Processes & Inter-Process Communication

Z. Cihan TAYŞİ

Outline

- Process management
  - creation, differentiation, termination
- Inter-process communcation (IPC)
  - Unix-based
  - Posix
- Threads
- Thread synchronization

Section Outline

- Process definition
- Process handling in Unix
- Process creation
- Process differentiation
- Process termination
- Process synchronization

What is a Process

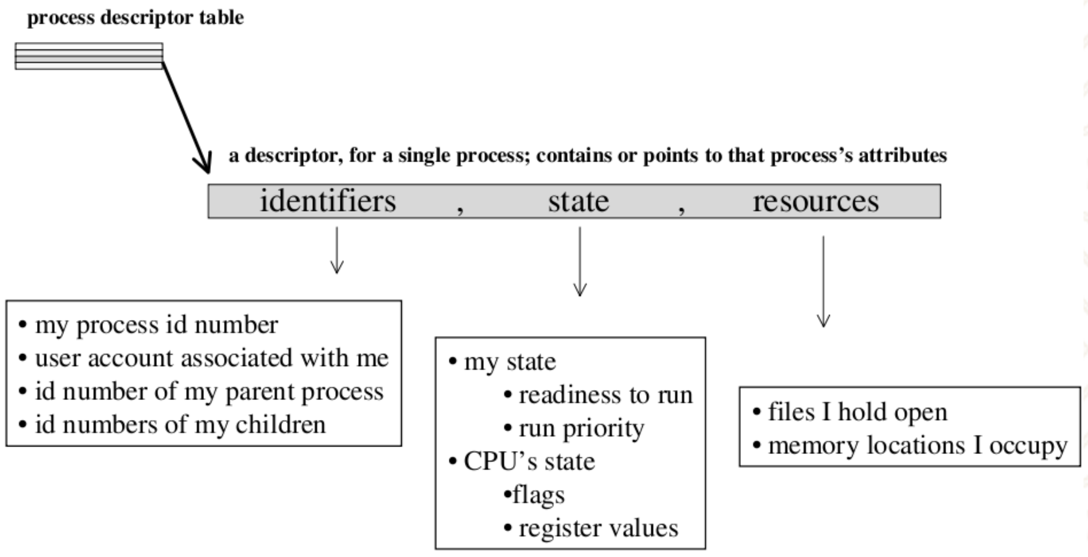
- **Definition ;**
  - A process is an instance of a running program.
- Not the same as "**program**" or "**processor**"
  - A program is a set of instructions and initialized data in a file, usually found on a disk.
- A **process** is an instance of that **program** while it is running, along with the state of all the CPU registers and the values of data in memory.

Process Handling

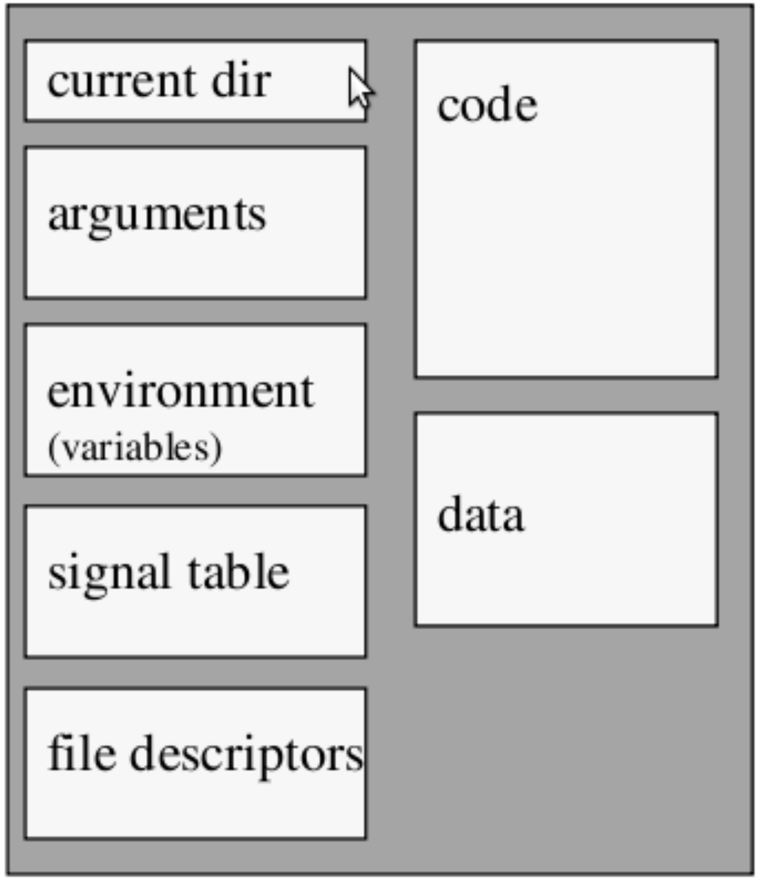
- Constituents of a process
  - Its code
  - Data
    - its own
    - OS’s data used by/for process
  - Various attributes OS needs to manage it

Process Handling

- OS keeps track of all processes
  - Process table/array/list
  - Elements are process descriptors (aka control blocks)
  - Descriptors reference code & data

