

MESSAGE QUEUES, NAMED PIPES, SIGNALS, ...

1

- 1) **Message Queues:** msgget, msgsnd, msgrcv, msgctl
- 2) **Named Pipes (FIFOs):** mkfifo
- 3) **Signals** (kill, sighandler, sigprocmask, sigaction, ...)

SYSTEM V IPC – MESSAGE QUEUES

Three types of asynchronous IPC originating from System V

- Shared Memory
- Message Queues
- Semaphores

Communication between processes on one and the same host
IPC structures, referred to by an identifier and a key



MESSAGE QUEUE

- Linked list of messages stored in the kernel
 - synchronization by the kernel
- Identified by a message queue identifier
- Different processes can use the same key to post/retrieve messages
 - Every message has a type, a nonnegative length, and actual data
 - `msgsnd` posts a message with a tag (type) at the <end of the queue>
 - `msgrcv` retrieves a message, optionally may not follow FIFO order (use type)
 - Resource limits: `MSGMAX`, `MSGMNB` (bytes), etc
- The use of type provides performance benefits
 - if interested in a particular type of message
 - server process can post/retrieve msg to/from clients using the client PID as type!
 - equivalent to merging multiple FIFO pipes in one



KERNEL DATA STRUCTURE FOR MESSAGE QUEUE

```
/* Structure of record for one message inside the kernel. */
struct msqid_ds
{
    struct ipc_perm msg_perm; /* structure describing operation permission
    */
    __time_t msg_stime; /* time of last msgsnd command */
    __time_t msg_rtime; /* time of last msgrcv command */
    __time_t msg_ctime; /* time of last change */
    unsigned long int __msg_cbytes; /* current number of bytes on queue */
    msgqnum_t msg_qnum; /* number of messages currently on queue */
    msglen_t msg_qbytes; /* max number of bytes allowed on queue */
    __pid_t msg_lspid; /* pid of last msgsnd() */
    __pid_t msg_lrpid; /* pid of last msgrcv() */
};
```



MESSAGE QUEUE: CREATE (OR OPEN)

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Creating a message queue id

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

```
int msgget(key_t key, int flag);
```

key_t key: **IPC_PRIVATE** (#include <sys/ipc.h>) creates a new segment. Otherwise, non-conflicting keys in diff processes derived by

```
key_t ftok(const char *path, int id);
```

- Path points to a file that the process can stat
- id is the project ID, only the last 8 bits are used

int flag: bitwise OR combination of 9-bit security r/w flag with

- **IPC_CREAT**: Create a new segment.
- **IPC_EXCL**: Used with **IPC_CREAT** to ensure that this call creates the segment. If the segment already exists, the call fails



MESSAGE QUEUE: OPERATIONS

- Message queue operations (send/receive a msg of size nbytes)

```
int msgsnd(int msgid, const void *msg, size nbytes, int flag);
```

```
int msgrcv(int msgid, void *msg, size_t nbytes, long type, int flag);
```

Return 0 on success, -1 on error

where msg is a user-defined structure (with type and buffer)

```
struct mymsg {  
    long mtype; // message type used as priority or identity  
    // for recv: type 0 -> first msg, >0 -> first type msg,  
    //                <0 first -> msg <= abs(type))  
    char mtext[512];}
```

- Send **blocks** if MSGMAX/MSGMNB, unless flag is IPC_NOWAIT (returns EAGAIN)
- Receive returns E2BIG if the message received is larger than nbytes
 - With MSG_NOERROR flag, the message is truncated (remaining is lost)

MESSAGE QUEUE STATS, SET & DESTROY

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Returns 0 if successful, -1 for error

This function accesses or changes privileges/characteristics on the message queue identified by `msqid` by executing command `cmd`

The `cmd` parameter can be

- `IPC_STAT` to obtain the `msqid_ds` structure (statistics) in `buf`
- `IPC_SET` to set (or get) privileges of owner, group, max bytes in `msqid_ds`
- `IPC_RMID` to destroy the queue immediately. Any process using the queue further will receive message `EIDRM` (identifier removed)



POSIX MESSAGE QUEUES: MQ

mq provides a similar C API for POSIX message queues. Briefly,

- message queues are created/opened using `mq_open`
 - identified by a named identifier (no `ftok`)
 - may be exposed in `/dev/mqueue`
- `mq_send` and `mq_receive` allow blocking/non-blocking calls
- `mq_send` allows priorities
 - equal priority messages queued as a FIFO
 - higher priority messages inserted before those of a lower priority
- mq provides asynchronous notification: `mq_notify`
- `mq_close`, `mq_unlink` to close/destroy the queue descriptor



