Chapter 3

Processes

(Part 2)

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

- Process Concept
- Process State and Scheduling
- Process API
- Signal
- Inter Process Communication

Signal

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived.
  - Each signal has a current disposition (action associated with a signal), which determines how the process behaves when it is delivered the signal.

Signal Types

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctrl-c)
8	SIGFPE	Terminate & Dump	Erroneous arithmetic operation (ex. divide by zero)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

```
yunmin@yunmin:~/workspace/ch3$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
```

Sending a Signal

- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process.
- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (`SIGFPE`) or the termination of a child process (`SIGCHLD`)
  - Another process has invoked the `kill()` system call to explicitly request the kernel to send a signal to the destination process.

Receiving a Signal

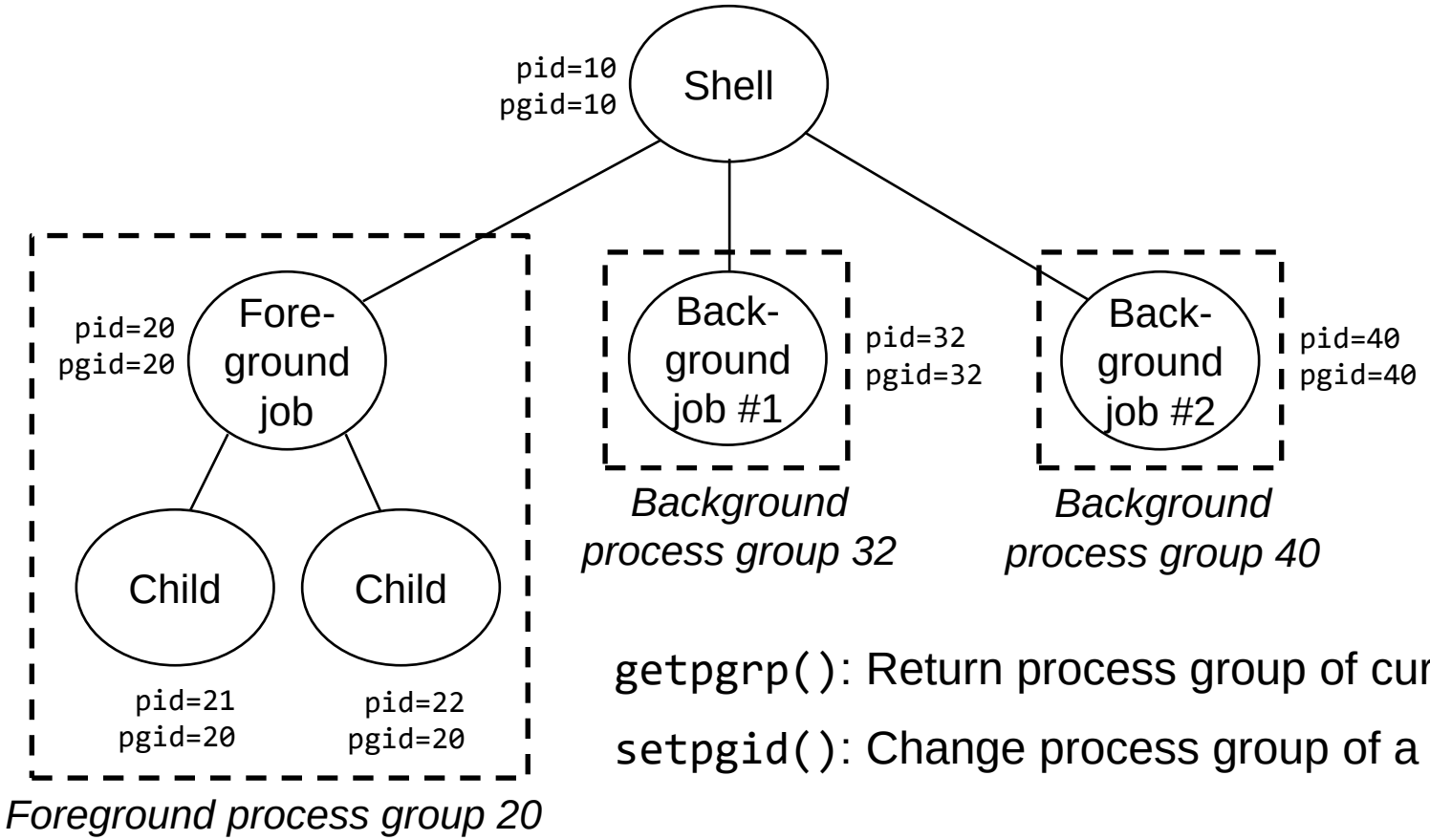
- A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.
- **Signal disposition**: Each signal has a default disposition, which determines how the process behaves when it is delivered the signal
  - Terminate the process
  - Core dump and terminate the process
  - Ignore the signal
  - Stop the process
  - Continue the process if it is currently stopped
- Also a process can change the disposition of a signal
  - Process executes a user-level function called a **signal handler** when the specified signal occurs.