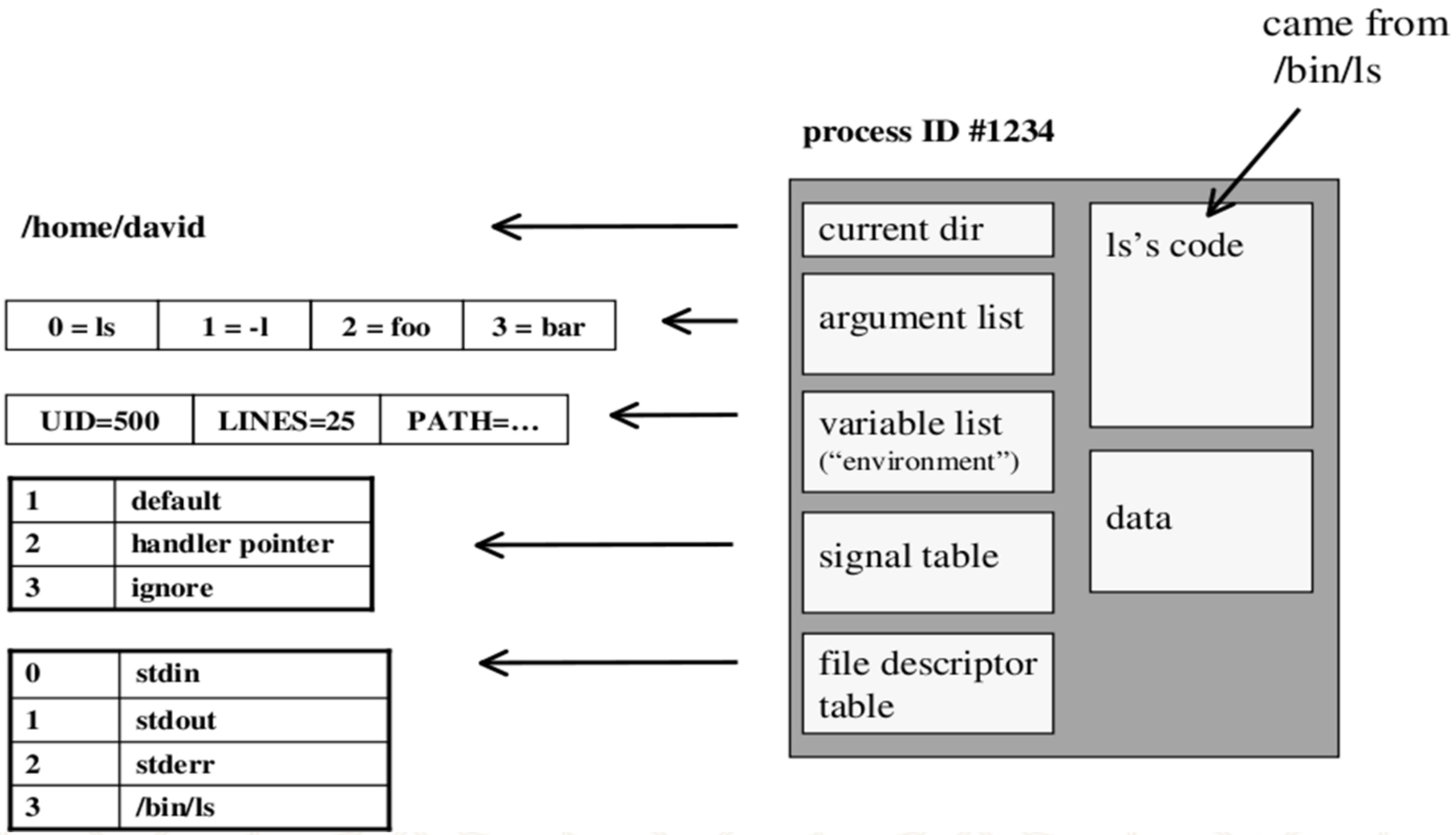


Single process in Unix (consolidated view)



- Some important properties
  - code
  - data
  - current directory
  - argument list
  - environment list
  - responses to signals
  - list of open files

ls -l foo bar

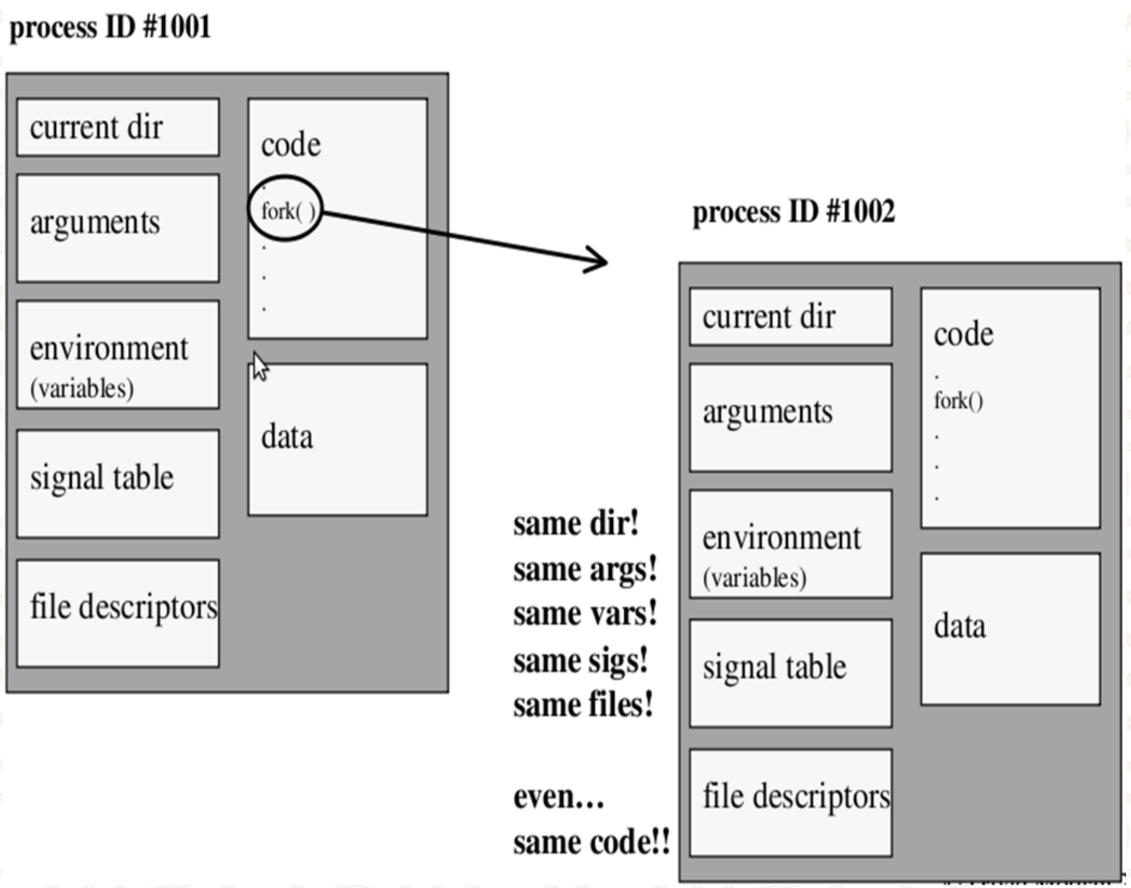


came from
/bin/ls
process ID #1234
/home/david
current dir
ls's code
0 = ls
1 = -l
2 = foo
3 = bar
argument list
UID=500
LINES=25
PATH=...
variable list
("environment")
1 default
2 handler pointer
3 ignore
data
signal table
0 stdin
1 stdout
2 stderr
3 /bin/ls
file descriptor
table

Process Creation

- OS perspective
  - find empty slot in process table
  - write a process descriptor and
  - put it there
  - read in program code from disk
- User perspective
  - System calls
    - **fork(), exec()**

fork() system call



A Simple fork() Example

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

int main ( void )  {

        printf("Message before fork\n");

        fork();

        printf("Message after fork\n");

        return 0;
}
```

- a simple fork example
- Message after fork is printed **twice** !!

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork1
Message before fork
Message after fork
lucid@ubuntu:~/Downloads$ Message after fork

lucid@ubuntu:~/Downloads$ 
```

- **fork1.c**

Self Identification

- for the parent process fork returns **child’s pid**
- for the child process fork returns **0**

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

int main ( void )  {

        int forkResult;

        printf("process id : %i\n",getpid());
        forkResult = fork();
        printf("process id : %i - result : %d\n",
                getpid(), forkResult);

        );
```

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork2
process id : 2682
process id : 2682 - result : 2683
lucid@ubuntu:~/Downloads$ process id : 2683 - result : 0
```

- **fork2.c**

Process Differentiation

- **identical?**
    - not what we had in mind!
- more useful if child does different stuff
- can we give it **different** behaviour?
    - in the form of source code
    - in the form of an existing binary executable
        - exec() family of functions