NAMED PIPE (OR FIFO): MKFIFO

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

```
int mkfifo(const char *path, mode_t mode);
```

Returns: 0 if OK, -1 otherwise

Unrelated **shell commands** (or processes) can communicate using kernel buffer (of `mkfifo`), without intermediate temp files

- half-duplex communication (one-way)
- just one type
- mode same as before, e.g., 0777
- can use regular I/O operations (`open`, `read`, `write`, `unlink`) on the file

NAMED PIPE (FIFO) vs PIPE

PIPEs achieve linear process connections (as in assembly line)

```
prog1 < inputfile | prog2 | prog3
```

- FIFOs can use non-linear connections (via named file & tee)

```
mkfifo fifo1 fifo2 # δημιουργία 2 fifo αρχείων
```

```
prog3 < fifo1 &
```

```
prog4 < fifo2 &
```

```
prog1 < inputfile | tee fifo1 fifo2 | prog2
```

(tee: copies stdin to both files fifo1,fifo2 & stdout)

- Alternatively, use C API

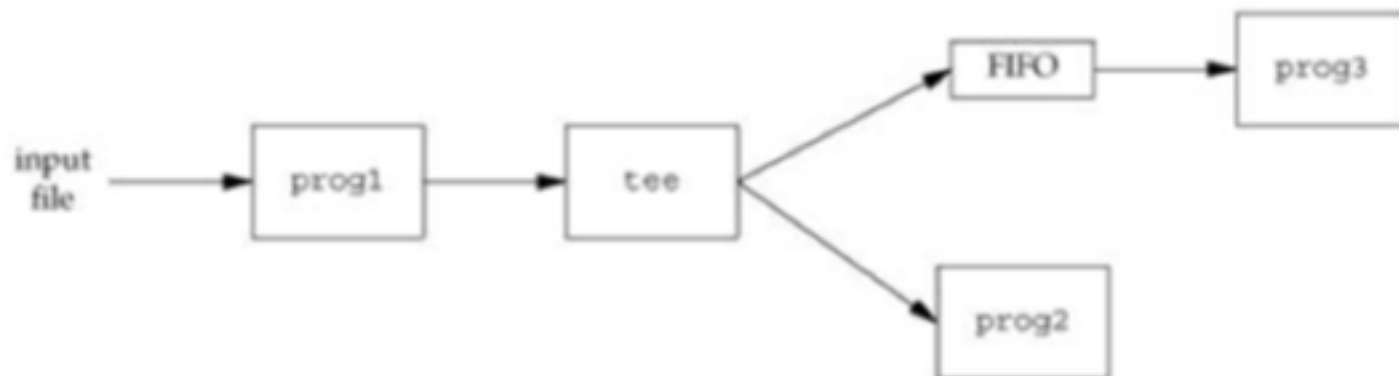
```
int open(const char *pathname, int flags);
```

```
int read(int fd, void *buf, size_t count);
```

```
int write(int fd, const void *buf, size_t count);
```

```
int close(fd);
```

NAMED PIPE: EXAMPLE



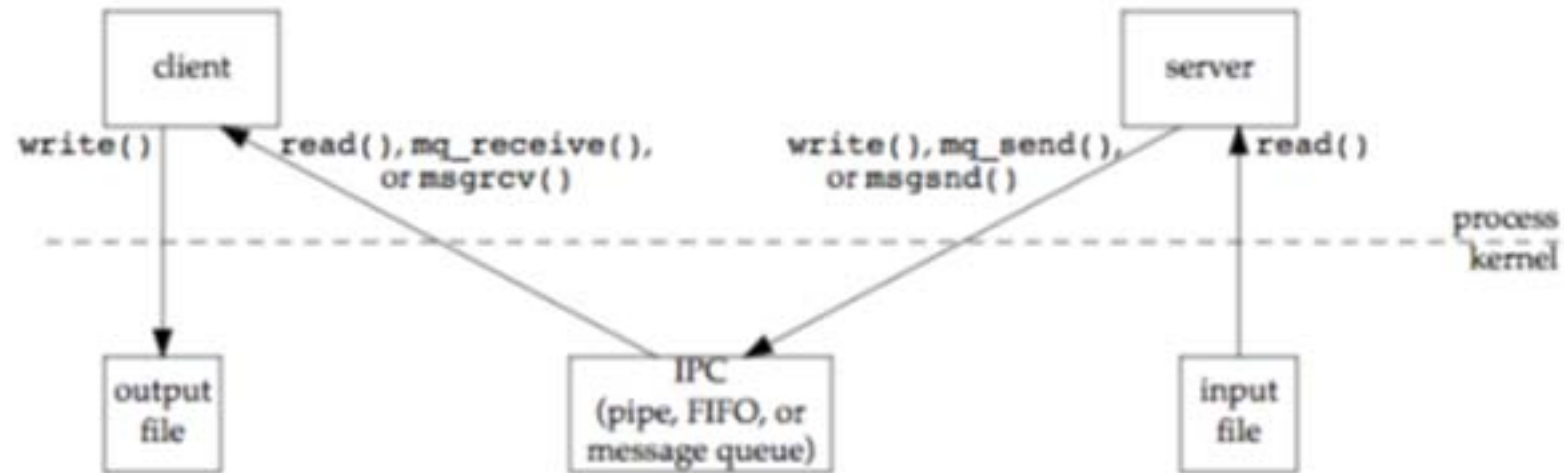
- Example: split input into sets

```
mkfifo fifo
```

```
grep pattern fifo > match &
```

```
gzcat file.gz | tee fifo | grep -v pattern > nomatch
```