Signal Pending and Blocked

- A signal is pending if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process can block the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.
- Kernel maintains pending and blocked bit vectors in the context of each process.

Process Groups

- Every process belongs to exactly one process group



`getpgrp()`: Return process group of current process

`setpgid()`: Change process group of a process

Kill Program

- kill program sends arbitrary signal to a process or process group

killproc1.c

```
int main()
{
    if (fork() == 0) {
        printf("Child1: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
        if (fork() == 0)
            printf("Child2: pid=%d pgrp=%d\n",
                    getpid(), getpgrp());
        while(1);
    }
    return 0;
}
```

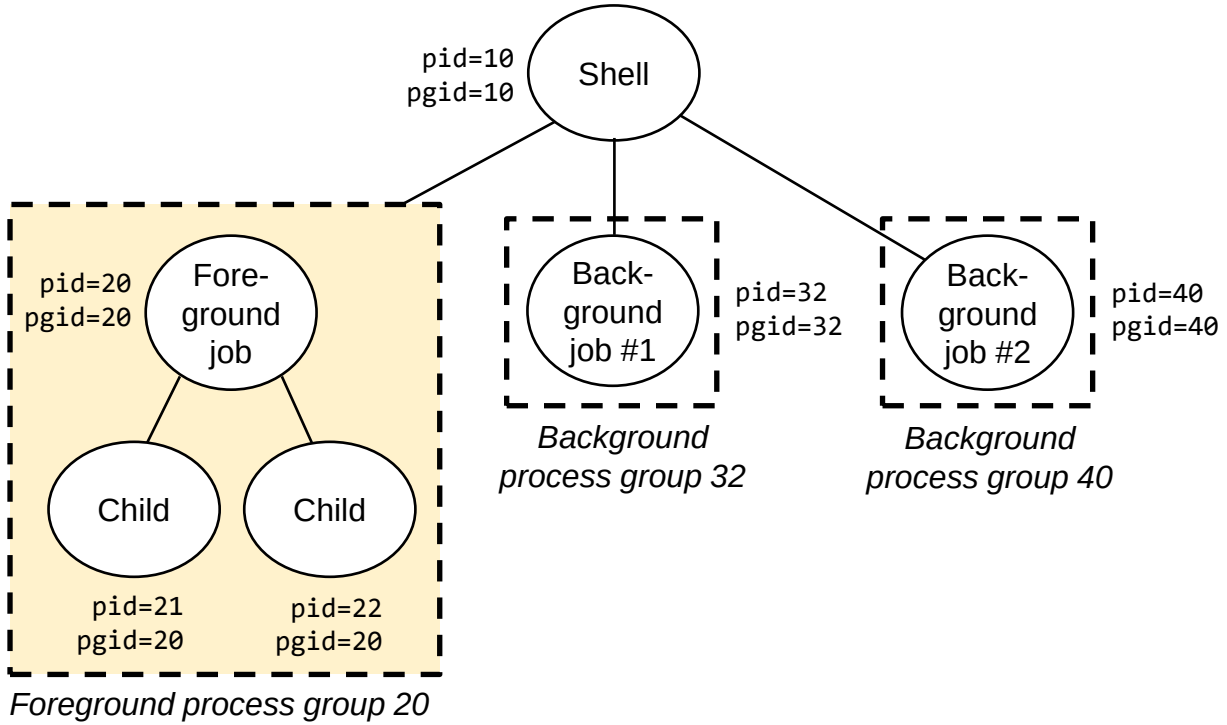
- $ kill -9 4009
  - Send SIGKILL to process 4009
- $ kill -9 -4224
  - Send SIGKILL to every process in process group 4224

```
yunmin@yunmin:~/ch3_2$ ./killproc1
Child1: pid=4009 pgrp=4008
Child2: pid=4010 pgrp=4008
yunmin@yunmin:~/ch3_2$ ps
    PID TTY          TIME CMD
   3605 pts/0    00:00:00 bash
   4009 pts/0    00:00:01 killproc1
   4010 pts/0    00:00:01 killproc1
   4023 pts/0    00:00:00 ps
yunmin@yunmin:~/ch3_2$ kill -9 4009
yunmin@yunmin:~/ch3_2$ ps
    PID TTY          TIME CMD
   3605 pts/0    00:00:00 bash
   4010 pts/0    00:00:10 killproc1
   4069 pts/0    00:00:00 ps
yunmin@yunmin:~/ch3_2$ kill -9 4010
yunmin@yunmin:~/ch3_2$ ps
    PID TTY          TIME CMD
   3605 pts/0    00:00:00 bash
   4086 pts/0    00:00:00 ps
```

```
yunmin@yunmin:~/ch3_2$ ./killproc1
Child1: pid=4225 pgrp=4224
Child2: pid=4226 pgrp=4224
yunmin@yunmin:~/ch3_2$ ps
    PID TTY          TIME CMD
   3605 pts/0    00:00:00 bash
   4225 pts/0    00:00:02 killproc1
   4226 pts/0    00:00:02 killproc1
   4238 pts/0    00:00:00 ps
yunmin@yunmin:~/ch3_2$ kill -9 -4224
yunmin@yunmin:~/ch3_2$ ps
    PID TTY          TIME CMD
   3605 pts/0    00:00:00 bash
   4291 pts/0    00:00:00 ps
```

