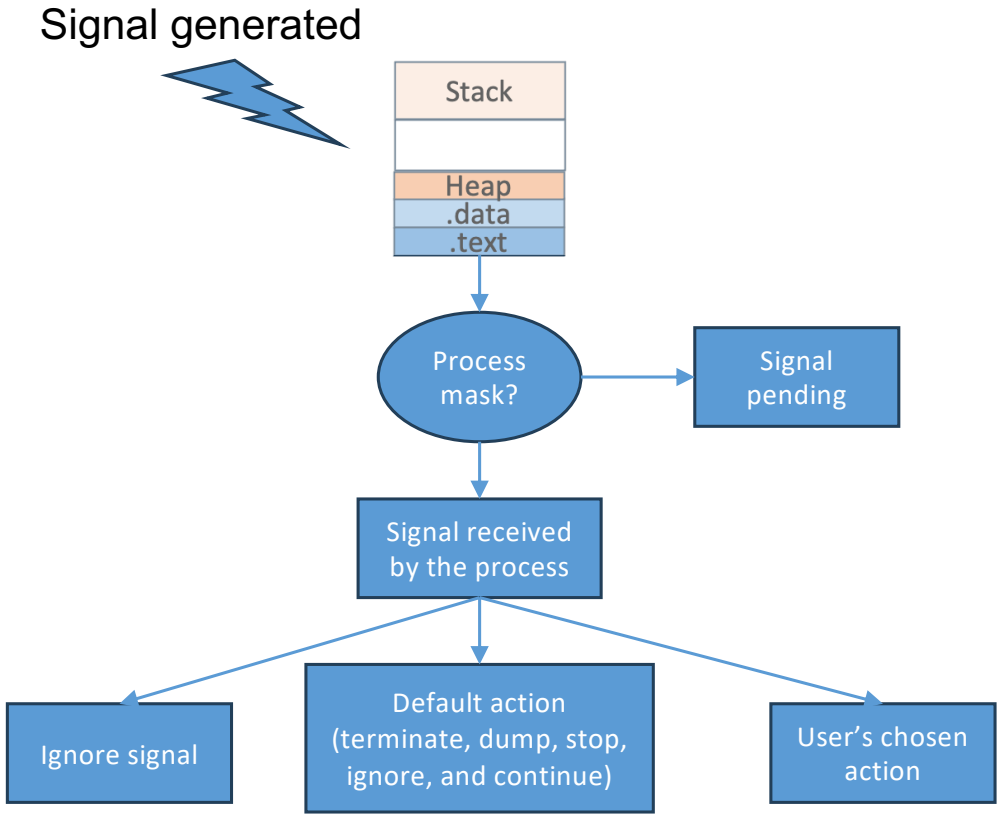
```
vivek@possum:~/os23$ vi infinite-fib.c
vivek@possum:~/os23$ gcc infinite-fib.c
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
^C
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
30
Fib(30) = 832040
Input i:
^Z
[1]+  Stopped                 ./a.out
vivek@possum:~/os23$ fg
./a.out
23
Fib(23) = 28657
Input i:
```

- What happens when we press the following key combination on a shell that is running some program
  - Ctrl-c
    - Process simply terminates even though it was executing some instruction inside the .text segment
  - Ctrl-z
    - Process moves to background (you can get it back to foreground using “fg”)
  - How come these actions are happening even though the program did not have any code to demonstrate the above mentioned behavior
- How does a child process that died inform about its fate to its parent?
- What happens when a program attempts to access an invalid memory address?