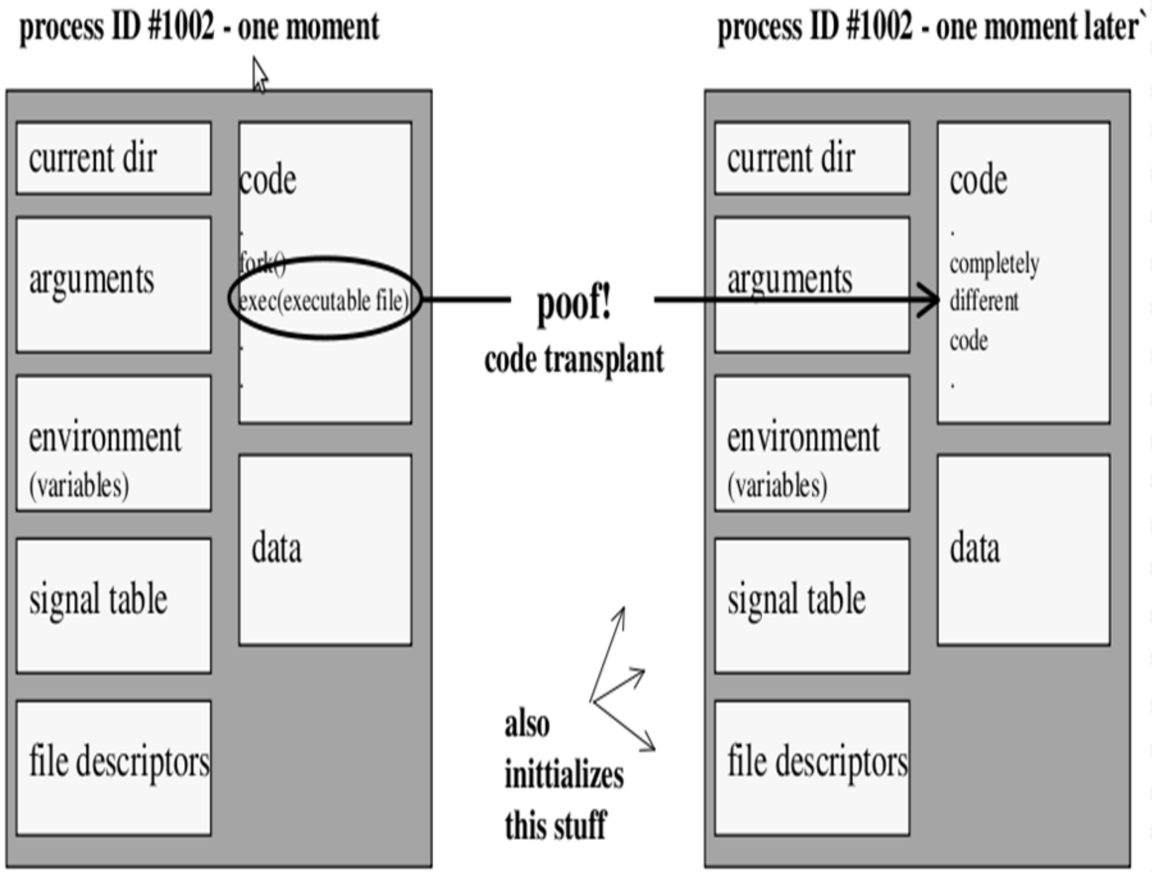
Process Differentiation
by source code

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

int main ( void )  {

        printf("(%i) Parent does something...\n", getpid());

        if(fork())  {  // Parent
                printf("(%i) Parent do completely different stuff\n",getpid());
        } else {        // Child
                printf("(%i) Child can do some stuff\n",getpid());
        }


        exit(0);
}
```

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork3
(2767) Parent does something...
(2767) Parent do completely different stuff
lucid@ubuntu:~/Downloads$ (2768) Child can do some stuff
```

- **fork3.c**

Process Differentiation
by exec() function



Process Differentiation
by exec() function

- **exec() family of functions**
  - int **execl** (const char *pathname, const char *arg0, …);
  - int **execv** (const char *pathname, char *const argv[]);
  - int **execle** (const char *pathname, const char *arg0, ..., 0, char *const envp[]);
  - int **execlp** (const char *filename, const char *arg0, …);
  - int **execvp** (const char *filename, char *const argv[]);
  - int **execve** (const char *pathname, char *const argv[],   char *const envp[]);

A Simple exec() Example

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

int main ( void )  {

        printf("Parent does stuff and then calls fork...\n");

        if(fork())  { // Parent
                printf("... parent do something completely different\n");
        } else  {       // Child
                printf("Child runs an executable...\n");
                execl("/bin/ls","/bin/ls","-l","/etc/apache2/conf.d/",NULL);
        }

        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Exec
Parent does stuff and then calls fork...
... parent do something completely different
lucid@ubuntu:~/Downloads$ Child runs an executable...
/bin/ls: cannot access /etc/apache2/conf.d/: No such file or directory
```

- **exec.c**

A Shell Example

```c
int main(void)
{
    char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
    int background;             /* equals 1 if a command is followed by '&' */
    char *args[MAX_LINE/2+1];/* command line (of 80) has max of 40 arguments */

    while (1){              /* Program terminates normally inside setup */
        background = 0;
        printf("COMMAND->");
        fflush(0);
        setup(inputBuffer, args, &background);       /* get next command */

        /* the steps are:
         (1) fork a child process using fork()
         (2) the child process will invoke execvp()
         (3) if background == 0, the parent will wait,
                otherwise returns to the setup() function. */
    }
}
```

- Shell is running
  - You type “ls” and Enter
  - Shell is parent, spawns ls as child

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Shell
COMMAND->
COMMAND->
COMMAND->
COMMAND->
```

- **shell.c**

Process Termination

- void **exit** (int status);
  - exits a process
  - normally return with status 0
- int **atexit** (void (*function)(void));
  - registers function to be executed on exit
- int **wait** (int *child_status)
  - suspends current process until one of its children terminates

exit() vs return

- **return**
  - is an instruction of the language that returns from a function call.
- **exit**
  - is a system call (not a language statement) that terminates the current process.

atexit() example

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

void parentCleaner ( void );

int main ( void )  {

        if(fork())  { // parent process
                atexit(parentCleaner);
                printf("this is parent %i\n",getpid());
        } else  {       // child process
                printf("this is child %i\n",getpid());
        }
        exit(0);
}

void parentCleaner ( void )  {

        printf("cleaning up parent...\n");

}
```

- **atexit.c**

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Exit1
this is parent 3262
cleaning up parent...
lucid@ubuntu:~/Downloads$ this is child 3263
```

- registers a function to clean up resource at process termination

Zombie Process

- When process terminates, still consumes system resources
  - Various tables maintained by OS
  - Called a zombie; living corpse, half alive, half dead
- **Reaping**
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent does not reap ?
  - if any parent terminates without reaping a child, then child will be reaped by **"init"** process
  - so, only need explicit reaping in long-running processes

Zombie example
non-terminating parent

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

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                while(1);
        } else  {     // Child
                printf("Terminating child, pid : %i\n", getpid());
                exit(0);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3380  2182 71 03:42 pts/0    00:00:21 ./Zombie1
lucid     3381  3380  0 03:42 pts/0    00:00:00 [Zombie1] <defunct>
lucid     3402  3382  0 03:43 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- **zombie1.c**

Zombie example
non-terminating child

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

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                exit(0);
        } else  {    // Child
                printf("Terminating child, pid : %i\n", getpid());
                while(1);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid      3467      1 77 03:45 pts/0    00:00:29 ./Zombie2
lucid      3473   3382  0 03:46 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- **zombie2.c**

Synchronizing with child

- int wait(int *child_status)
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
  - If the child has already terminated, then wait returns its pid immediately
  - If child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated

wait() Example

```
lucid@ubuntu:~/Downloads$ ./Wait1
Child 3630 terminated with exit status 100
Child 3631 terminated with exit status 101
Child 3633 terminated with exit status 103
Child 3634 terminated with exit status 104
Child 3632 terminated with exit status 102
lucid@ubuntu:~/Downloads$
```

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

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);            // create & exit child
                }
        }

        for (i = 0; i < numOfChilds; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```

- **wait1.c**

wait() Example

```
lucid@ubuntu:~/Downloads$ ./Wait2
Child 3656 terminated with exit status 101
Child 3657 terminated with exit status 102
Child 3658 terminated with exit status 103
Child 3659 terminated with exit status 104
Child 3655 terminated with exit status 100
Child -1 terminated with exit status 100
lucid@ubuntu:~/Downloads$
```

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

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);              // create & exit child
                }
        }

        for (i = 0; i < numOfChilds+1; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```

- **wait2.c**

References

- **man pages**
- http://www.cs.princeton.edu/courses/archive/fall01/cs217/slides/process.pdf
- http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f08/www/lectures/lecture-11.pdf
- http://csapp.cs.cmu.edu
- http://homepage.smc.edu/morgan_david/linux/a12-processes.pdf

Section Outline

- What is IPC
- IPC standards
- Posix IPC Methods
  - Pipes
  - Fifos
  - Signals
  - Semaphores
  - Message queues
  - Shared memory

What is IPC

- **Inter-process communication (IPC)** is a set of methods for the exchange of data among multiple threads in one or more processes.
  - Processes may be running on one or more computers connected by a network.
  - IPC methods are divided into methods for **message passing**, **synchronization**, **shared memory**, and **remote procedure calls** (RPC).