Signals from Keyboard

- Typing `ctrl-c` (`ctrl-z`) sends a `SIGTERM` (`SIGTSTP`) to every job in the foreground process group.
  - `SIGTERM`: default action is to terminate each process
  - `SIGTSTP`: default action is to stop (suspend) each process



Signals from Keyboard: Example

- Example of ctrl-c and ctrl-z

```
int main()
{
    if (fork() == 0) {
        printf("Child: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
    } else {
        printf("Parent: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
    }
    while(1);
    return 0;
}
```

```
yunmin@yunmin:~/ch3_2$ ./killproc2
Parent: pid=5339 pgrp=5339
Child: pid=5340 pgrp=5339
^Z      ←—— Typed ctrl-z
[1]+  Stopped                 ./killproc2
yunmin@yunmin:~/ch3_2$ ps -o pid,tty,time,stat,cmd
    PID TT           TIME STAT CMD
   3605 pts/0    00:00:00 Ss   /bin/bash --init-file /home/y
   5339 pts/0    00:00:03 T    ./killproc2
   5340 pts/0    00:00:03 T    ./killproc2
   5390 pts/0    00:00:00 R+   ps -o pid,tty,time,stat,cmd
yunmin@yunmin:~/ch3_2$ fg
./killproc2
^C      ←—— Typed ctrl-c
yunmin@yunmin:~/ch3_2$ ps -o pid,tty,time,stat,cmd
    PID TT           TIME STAT CMD
   3605 pts/0    00:00:00 Ss   /bin/bash --init-file /home/y
   5455 pts/0    00:00:00 R+   ps -o pid,tty,time,stat,cmd
```

`fg`: By default, this brings the last background job to the foreground.