Signals – A Limited Form of IPC

- Signals (interrupts) are technique used by the OS to communicate with a process
- Every signal has a name that has a unique number assigned
  - Ctrl-c and Ctrl-z generates hardware interrupt from the keyboard that is handled by the OS by sending SIGINT (2) and SIGSTOP (20) signal, respectively, to the running process
- Each signal type has a default handler
  - A program can install its own handler for signals of any type
    - Except SIGKILL (default action is to exit the process) and SIGSTOP (default action is to suspend the process)
- States
  - Generated – due to some event
  - Delivered – received by the process
  - Blocked – process has blocked the signal
  - Pending – to be delivered
  - Caught – action associated with signal has been taken by the process

Signal Delivery and Handling



- Process can use sigprocmask function to make changes to its signal mask (member in task_struct) if it wants to temporarily block a set of signals (not all signals can be blocked, e.g., SIGKILL and SIGSTOP)
- User can create his own signal handler function (we will see soon)
- Signal name has prefix "SIG". Some of the signals that you can easily come across are:
  - SIGINT (ctrl-c), SIGQUIT (ctrl-\), SIGSTOP (ctrl-z), SIGTRAP (breakpoint), SIGSEV (segmentation fault), SIGTERM (process termination), SIGCHLD (child stopped/terminated), SIGFPE (floating point exception), etc.

Catching the Signal

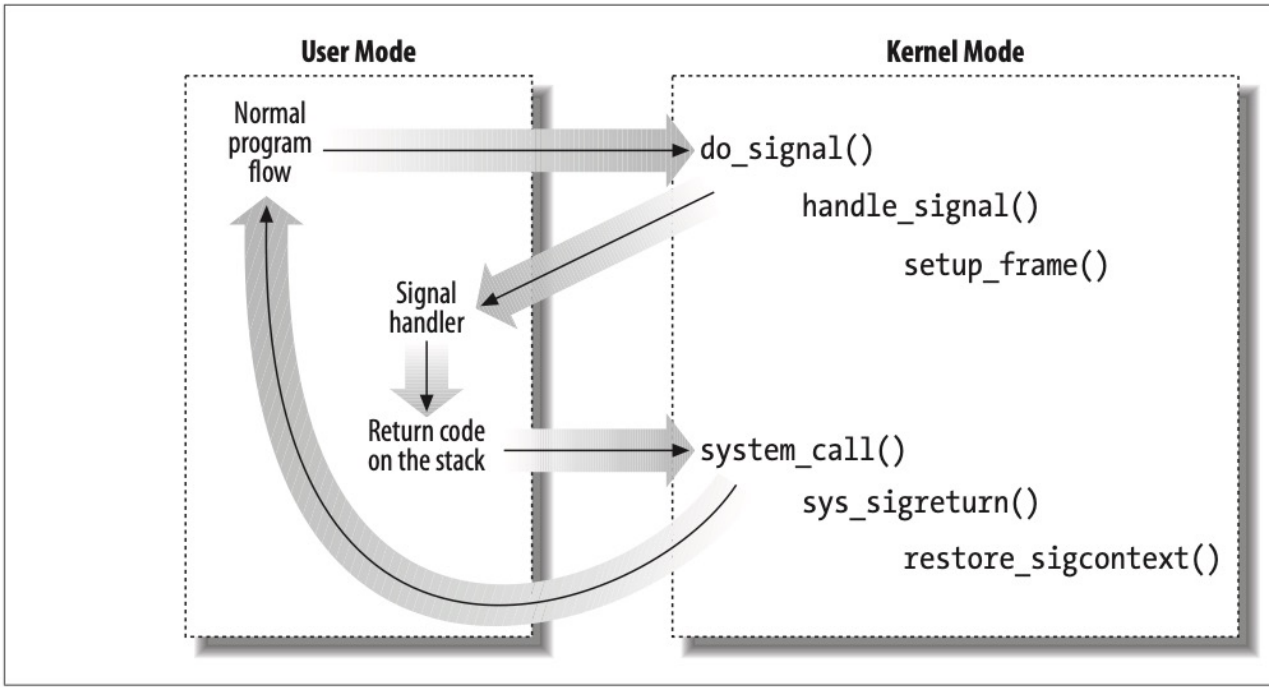


Figure 11-2. Catching a signal

- In the absence of user signal handler, signal is handled in kernel mode, otherwise it is handled in the user mode
- Process switches into kernel mode after receiving a non-blocking signal
- Kernel handle the signal by first setting up stack frame on user stack (save context)
- Control is passed to user stack and user signal handler is executed
- Control is returned back into kernel mode which restores the user stack to its original state to resume the execution of the program
  - **sigreturn** system call used to restore process state