IPC DATA FLOW EXAMPLE: ALTERNATIVES



POSIX SIGNALS

A signal (number) is sent from kernel, possibly at the request of a process, to notify another process that an asynchronous event occurred. Signals can be

- ◉ **blocked** by a process (masked out via `sigprocmask`, not delivered)
- ◉ **ignored** (via `signal`, `sigaction`)
- ◉ allowed to cause the **default** action
- ◉ caught at any point in the program & control transferred to a user-defined **signal handler** (`sighandler`)

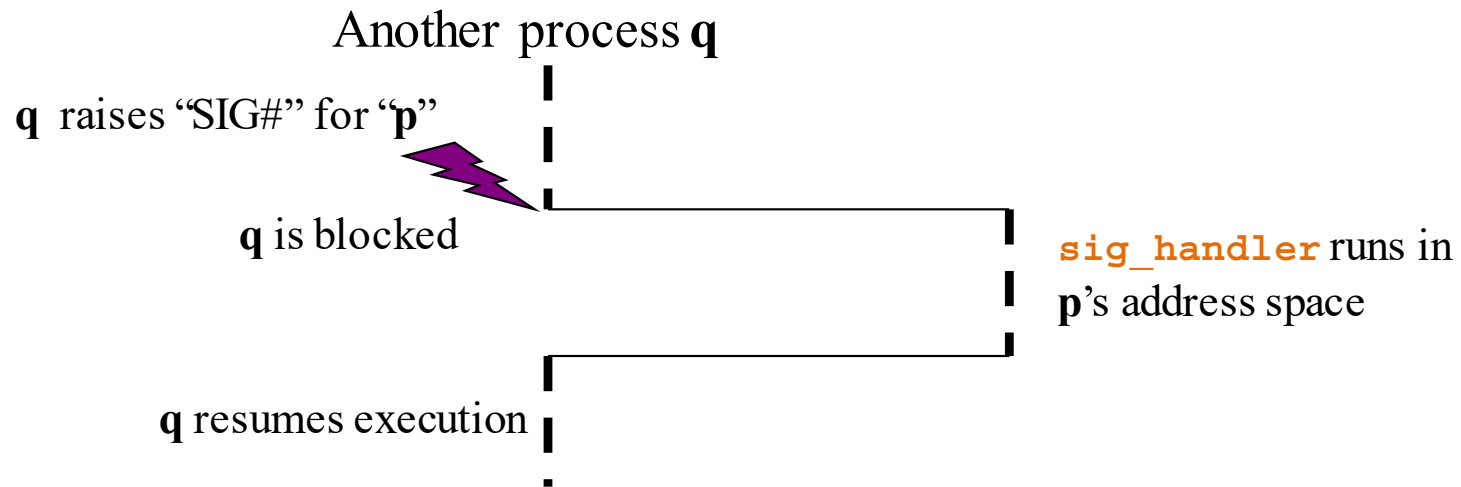
A signal is **pending** if it has been sent but not yet received.

Only one instance of a signal is queued (exception: POSIX real time signals)

SIGNAL HANDLER

```
/* code for process p */  
. . .  
signal(SIG#, sig_hndlr);  
. . .  
/* ARBITRARY CODE */
```

```
void sig_handler(...){  
...  
    /* handler code */  
}
```



SIGNAL HANDLER

```
#include <signal.h>
```

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

```
void (*signal(int signo, void (*sighandler)(int)))(int);
```

Where `signo` is the signal (defined in `signal.h`),

`sighandler` can be replaced by `SIG_IGN`, `SIG_DFL` (`SIGKILL/SIGSTOP`)

When a process forks, the child inherits the parent's signal mask (not for `SIGALRM`)

When the handler executes, control back to the interrupted process

- High overhead for signals (thousands of cycles)
- Handler may be erased after one invocation (`SIGALRM`, continuous coverage?)

uses only re-entrant functions:

“A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.”