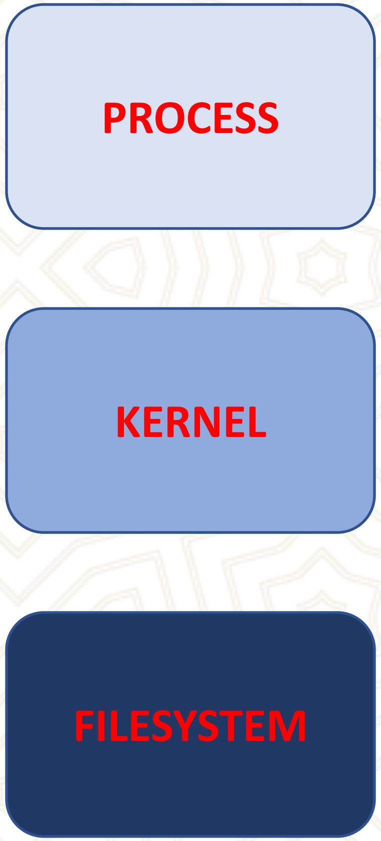
IPC Methods

- Unix
- System V
- POSIX
- Others
  - Sockets
  - Dbus
  - So on…

POSIX IPC

- Pipe
- FIFO
- **Signals**
- **Semaphores**
- Message Queues
- Shared Memory

Persistence of IPC Objects

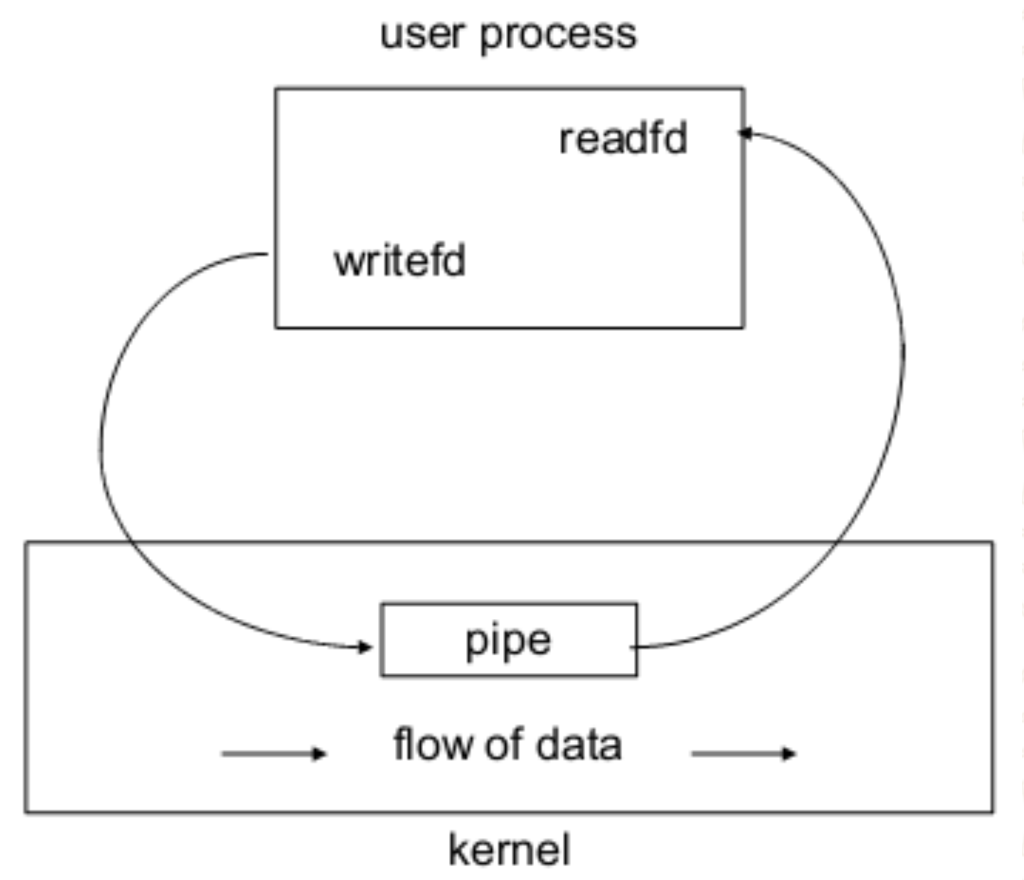


- process-persistent IPC:
  - exists until last process with
  - IPC object closes the object
- kernel-persistent IPC
  - exists until kernel reboots or
  - IPC object is explicitly deleted
- file-system-persistent IPC
  - exists until IPC object is
  - explicitly deleted

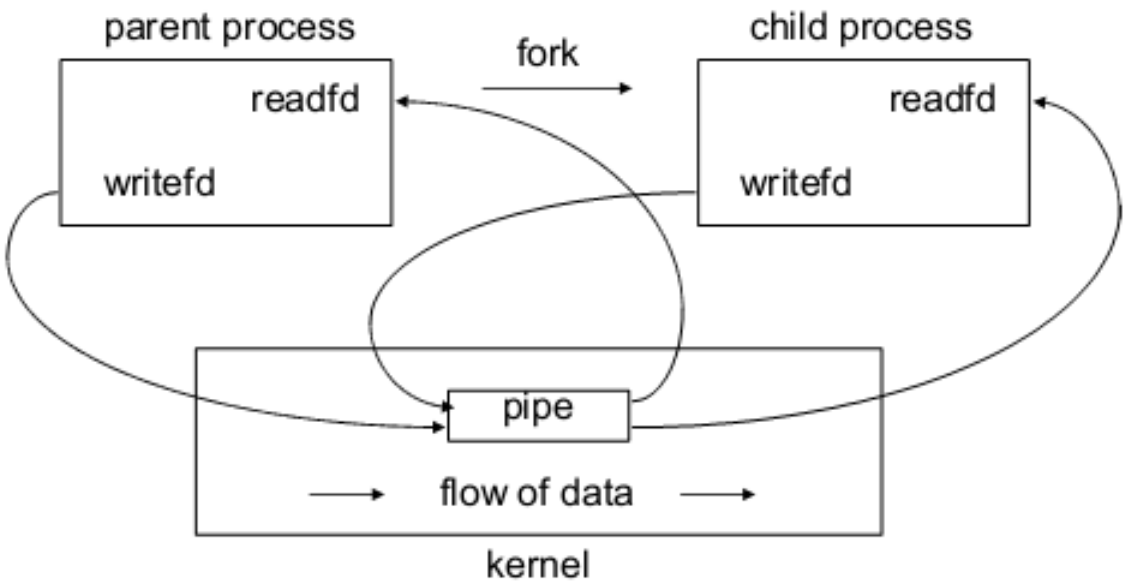
Pipes

- A pipe provides a one-way flow of data
  - example: who | sort| lpr
- The difference between a file and a pipe:
  - pipe is a data structure in the kernel.
- A pipe is created by using the pipe system call
  - int **pipe** (int* filedes);
  - Two file descriptors are returned
- **filedes[0]** is open for reading
- **filedes[1]** is open for writing
- Typical size is **512 bytes** (Minimum limit defined by POSIX)