`fg %job_number`: Brings a specific job (by job number) to the foreground.

https://man7.org/linux/man-pages/man1/ps.1.html

kill()

- `kill()`: Send signal to a process

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

  - The `kill()` system call can be used to send any signal to any process group or process.
  - *pid*>0: *sig* is sent to the process with specified by *pid*
  - *pid*=0: *sig* is sent to every process in the process group of the calling process
  - *pid*=-1: *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (init)
  - *pid*<-1: *sig* is sent to every process in the process group whose ID is *-pid*.
  - Return value
    - Success: zero is returned
    - Error: -1 is returned

kill(): Example

killproc3.c

```
void main()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
yunmin@yunmin:~/ch3_2$ ./killproc3
Killing process 5703
Killing process 5704
Killing process 5705
Killing process 5706
Killing process 5707
Child 5705 terminated abnormally
Child 5704 terminated abnormally
Child 5706 terminated abnormally
Child 5707 terminated abnormally
Child 5703 terminated abnormally
```

signal()

- `signal()`: Define the action associated with a signal

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

  - Sets the disposition of the signal *`signum`* to *`handler`*
  - *`signum`*: the name of signal
  - *`handler`*: could be one among three possible reaction to the signal *`signum`*
    - `SIG_IGN`: ignore the signals
    - `SIG_DFL`: use default action
    - `Address of signal handler`: use a programmer-defined function

    → Installing the handler

  - Return value
    - Success: the previous value of the signal handler
    - Error: SIG_ERR

https://man7.org/linux/man-pages/man7/signal.7.html

signal(): Example

signal1.c

```
// SIGINT handler
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}

void main()
{
    pid_t pid[N];
    int i;
    int child_status;

    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0)
            while(1);   /* Child: Infinite Loop */
    }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
```

```
/* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
            wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
yunmin@yunmin:~/ch3_2$ ./signal1
Killing process 2597
Killing process 2598
Killing process 2599
Killing process 2600
Process 2600 received signal 2
Process 2599 received signal 2
Killing process 2601
Process 2601 received signal 2
Child 2599 terminated with exit status 0
Child 2600 terminated with exit status 0
Child 2601 terminated with exit status 0
Process 2598 received signal 2
Child 2598 terminated with exit status 0
Process 2597 received signal 2
Child 2597 terminated with exit status 0
```

Signal Handler Funkiness

signal2.c

```
int ccount = 0;

// SIGCHLD handler that reaps one terminated child
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n", sig, pid);
}

// Signal funkiness: Pending signals are not queued
void main()
{
    pid_t pid[N];
    int i;
    ccount = N;

    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(0);   /* Child: Exit */
        }
    while (ccount > 0)
        pause();   /* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending even if multiple processes have sent this signal

```
yunmin@peace:~/ch10$ ./signal2
Received signal 17 from process 34372
Received signal 17 from process 34373
Received signal 17 from process 34374
Received signal 17 from process 34375
```

Program does not exit

Living With Nonqueuing Signals

signal3.c

```
int ccount = 0;

// SIGCHLD handler that reaps all terminated children
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

// Using a handler that reaps multiple children
void main()
{
    pid_t pid[N];
    int i;
    ccount = N;

    signal(SIGCHLD, child_handler2);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(0); /* Child exits */
        }
    while (ccount > 0);  /* Parent spins */
}
```

- Must check for all terminated jobs
  - Typically loop with wait

```
yunmin@yunmin:~/ch3_2$ ./signal3
Received signal 17 from process 2875
Received signal 17 from process 2876
Received signal 17 from process 2877
Received signal 17 from process 2879
Received signal 17 from process 2878
```

Example for SIGINT

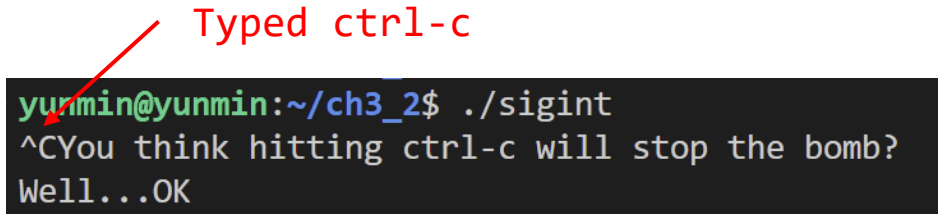
- A program that reacts to externally generated events (ctrl-c)

sigint.c

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

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

void main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
    }
}
```