Figure source: from the book "Understanding the Linux Kernel"

Programming Signals

```
static void my_handler(int signum) {
  static int counter = 0;
  if(signum == SIGINT) {
    char buff1[23] = "\nCaught SIGINT signal\n";
    write(STDOUT_FILENO, buff1, 23);
    if(counter++=1) {
      char buff2[20] = "Cannot handle more\n";
      write(STDOUT_FILENO, buff2, 20);
      exit(0);
    }
  } else if (signum == SIGCHLD) {
    char buff1[23] = "Caught SIGCHLD signal\n";
    write(STDOUT_FILENO, buff1, 23);
  }
}
```

```
int main() {
  struct sigaction sig;
  memset(&sig, 0, sizeof(sig));
  sig.sa_handler = my_handler;
  sigaction(SIGINT, &sig, NULL);
  sigaction(SIGCHLD, &sig, NULL);
  int n;
  while(1) {
    printf("Input i: \n");
    scanf("%d",&n);
    if(fork()==0) {
      printf("Fib(%d) = %d\n",n,fib(n));
      exit(0);
    }
    else wait(NULL);
  }
  return 0;
}
```

Why Fib(20) is calculated twice?

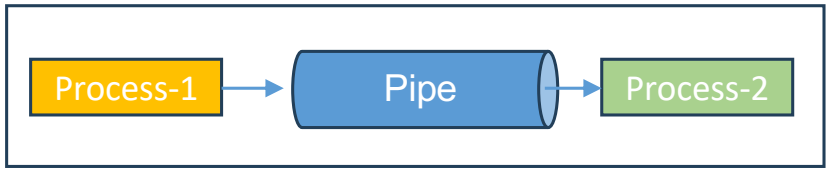
```
vivek@possum:~/os23$ ./a.out
Input i:
10
Fib(10) = 55
Caught SIGCHLD signal
Input i:
20
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Cannot handle more
```

- sigaction is used to change the default action associated with a signal
  - Except SIGKILL and SIGSTOP
- User can assign his own signal handler method that would get invoked when the assigned signal is received by the process
- Only asynchronous safe functions should be called inside signal handler
  https://man7.org/linux/man-pages/man7/signal-safety.7.html

Today's Class

- Inter-process communication
  - Signals
  - Pipes
  - Shared memory

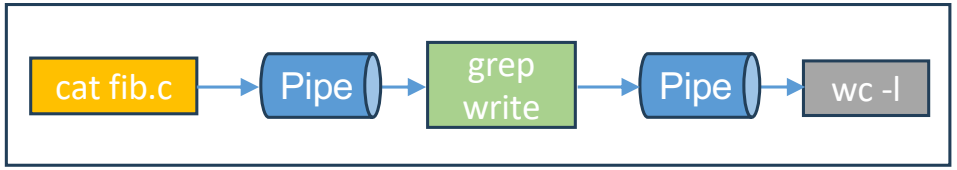
IPC Using Pipes



Processes **must** be running on the **SAME** machine

- Pipes (analogous to water pipe) is a unidirectional stream of data flowing from a source process to a destination process
  - Kernel buffer exposed to processes as a pair of file descriptors (readable end and writable end). Default buffer size is 16 pages (16x4096 bytes)
  - Data delivered in the same order as sent
  - Uses blocking IO and the writer process will block if the pipe is full
- We use it frequently on Shell
  - Better than using temporary files as pipes automatically clean up themselves unlike files that must be explicitly removed using the “rm” command on Shell

```
vivek@possum:~/os23$ cat fib.c | wc -c
1788
vivek@possum:~/os23$ cat fib.c | wc -l
77
vivek@possum:~/os23$ cat fib.c | grep write | wc -l
10
```



Programming With pipe

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        char buff[] = "Hello my dear good Parent";
        sleep(2);
        write(fd[1], buff, sizeof(buff));
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- pipe expects an int array of size two only for a readable and writable file descriptor
- Reader and writer processes must close the pipe end they are not going to use
  - It is very important as fork duplicates the open file descriptors in the parent process
- What would happen if the child sleeps before writing to the pipe, as the parent has already issued the read?