Pipe (Single Process)

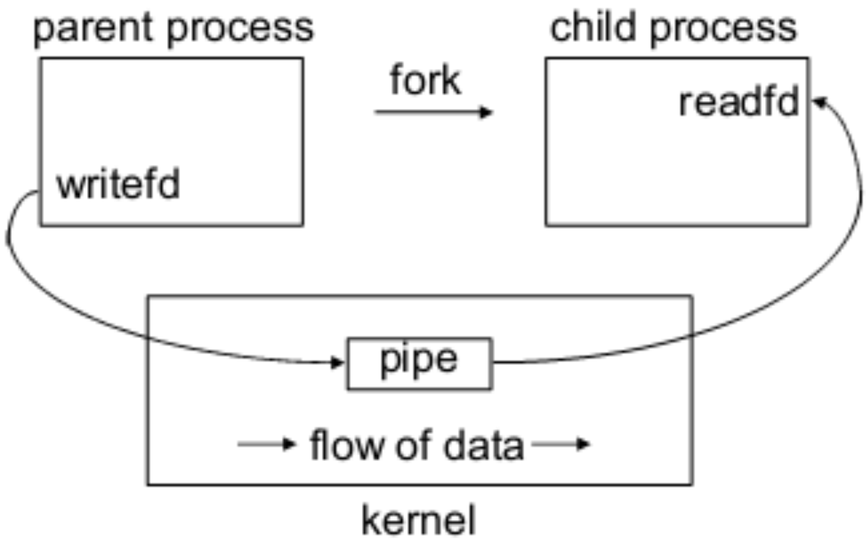


Pipe (Two Process)



Just after fork

Pipe
(Two Process)



- Parent opens file, child reads file
  - parent closes **read** end of pipe
  - child closes **write** end of pipe

A simple pipe example

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

        int pipe1[2];
        pid_t childpid;

        pipe(pipe1);

        if((childpid=fork())==0)  {// Child
                close(pipe1[1]);
                server(pipe1[0]);
                exit(0);
        }

        close(pipe1[0]);
        client(pipe1[1]);

        waitpid(childpid, NULL,0); // wait for child to terminate
        exit(0);
}
```

```
void client (int writefd ) {

        size_t len;
        char buff[MAX_LINE];

        fgets(buff, MAX_LINE, stdin);
        len = strlen(buff);
        if(buff[len-1]=='\n')
                len--;
        write(writefd, buff, len);
}
```

```
lucid@ubuntu:~/Downloads$ ./Pipe
hello.txt
This
is
the
content
of
Hello.txt
file
lucid@ubuntu:~/Downloads$
```

- **pipe.c**

More on Pipes

- FILE ***popen** (const char * command, const char *type)
  - Type is r, the calling process reads the standart output of the command,
  - Type is w, the calling process writes to the standart input of the command
  - Return file * if OK, NULL on error
- int **pclose** (FILE *stream);
  - Closes a standard I/O stream that was created by popen

FIFOs

- Pipes have no names, they can only be used between processes that have a parent process in common.
- FIFO stands for first-in, first-out
- Similar to a pipe, it is a one-way (half duplex) flow of data
- A FIFO has a pathname associated with it, allowing unrelated processes to access a single FIFO
- FIFOs are also called ***named pipes***

FIFOs

```
#include <sys/types.h>
#include <sys/stat.h>
```

int **mkfifo** (const char *pathname, mode_t mode)

returns 0 if OK, -1 on error

FIFO example

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

        int readfd, writefd;

        if((mkfifo(FIFO1, FIFO_MODE )<0)&&(errno != EEXIST))  {
                printf("can not open %s\n",FIFO1);
                exit(-1);
        }

        if((mkfifo(FIFO2, FIFO_MODE )<0)&&(errno!= EEXIST))  {
                printf("can not open %s\n",FIFO2);
                exit(-1);
        }

        readfd  = open(FIFO1, O_RDONLY);
        writefd = open(FIFO2, O_WRONLY);

        server(readfd, writefd);

        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Client
enter a file name
hello.txt

sending file name to server

This
is
the
content
of
Hello.txt
file
lucid@ubuntu:~/Downloads$
```

```
lucid@ubuntu:~/Downloads$ ./Server
received file name (hello.txt)
sending contents of the file back to client...
lucid@ubuntu:~/Downloads$
```

- **client.c**
- **server.c**

Signals

- Definition
- Signal Types
- Generating Signals
- Responding to a Signal
- POSIX Signal Functions
- Signals & System Calls

Definition

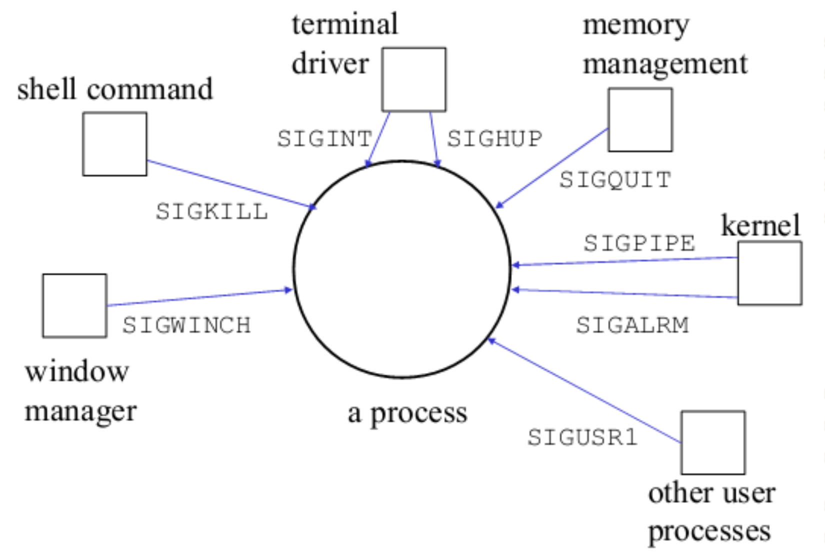
- A signal is an **asynchronous** event which is delivered to a process
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types ctrl-C, or the modem hangs

Common use of Signals

- Ignore a Signal
- Clean up and Terminate
- Dynamic Reconfiguration
- Report Status
- Turn Debugging on/off
- Restore Previous Handler
- Signals & System Calls

Signal Sources

Name	Description	Default Action
`SIGINT`	Interrupt character typed	terminate process
`SIGQUIT`	Quit character typed (`^\`)	create core image
`SIGKILL`	`kill -9`	terminate process
`SIGSEGV`	Invalid memory reference	create core image
`SIGPIPE`	Write on pipe but no reader	terminate process
`SIGALRM`	`alarm()` clock 'rings'	terminate process
`SIGUSR1`	user-defined signal type	terminate process
`SIGUSR2`	user-defined signal type	terminate process



- After receiving a signal, the process may
  - terminate abnormally
  - ignore
  - stop
  - continue

Generating a Signal

- Use the Unix command
  - $> kill -KILL 4481
  - Sends a SIGKILL signal to pid 4481
- ps -l
  - To make sure process died

```
#include <sys/types.h>
#include <signal.h>
```

int **kill** (pid_t pid, int sig);

Sends a signal to a process or a group of processes

Return 0 if ok, -1 on error

PID Options

- If pid is positive, then signal sig is sent to the process with the ID specified by pid.
- If pid equals 0, then sig is sent to every process in the process group of the calling process.
- If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.
- If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.
- If sig is 0, then no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID

Responding to a Signal

- A process can;
  - Ignore/discard the signal (not possible for SIGKILL & SIGSTOP)
  - Execute a signal handler function, and then possibly resume execution or terminate
  - Carry out default action for that signal
- The choice is called the process' **signal disposition**

POSIX Signal System

- The POSIX signal system, uses signal sets, to deal with pending signals that might otherwise be missed while a signal is being processed
- The signal set stores collection of signal types
- Sets are used by signal functions to define which signal types are to be processed
- POSIX contains several functions for creating, changing and examining signal sets

POSIX Functions

```
#include <signal.h>
int sigemptyset ( sigset_t *set );
int sigfillset ( sigset_t *set );
int sigismember ( const sigset_t *set,int signo );
int sigaddset ( sigset_t *set, int signo );
int sigdelset ( sigset_t *set, int signo );
int sigprocmask ( int how, const sigset_t *set,
sigset_t *oldset);
```

sigprocmask()

- A process uses a signal set to create a mask which defines the signals it is blocking from delivery
  - Good for critical sections where you want to block certain signals.
- How meanings
  - SIG_BLOCK set signals are added to mask
  - SIG_UNBLOCK set signals are removed from mask
  - SIG_SETMASK set becomes new mask

A Critical Code Region

sigset_t newmask, oldmask;

sigemptyset(&newmask);

sigaddset(&newmask, SIGINT);

/ block SIGINT; save old mask */*

sigprocmask(SIG_BLOCK, &newmask, &oldmask);

/* critical region of code */

/ reset mask which unblocks SIGINT */*

sigprocmask(SIG_SETMASK, &oldmask, NULL);

sigaction()

- Supercedes (more powerful than) signal()
  - can be used to code a non-resetting signal()

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

sigaction Structure

```
struct sigaction   {
    void (*sa_handler)( int );  //action to be taken or  SIG_IGN, SIG_DFL
    sigset_t sa_mask;             //additional signal to be blocked
    int sa_flags;                 // modifies action of the signal
    void (*sa_sigaction)( int, siginfo_t *, void * );
}
```

sa_flag

SIG_DFL reset handler to default upon return

SA_SIGINFO denotes **extra information** is passed to handler (.i.e. specifies the use of the “second” handler in the structure.

sigaction() Behavior

- A signo signal causes the sa_handlersignal handler to be called.
- While sa_handler executes, the signals in sa_mask are blocked. Any more signo signals are also blocked.
- sa_handler remains installed until it is changed by another sigaction()call. **No reset problem**

A Simple Example !!!

```
lucid@ubuntu:~/Downloads$ ./Signal
Hello world
Hello world
Hello world
Hello world
Hello world
^Creceived SIGINT...
Hello world
Hello world
Hello world
^Creceived SIGINT...
Hello world
Hello world
Hello world
^Z
[1]+  Stopped                 ./Signal
lucid@ubuntu:~/Downloads$
```

- **signal1.c**

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

void ouch( int );

int main (void)  {

        struct sigaction act;

        act.sa_handler = ouch;

        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        sigaction(SIGINT, &act, 0);

        while(1)  {
                printf("Hello world\n");
                sleep(1);
        }
        exit (0);
}

void ouch( int sigNo ) {

        printf("received SIGINT...\n");
}
```

Ignoring Signals

- Other than SIGKILL and SIGSTOP, signals can be ignored.
- Instead of in the previous program:

act.sa_handler = catchint /* or whatever */

We use :

act.sa_handler = SIG_IGN;

The ^C key will be ignored

Restoring Previous Action

- The third parameter to sigaction, oact, can be used:

```
/* save old action */
sigaction( SIGTERM, NULL, &oact);
/* set new action */
act.sa_handler = SIG_IGN;
sigaction( SIGTERM, &act, NULL );
/* restore old action */
sigaction( SIGTERM, &oact, NULL );
```

Signals & System Calls

- Interrupted System Calls
- System Calls Inside Handlers

Interrupted System Calls

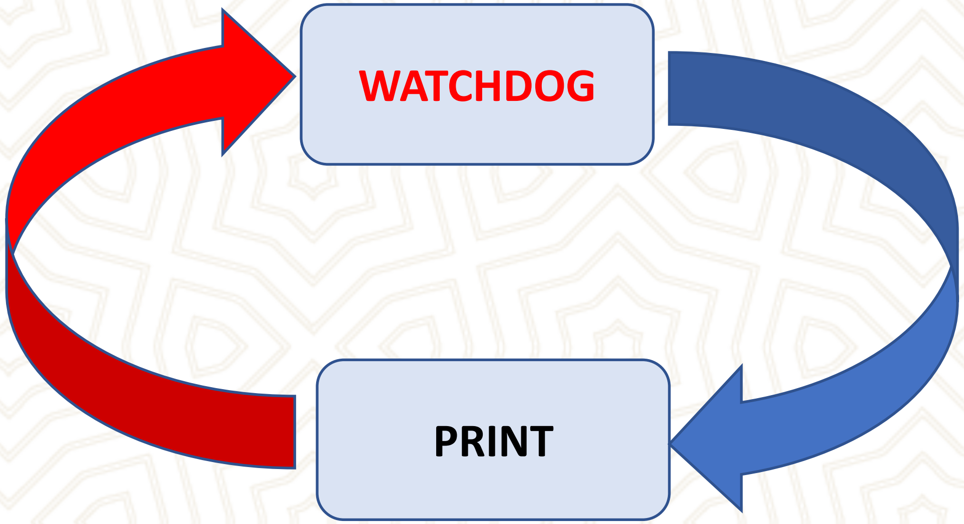
- When a system call is interrupted by a signal, a signal handler is called, returns, and then what ?
- Slow system function calls do not resume, Instead they return an error and errno is assigned EINTR.
  - Example !!!
- Some UNIXs resume non-slow system functions after the handler has finished.
- Some UNIXs only call the handler after non-slow system function call has finished

System Calls Inside Handlers

- If a system function is called inside a signal handler then it may interact with an interrupted call to the same function in the main code.
  - e.g. malloc()
- Not a problem if the function is re-entrant
  - A process can contain multiple calls to these functions at the same time. e.g. read(), write()
- A function may be non-reentrant for a number of reasons
  - It uses a static data structure
  - It manipulates the heap. e.g. malloc(), free()
  - It uses standart I/O library. e.g. printf()

A Complex Signal Example

- A simple wathdog implementation
  - watchdog.c
    - watchdog for prınt process
  - print.c
    - a simple program that prints a message to stdout
  - hbeat.c
    - sıgnal handler and registration functions



Semaphores

- A semaphore is a data structure that is shared by several processes.
- Semaphores are most often used to synchronize operations, when multiple processes access a common, non-shareable resource.
- By using semaphores, we attempt to avoid other multi-programming problems such as:
    - Starvation
    - Deadlock

POSIX Semaphores

- POSIX semaphores allow processes and threads to synchronize their actions.
- A semaphore is an integer whose value is **never allowed** to fall below **zero**.
- POSIX semaphores come in **two forms**:
  - named semaphores
  - unnamed semaphores.

Named Semaphores

- A named semaphore is identified by a name of the form /somename; that is, a null-terminated string
- Two processes can operate on the same named semaphore by passing **the same name** to sem_open().
- Named semaphore functions
  - sem_open()
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - sem_close()
  - sem_unlink()

Unnamed Semaphores

- An unnamed semaphore does not have a name.
  - The semaphore is placed in a region of memory that is shared between multiple threads or processes.
- A thread-shared semaphore
  - a global variable.
- A process-shared semaphore
  - must be placed in a shared memory region
    - POSIX or System V shared memory segment

Unnamed Semaphores

- Unnamed semaphore functions
  - **sem_init()**
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - **sem_destroy()**

A simple semaphore example