Example for SIGALRM

- A program that reacts to internally generated events

sigalrm.c

```
int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);

    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

void main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */

    while (1) {
        /* handler returns here */
    }
}
```

```
yunmin@yunmin:~/ch3_2$ ./sigalrm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

sigaction()

- `sigaction()`: Examine and change a signal action

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

  - Used to change the action taken by a process on receipt of a specific signal.
  - *`signum`*: the name of signal
  - *`act`*: the new action
  - *`oldact`*: the previous actions
  - Return value
    - Success: 0
    - Error: -1

  - struct sigaction

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int      sa_flags;
    void     (*sa_restorer)(void);
};
```

    - sa_handler: the address of a function to be called when the signal occurs
    - sa_mask: specifies a set of signals that will be blocked when the signal handler is called
    - sa_flags: options. Basically 0

sigaction(): Example

sigaction.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// Signal handler for timeout
void timeout(int sig)
{
    if (sig == SIGALRM)
        puts("Time out!");

    // generate SIGALRM after 2 seconds
    alarm(2);
}
```

```c
int main(int argc, char *argv[])
{
    int i;
    struct sigaction act;
    act.sa_handler = timeout;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    // specify signal type and signal handler
    sigaction(SIGALRM, &act, 0);

    alarm(2);

    for (i = 0; i < 3; i++)
    {
        puts("wait...");
        sleep(10);
    }
    return 0;
}
```

```
yunmin@yunmin:~/ch3_2$ ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

Agenda

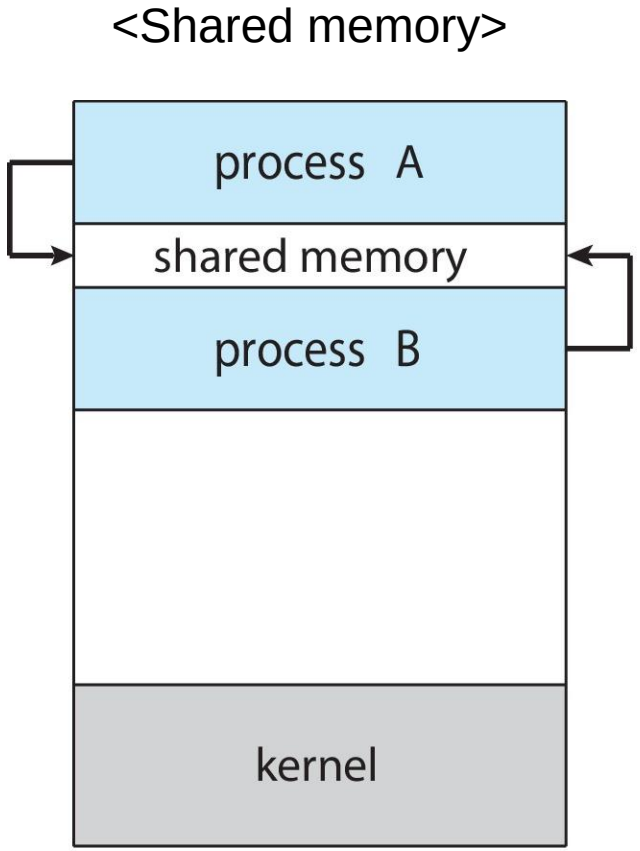
- Process Concept
- Process State and Scheduling
- Process API
- Signal
- Inter Process Communication

Inter Process 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
- Cooperating processes need **inter process communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Shared Memory

- **Shared memory**: an area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
  - Synchronization is discussed in great details in Chapter 5.



Shared Memory

- POSIX shared memory
  - Communication through **memory-mapped file**.
  - Process creates or open shared memory segment

    ```
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ```

    - O_CREAT: create if it does not yet exist / O_RDWR: open for reading and writing
  - Set the size of the object

    ```
    ftruncate(shm_fd, 4096);
    ```
  - Map shared memory segment to process address space

    ```
    ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    ```
  - Now the process could write to the shared memory

    ```
    sprintf(ptr, "Writing to shared memory");
    ```

Shared Memory: Example - Producer

shm_pro.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>

int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    const char *message0 = "Studying ";
    const char *message1 = "Operating Systems ";
    const char *message2 = "Is Fun!\n";

    int shm_fd;
    void *ptr;

    // create the shared memory segment
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    // configure the size of the shared memory segment
    ftruncate(shm_fd,SIZE);