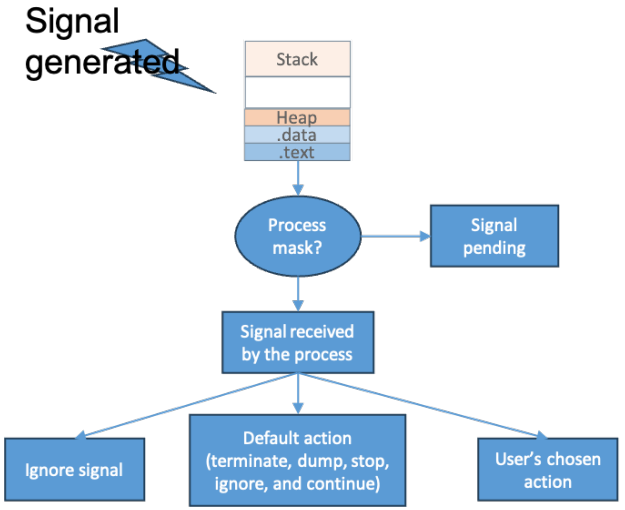
Programming With pipe and dup

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- dup2 can be used to duplicate a file descriptor
  - E.g., duplicate one of the end of the pipe as STDOUT or STDIN
    - Duplicating to STDOUT will cause printf to print to the pipe instead of the STDOUT
- Used by the Shell when we pipe the output of one command to another command

Last Lecture

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```




```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- Child process should call **exit** system call before terminating to pass its exit status to the parent process
  - Parent can retrieve it by calling **wait**
- Zombies and orphan processes
- The exec system call
- IPC using signaling
- IPC using pipe and dup

Today's Class

- Inter-process communication
  - Signals
  - Pipes
  - Shared memory

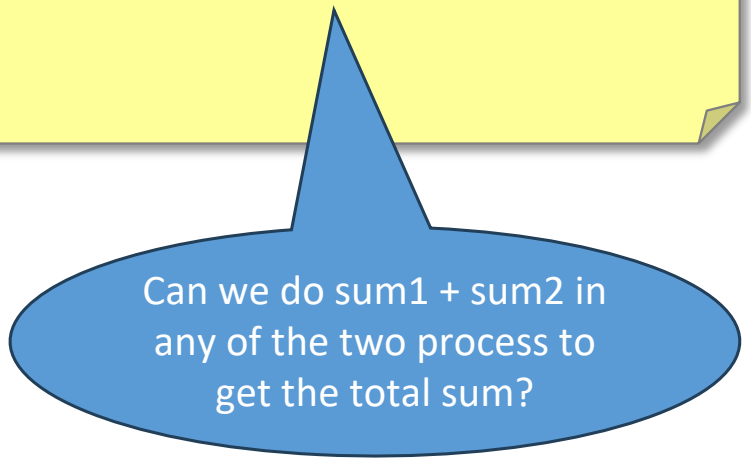
Array Sum Program