```c
//create & initialize semaphore
mutex = sem_open(SEM_NAME,O_CREAT,0644,1);
if(mutex == SEM_FAILED)  {
    perror("unable to create semaphore");
    sem_unlink(SEM_NAME);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process A enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process A leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(3);
}
sem_close(mutex);
sem_unlink(SEM_NAME);
```

```c
//create & initialize existing semaphore
mutex = sem_open(SEM_NAME,0,0644,0);
if(mutex == SEM_FAILED)  {
    perror("reader:unable to execute semaphore");
    sem_close(mutex);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process B enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process B leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(2);
}

sem_close(mutex);
```

A simple semaphore example

```
lucid@ubuntu:~$ ./PB
Process B enters the critical section at 1376420556
Process B leaves the critical section at 1376420556
Process B enters the critical section at 1376420558
Process B leaves the critical section at 1376420558
Process B enters the critical section at 1376420560
Process B leaves the critical section at 1376420560
Process B enters the critical section at 1376420562
Process B leaves the critical section at 1376420562
Process B enters the critical section at 1376420564
Process B leaves the critical section at 1376420564
Process B enters the critical section at 1376420566
Process B leaves the critical section at 1376420566
Process B enters the critical section at 1376420568
Process B leaves the critical section at 1376420568
Process B enters the critical section at 1376420570
Process B leaves the critical section at 1376420570
Process B enters the critical section at 1376420572
```

lucid@ubuntu: ~

File Edit View Terminal Help

```
lucid@ubuntu:~$ ./PA
Process A enters the critical section at 1376420554
Process A leaves the critical section at 1376420554
Process A enters the critical section at 1376420557
Process A leaves the critical section at 1376420557
Process A enters the critical section at 1376420560
Process A leaves the critical section at 1376420560
Process A enters the critical section at 1376420563
Process A leaves the critical section at 1376420563
Process A enters the critical section at 1376420566
```

Message Queues

- **Unlike pipes and FIFOs**, message queues support messages that have **structure**.
- Like FIFOs, message queues are persistent objects that must be initially created and eventually deleted when no longer required.
- Message queues are created with a specified maximum message size and maximum number of messages.
- Message queues are created and opened using a special version of the open system call, **mq_open**.

POSIX Message Queue Functions

- mq_open()
- mq_close()
- mq_unlink()
- mq_send()
- mq_receive()
- mq_setattr()
- mq_getattr()
- mq_notify()

mq_open(const char *name, int oflag,...)

- name
  - Must start with a slash and contain no other slashes
  - QNX puts these in the /dev/mqueue directory
- oflag
  - **O_CREAT** – to create a new message queue
  - **O_EXCL** – causes creation to fail if queue exists
  - **O_NONBLOCK** – usual interpretation
- mode – usual interpretation
- &mqattr – address of structure used during creation

mq_attr structure

- This structure, pointed to by the last argument of **mq_open**, has at least the following members:
  - mq_maxmsg – maximum number of messages that may be stored in the message queue
  - mq_msgsize – the size of each message, in bytes
  - mq_flags – not used by mq_open, but accessed by mq_getattr and mq_setattr
  - mq_curmsgs – number of messages in the queue

mq_close(mqd_t mqdes)

- This function is used to close a message queue after it has been used.
- As noted earlier, the message queue is not deleted by this call; it is persistent.
- The message queue's contents **are not altered** by **mq_close** unless a prior call(by this or another process) called **mq_unlink** (see next slide). In this respect, an open message queue is just like an open file: deletion is deferred until all open instances are closed.

mq_unlink(const char *name)

- This call is used to remove a message queue.
- Recall (from the previous slide) that the deletion is deferred until all processes that have the message queue open have closed it (or terminated).
- It is usually a good practice to call mq_unlink immediately after all processes that wish to communicate using the message queue have opened it. In this way, as soon as the last process terminates (closing the message queue), the queue itself is deleted.

Message Queue Persistence - I

- As noted, a message queue is persistent.
- Unlike a FIFO, however, the contents of a message queue are also persistent.
- It is not necessary for a reader and a writer to have the message queue open at the same time. A writer can open (or create) a queue and write messages to it, then close it and terminate.
- Later a reader can open the queue and read the messages.

Message Queue Persistence - II

- mkdir /dev/mqueue
- mount -t mqueue none /dev/mqueue
- ls -la /dev/mqueue

```
cihan@sdf-1:~/Desktop> ls -la /dev/mqueue/
total 0
drwxrwxrwt  2 root  root    80 2009-03-18 20:59 .
drwxr-xr-x 14 root  root  4600 2009-03-18 20:52 ..
-rwxr-x---  1 cihan users   80 2009-03-18 19:40 myqueue123
-rwxr-x---  1 cihan users   80 2009-03-18 20:59 test
cihan@sdf-1:~/Desktop>
```

mq_send(mqd_t mqdes, const char *msq_ptr, size_t msglen, unsigned msg_prio)

- **mqdes**
  - the descriptor required by mq_open
- **msg_ptr**
  - pointer to a char array containing the message
- **msglen**
  - number of bytes in the message; this must be no larger than the maximum message size for the queue
- **prio**
  - the message priority (0..MQ_PRIO_MAX); messages with larger (higher) priority leap ahead of messages with lower (smaller) priority

mq_receive(mqd_t mqdes, char *msq_ptr, size_t msglen, unsigned *msg_prio)

- **mqdes**
  - the descriptor returned by mq_open
- **msg_ptr**
  - pointer to a char array to receive the message
- **msglen**
  - number of bytes in the msg buffer; this should normally be equal to the maximum message size specified when the message queue was created
- **msg_prio**
  - pointer to a variable that will receive the message's priority
- The call returns the **size of the message, or -1**

A simple Message Queue Example

Sender

```
lucid@ubuntu:~/Downloads$ ./Drop
Usage: ./Drop [-q] -p msg_prio
lucid@ubuntu:~/Downloads$ ./Drop -q -p 11
I (5012) will use priority 11
lucid@ubuntu:~/Downloads$ ./Drop -p 110
I (5015) will use priority 110
lucid@ubuntu:~/Downloads$ ./Drop -p 17
I (5016) will use priority 17
lucid@ubuntu:~/Downloads$
```

```c
/* forcing specification of "-i" argument */
if (msgprio == 0) {
    printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
    exit(1);
}

/* opening the queue        --  mq_open() */
if (create_queue) {
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, NULL);
} else {
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
}
if (msgq_id == (mqd_t)-1) {
    perror("In mq_open()");
    exit(1);
}

/* producing the message */
currtime = time(NULL);
snprintf(msgcontent, MAX_MSG_LEN, "Hello from process %u (at %s).", my_pid, ctime(&currtime));

/* sending the message      --  mq_send() */
mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, msgprio);

/* closing the queue        -- mq_close() */
mq_close(msgq_id);
```

- **mq_dropone.c**

A simple Message Queue Example

Receiver

```
lucid@ubuntu:~/Downloads$ ./Take
Queue "/test":
        - stores at most 10 messages
                - large at most 8192 bytes each
        - currently holds 3 messages
Received message (56 bytes) from 110: Hello from process 5015 (at Fri Aug  9 07:
34:05 2013
).
```

```c
/* opening the queue        --  mq_open() */
msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
if (msgq_id == (mqd_t)-1) {
    perror("In mq_open()");
    exit(1);
}