    // now map the shared memory segment
    // in the address space of the process
    ptr = (char*) mmap(0,SIZE, PROT_READ | PROT_WRITE,
                       MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        return -1;
    }

    // Now write to the shared memory region.
    // Note we must increment the value of ptr
    // after each write.
    sprintf(ptr,"%s",message0);
    ptr += strlen(message0);
    sprintf(ptr,"%s",message1);
    ptr += strlen(message1);
    sprintf(ptr,"%s",message2);
    ptr += strlen(message2);

    return 0;
}
```

Shared Memory: Example - Consumer

shm_con.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    const char *name = "OS";
    const int SIZE = 4096;

    int shm_fd;
    void *ptr;
    int i;

    // open the shared memory segment
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1) {
        printf("shared memory failed\n");
        exit(-1);
    }

    // now map the shared memory segment
    // in the address space of the process
    ptr = (char*) mmap(0,SIZE, PROT_READ, MAP_SHARED,
                       shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        exit(-1);
    }

    // now read from the shared memory region
    printf("%s\n",(char *)ptr);

    // remove the shared memory segment
    if (shm_unlink(name) == -1) {
        printf("Error removing %s\n",name);
        exit(-1);
    }

    return 0;
}
```

```
yunmin@yunmin:~/ch3_2$ gcc shm_pro.c -o shm_pro -lrt
yunmin@yunmin:~/ch3_2$ gcc shm_con.c -o shm_con -lrt
yunmin@yunmin:~/ch3_2$ ./shm_pro
yunmin@yunmin:~/ch3_2$ ./shm_con
Studying Operating Systems Is Fun!
yunmin@yunmin:~/ch3_2$ ./shm_con
shared memory failed
```

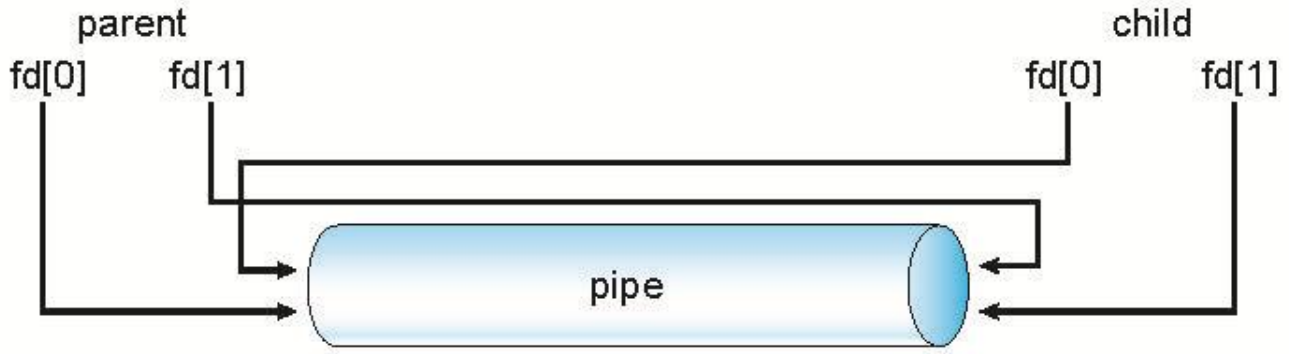
Shared Memory: Reference

Search the following topics from Internet and study them

- POSIX shared memory (requires compilation option '-lrt')
  - shm_open(): http://man7.org/linux/man-pages/man3/shm_open.3.html
  - shm_unlink(): http://man7.org/linux/man-pages/man3/shm_unlink.3p.html
  - ftruncate(): http://man7.org/linux/man-pages/man3/ftruncate.3p.html
  - mmap(): http://man7.org/linux/man-pages/man2/mmap.2.html
  - munmap(): http://man7.org/linux/man-pages/man3/munmap.3p.html