```
int main() {
    int A[SIZE]; //initialized
    int sum=0;
    for(int i=0; i<SIZE; i++) sum += A[i];
    return 0;
}
```

- A simple program to calculate the sum of all the elements inside a huge array
- How to shorten the execution time for calculating the sum for a very big array?

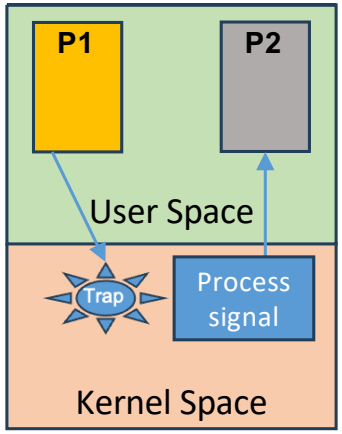
Array Sum Program

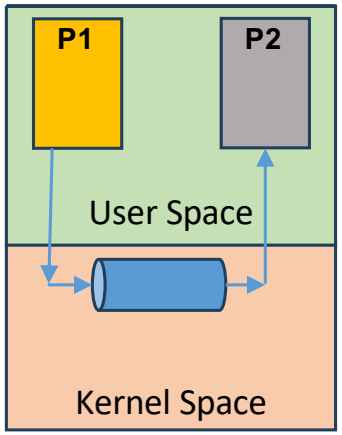
```
int main() {
    int A[SIZE]; //initialized
    int sum1=0, sum2=0;
    if(fork() == 0) {
        for(int i=SIZE/2; i<SIZE; i++) sum1 += A[i];
        exit(0);
    }
    for(int i=0; i<SIZE/2; i++) sum2 += A[i];
    wait(NULL);
    return 0;
}
```



- We can fork a child and let the parent and child calculate the sum of left and right half of the array, respectively
- The problem(s)
  - How the child and parent would communicate their partial results to get the total sum?
    - Can the child pass the partial sum as its exit code?
      - exit API accepts only "int" type parameter. We cannot pass the sum of "double" type array, etc.
  - Can we setup a pipe between the parent and child?
    - Yes, but what if there are several children to improve the execution time further? We will have to use multiple pipes and keep track of them (closing fds differently at child v/s parent, etc.)
  - What if its some other kind of program, such as vector addition or matrix multiplication?
    - Three arrays (or matrices) would require accesses by child and parent

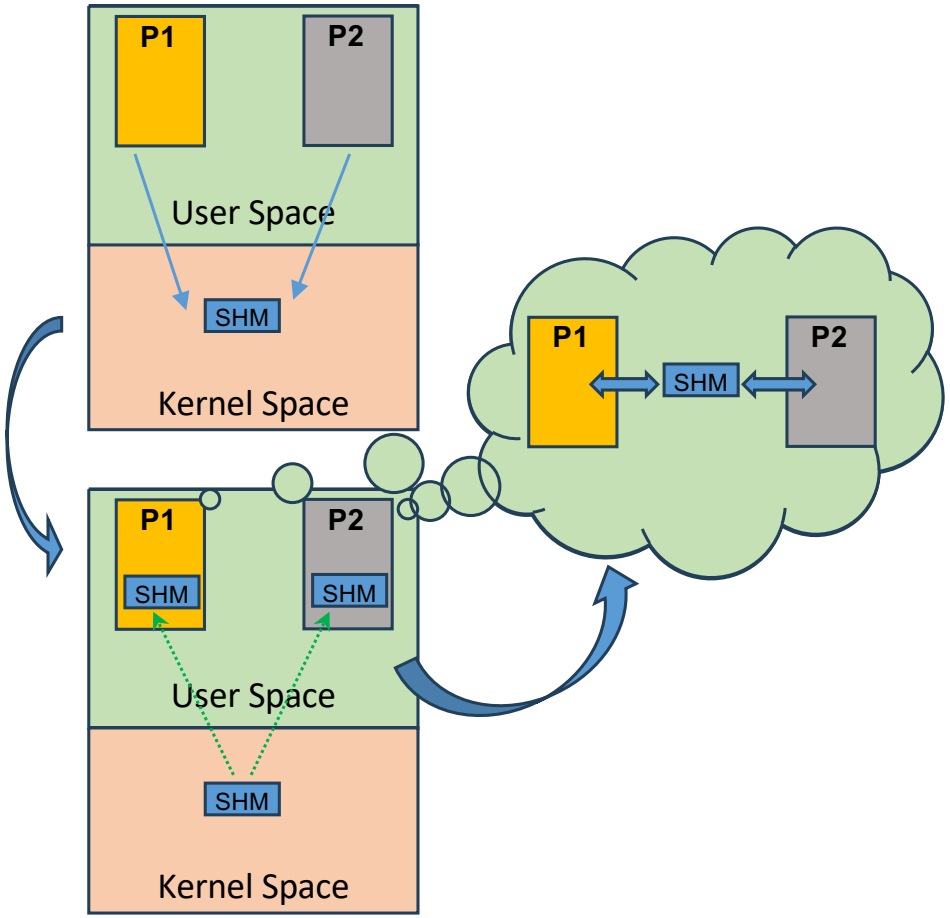
IPC Using Signals and Pipes





- Processes running inside a single machine can use simple IPC techniques such as signaling and pipes
- Issues with signaling
  - Limited form of IPC
  - Requires help from OS for every signaling instances
  - Transition between user space and kernel space (overheads)
- Issues with pipes
  - Limited buffer size (although significantly better than signaling)
  - Use of system calls for every communication
    - Transition between user space and kernel space (overheads)
  - One way communication
  - Blocking communication

IPC Using POSIX Shared Memory



- Shared memory is the fastest form of IPC in a single machine in which processes can asynchronously write to a shared region of memory **without** the need for switching to kernel space
  - User specified memory size
- Access to the shared memory is achieved as follows
  1. Create and open a new shared memory object using shm_open system call in the parent process
  2. Set the desired size for the newly created shared memory region using ftruncate
  3. Map the shared memory region into the process's address space using mmap (parent process calls mmap that gets shared in child's address space after fork)
  4. Both child and the parent processes un-maps the shared memory from their address space upon completion by using munmap system call, followed by closing the file descriptor for the shared memory object (close system call)
  5. Parent deletes the shared memory object by using shm_unlink
- Can be used for parallel programming!

Array Sum Program