/* getting the attributes from the queue        --  mq_getattr() */
mq_getattr(msgq_id, &msgq_attr);
printf("Queue \"%s\":\n\t- stores at most %ld messages\n\t\
       - large at most %ld bytes each\n\t- currently holds %ld messages\n",
       MSGQOBJ_NAME, msgq_attr.mq_maxmsg, msgq_attr.mq_msgsize, msgq_attr.mq_curmsgs);

/* getting a message */
msgsz = mq_receive(msgq_id, msgcontent, MAX_MSG_LEN, &sender);
if (msgsz == -1) {
    perror("In mq_receive()");
    exit(1);
}
printf("Received message (%d bytes) from %d: %s\n", msgsz, sender, msgcontent);

/* closing the queue    --  mq_close() */
mq_close(msgq_id);

mq_unlink(MSGQOBJ_NAME);
return 0;
```

- **mq_takeone.c**

The effect of fork on a message queue

- Message queue descriptors **are not (in general) treated as file descriptors**; the unique open, close, and unlink calls should already suggest this.
- Open message queue descriptors **are not inherited by child processes** created by fork.
- Instead, a child process must explicitly open (using mq_open) the message queue itself to obtain a message queue descriptor

Detecting non-empty queues

- mq_receive on an empty queue **normally causes a process to block**, and this may not be desirable.
- Of course, **O_NONBLOCK** could be applied to the queue to prevent this behavior, but in that case the mq_receive call will return -1, and our only recourse is to try mq_receive again later.
- With the **mq_notify** call we can associate a single process with a message queue so that it (the process) will be notified when the message queue changes state from empty to non-empty

mq_notify(mqd_t mqdes, const struct sigevent *notification)

- queuefd
  - as usual, to identify the message queue
- sigev
  - a struct sigevent object that identifies the signal to be sent to the process to notify it of the queue state change.
- Once notification has been sent, **the notification mechanism is removed**. That is, to be notified of the next state change (from empty to non-empty), the notification **must be reasserted**.

Changing the process to be notified

- Only one process can be registered (at a time) to receive notification when a message is added to a previously-empty queue.
- If you wish to change the process that is to be notified, you must remove the notification from the process which is currently associated (call mq_notify with NULL for the sigev argument), and then associate the notification with a different process.

Attributes

- **mq_getattr** (queuefd,&mqstat)
  - retrieves the set of attributes for a message queue to the struct mq_attr object named mqstat.
  - the mq_flags member of the attributes is not significant during mq_open, but it can be set later
- **mq_setattr** (queuefd,&mqstat,&old)
  - Set (or clear) to O_NONBLOCK flag in the mqattr structure for the identified message queue
  - Retrieve (if old is not NULL) the previously existing message queue attributes
  - Making changes to any other members of the mqattr structure is **ineffective**.

Timed send and receive

- Two additional functions, mq_timedsend and mq_timedreceive, are like mq_send and mq_receive except they have an additional argument, a pointer to a struct timespec.
- This provides the absolute time at which the send or receive will be aborted if it cannot be completed (because the queue is full or empty, respectively).

Shared Memory

- Sharing memory in POSIX (and many other systems) requires
  - creating a persistent "object" associated with the shared memory, and
  - allowing processes to connect to the object.
- creating or connecting to the persistent object is done in a manner similar to that for a file, but uses the shm_open system call.

Shared Memory Functions

- shm_open()
- mmap()
- munmap()
- ftruncate()
- shm_unlink()

shm_open (name, oflag, mode)

- name is a string identifying an existing shared memory object or a new one (to be created). It should begin with '/', and contain only one slash. In QNX 6, these objects will appear in a special directory.
- mode is the protection mode (e.g. 0644).
- shm_open returns a file descriptor, or –1 in case of error

shm_open (name, oflag, mode)

- oflag is similar to the flags for files:
  - O_RDONLY – read only
  - O_RDWR – read/write
  - O_CREAT – create a new object if necessary
  - O_EXCL – fail if O_CREAT and object exists
  - O_TRUNC – truncate to zero length if opened R/W

ftruncate(int fd, off_t len)

- This function (inappropriately named) causes the file referenced by fd to have the size specified by len.
- If the file was previously longer than len bytes, the excess is discarded.
- If the file was previously shorter than len bytes, it is extended by bytes containing zero.

mmap (void *addr, size_t len, int prot, int flags, int fd, off_t off);

- mmap is used to map a region of the shared memory object (fd) to the process' address space.
- The mapped region has the given len starting at the specified offset off.
- Normally addr is 0, and allows the OS to decide where to map the region. This can be explicitly specified, if necessary.
- mmap returns the mapped address, or –1 on error.(more on next slide)

mmap, continued

- prot – selected from the available protection settings:
  - PROT_EXEC
  - PROT_NOCACHE
  - PROT_NONE
  - PROT_READ
  - PROT_WRITE
- flags – one or more of the following:
  - MAP_FIXED – interpret addr parameter exactly
  - MAX_PRIVATE – don't share changes to object
  - MAP_SHARED – share changes to object

munmap (void *addr, size_t len)

- This function removes mappings from the specified address range.
- This is not a frequently-used function, as most processes will map a fixed-sized region and use shm_unlink at the end of execution to destroy the shared memory object (which effectively removes the mappings).

shm_unlink (char *name);

- This function, much like a regular unlink system call, removes a reference to the shared memory object.
- If the are other outstanding links to the object, the object itself continues to exist.
- If the current link is the last link, then the object is deleted as a result of this call.

A Simple Shared Memory Example

Sender

```c
/* creating the shared memory object    --  shm_open()  */
shmfd = shm_open(SHMOBJ_PATH, O_CREAT | O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
if (shmfd < 0) {
    perror("In shm_open()");
    exit(1);
}
fprintf(stderr, "Created shared memory object %s\n", SHMOBJ_PATH);

/* adjusting mapped file size (make room for the whole segment to map)      --  ftruncate() */
ftruncate(shmfd, shared_seg_size);

/* requesting the shared segment    --  mmap() */
shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shared_msg == NULL) {
    perror("In mmap()");
    exit(1);
}
fprintf(stderr, "Shared memory segment allocated correctly (%d byte

srandom(time(NULL));
/* producing a message on the shared segment */
shared_msg->type = random() % TYPES;
snprintf(shared_msg->content, MAX_MSG_LENGTH, "My message, type %d, num %ld", shared_msg->type, random());
```

```
lucid@ubuntu:~/Downloads$ ./SHMServer
Created shared memory object /foo1423
Shared memory segment allocated correctly (56 bytes).
lucid@ubuntu:~/Downloads$
```

- **shm_server.c**

A Simple Shared Memory Example

Receiver

```
/* creating the shared memory object    --  shm_open()  */
shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU | S_IRWXG);
if (shmfd < 0) {
    perror("In shm_open()");
    exit(1);
}
printf("Created shared memory object %s\n", SHMOBJ_PATH);

/* requesting the shared segment    --  mmap() */
shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shared_msg == NULL) {
    perror("In mmap()");
    exit(1);
}
printf("Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


printf("Message type is %d, content is: %s\n", shared_msg->type, shared_msg->content);
```

- **shm_client.c**

```
lucid@ubuntu:~/Downloads$ ./SHMClient
Created shared memory object /foo1423
Shared memory segment allocated correctly (56 bytes).
Message type is 6, content is: My message, type 6, num 1256344664
lucid@ubuntu:~/Downloads$
```

References

- http://cs.unomaha.edu/~stanw/091/csci8530/
- http://mij.oltrelinux.com/devel/unixprg/
- Man pages
- man mq_overview
- man mq_open, mq_close etc. etc. etc.
- http://forum.soft32.com/linux2/Utilities-listing-removing-POSIX-IPC-objects-ftopict15659.html