Pipes

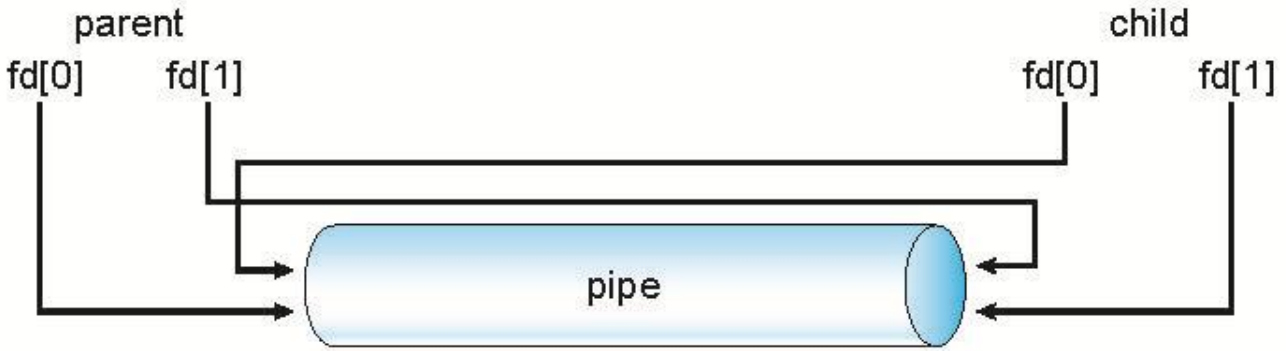
- Pipes acts as a conduit allowing two processes to communicate



- Ordinary pipes
  - Unidirectional communication between parent and child
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created
- Named pipes
  - Can be bidirectional and no parent-child relationship is required

Ordinary Pipes

- Ordinary pipes allow unidirectional communication in standard producer-consumer style
  - Producer writes to one end (the write-end of the pipe)
  - Consumer reads from the other end (the read-end of the pipe)
  - Require parent-child relationship between communicating processes



- Windows calls these anonymous pipes

Ordinary Pipes: Example

pipe.c

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END    0
#define WRITE_END   1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    pid_t pid;
    int fd[2];

    /* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }

    /* now fork a child process */
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        return 1;
    }
```

```c
    if (pid > 0) {  /* parent process */
        /* close the unused end of the pipe */
        close(fd[READ_END]);

        /* write to the pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

        /* close the write end of the pipe */
        close(fd[WRITE_END]);
    }
    else { /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE_END]);

        /* read from the pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("child read %s\n",read_msg);

        /* close the write end of the pipe */
        close(fd[READ_END]);
    }

    return 0;
}
```

```
yunmin@yunmin:~/ch3_2$ gcc pipe.c -o pipe
yunmin@yunmin:~/ch3_2$ ./pipe
child read Greetings
```

Named Pipes

- Named pipes are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication
  - Provided on both UNIX and Windows systems
  - Named pipes are referred to as FIFOs in UNIX systems

Named Pipes: Example

npipe1.c

```c
// This side writes first, then reads
int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";           // FIFO file path

    // Creating the named file(FIFO)
    mkfifo(myfifo, 0666);
    char arr1[80], arr2[80];
    while (1) {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);

        // Take an input arr2 from user. 80 is maximum length
        fgets(arr2, 80, stdin);

        // Write the input arr2 on FIFO and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);

        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);

        // Read from FIFO
        read(fd, arr1, sizeof(arr1));

        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

npipe2.c

```c
// This side reads first, then reads
int main() {
    int fd1;
    char *myfifo = "/tmp/myfifo";          // FIFO file path

    // Creating the named file(FIFO)
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1) {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);

        // Now open in write mode and write string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

```
yunmin@yunmin:~/ch3_2$ ./npipe1
Hello!
User2: Handong!

Operating Systems
User2: University
```

```
yunmin@yunmin:~/ch3_2$ ./npipe2
User1: Hello!

Handong!
User1: Operating Systems

University
```

Message Passing: Reference

Search the following topics from Internet and study them

- POSIX message passing
  - msgget(): create a message queue
    - https://man7.org/linux/man-pages/man2/msgget.2.html
  - msgsnd(): send a message to a message queue
    - https://man7.org/linux/man-pages/man3/msgsnd.3p.html
  - msgrcv(): receive a message from a message queue
    - https://man7.org/linux/man-pages/man3/msgrcv.3p.html
  - msgctl(): control/deallocate message queue
    - https://man7.org/linux/man-pages/man2/msgctl.2.html