```
typedef struct shm_t {
    int A[SIZE];
    int sum1;
    int sum2;
} shm_t;
```

```
int main() {









    return 0;
}
```

- We will create a shared memory segment of our own type shm_t that contains the array and a variable sum to store the partial result

Array Sum Program

```
int main() {
    shm_t* shm = setup();

    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum1;
    int sum2;
} shm_t;
```

1) shm_open (...)
2) ftruncate (...)
3) shm = mmap (...)

1) munmap (shm, ...)
2) close (...)
3) shm_unlink (...)

- The three steps for setting up a shared memory segment are:
  - Creating a shared memory object (shm_open)
  - Assigning a size for the shared memory object (ftruncate)
  - Mapping the shared memory into the process's address space (mmap)
- The three steps in cleanup are:
  - Removing the mapping of shared memory from process's address space
  - Closing the file descriptor for the shared memory object
  - Deleting the shared memory object

Array Sum Program

```
int main() {
    shm_t* shm = setup();
    if(fork()==0) {



        cleanup_and_exit();
    } else {



        wait(NULL);
    }

    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum1;
    int sum2;
} shm_t;
```

1) munmap (shm, ...)
2) close (...)
3) exit(0)

- Parent process will fork children that gets a copy of the parent's address space (inherits the file descriptor and shared memory object from the parent)
- Child process also carries out cleanup similar to parent except for calling **shm_unlink** that is called only by the parent
- If parent process fail to call shm_unlink, then the **shm_open** will fail the next time (unless you change the name of the memory region)
- Child must also call **exit** in their cleanup call
  - **Why?**

Array Sum Program (Version 1)

```
typedef struct shm_t {
    int A[SIZE];
    int sum1;
    int sum2;
} shm_t;
```

```
int main() {
    shm_t* shm = setup();
    if(fork()==0) {
        int local=0;
        for(int i=0; i<SIZE/2; i++) local += shm->array[i];
        shm->sum1 += local;
        cleanup_and_exit();
    } else {
        int local=0;
        for(int i=SIZE/2; i<SIZE; i++) local += shm->array[i];
        shm->sum2 += local;
        wait(NULL);
    }
    int total = shm->sum1 + shm->sum2;
    cleanup();
    return 0;
}
```

- Both child and the parent process calculate the partial sum independently
- Parent process combine the partial sum from the child to its own calculation to get the total
  - Done only after **wait**

Next Lecture

- Semaphores