REENTRANT FUNCTIONS (IN SIGNAL HANDLER OR FORK)

abort faccessat linkat select socketpair accept fchmod listen sem_post
stat access fchmodat lseek send symlink aio_error fchown lstat sendmsg
symlinkat aio_return fchownat mkdir sendto tcdrain aio_suspend fcntl
mkdirtat setgid tcflow alarm fdatasync mkfifo setpgid tcflush bind
fexecve mkfifoat setsid tcgetattr cfgetispeed fork mknod setsockopt
tcgetpgrp cfgetospeed fstat mknodat setuid tcseendbreak cfsetispeed
fstatat open shutdown tcsetattr cfsetospeed fsync openat sigaction
tcsetpgrp chdir ftruncate pause sigaddset time chmod futimens pipe
sigdelset timer_getoverrun chown getegid poll sigemptyset timer_gettime
clock_gettime geteuid posix_trace_event sigfillset timer_settime close
getgid pselect sigismember times connect getgroups raise signal umask
creat getpeername read sigpause uname dup getpgrp readlink sigpending
unlink dup2 getpid readlinkat sigprocmask unlinkat execl getppid recv
sigqueue utime execl getsockname recvfrom sigset utimensat execv
getsockopt recvmsg sigsuspend utimes execve getuid rename sleep wait
_Exit kill renameat socketatmark waitpid _exit link rmdir socket write

REENTRANT FUNCTIONS (IN SIGNAL HANDLER OR FORK)

```
_Exit(2), _exit(2), abort(3), accept(2), access(2), alarm(3), bind(2),  
cfgetispeed(3), cfgetospeed(3), cfsetispeed(3), cfsetospeed(3), chdir(2),  
chmod(2), chown(2), clock_gettime(2), close(2), connect(2), creat(3),  
dup(2), dup2(2), execl(3), execve(2), fchmod(2), fchown(2), fcntl(2),  
fdatasync(2), fork(2), fpathconf(2), fstat(2), fsync(2), ftruncate(2),  
getegid(2), geteuid(2), getgid(2), getgroups(2), getpeername(2),  
getpgrp(2), getpid(2), getppid(2), getsockname(2), getsockopt(2),  
getuid(2), kill(2), link(2), listen(2), lseek(2), lstat(2), mkdir(2),  
mkfifo(2), open(2), pathconf(2), pause(3), pipe(2), poll(2),  
pthread_mutex_unlock(3), raise(3), read(2), readlink(2), recv(2),  
recvfrom(2), recvmsg(2), rename(2), rmdir(2), select(2), sem_post(3),  
send(2), sendmsg(2), sendto(2), setgid(2), setpgid(2), setsid(2),  
setsockopt(2), setuid(2), shutdown(2), sigaddset(3), sigdelset(3),  
sigemptyset(3), sigfillset(3), sigismember(3), sleep(3), signal(3),  
sigpause(3), sigpending(2), sigprocmask(2), sigset(3), sigsuspend(2),  
socketatmark(3), socket(2), socketpair(2), stat(2), symlink(2), sysconf(3),  
tcdrain(3), tcflow(3), tcflush(3), tcgetattr(3), tcgetpgrp(3),  
tcsendbreak(3), tcsetattr(3), tcsetpgrp(3), time(3), timer_getoverrun(2),  
timer_gettime(2), timer_settime(2), times(3), umask(2), uname(3),  
unlink(2), utime(3), wait(2), waitpid(2), write(2).
```

EXAMPLES OF SIGNALS & KILL FUNCTION

Function `kill` can send signals explicitly to a process

- `int kill(pid_t pid, int signo);`
 - `pid > 0`, normal
 - `pid == 0`, send to every process with group ID same as the current process group ID
 - `pid < 0`, send to every process whose group ID = `|pid|`

There are more than 40 POSIX signals (see `signals.h`, or `man signal`). Examples

- job control, e.g., "Ctrl-C" (`SIGINT`, `SIGTERM`), "Ctrl-Z" (`SIGTSTP`), `fg` (`SIGCONT`)
- system events: mem error (`SIGSEGV`), /0 (`SIGFPE`), pipe fault (`SIGPIPE`), process dies (`SIGCHLD`), terminal hangup (`SIGHUP`), packet arrives (`SIGIO`), timer (`SIGALRM`)
`#include <unistd.h>`
`unsigned int alarm(unsigned int seconds)`
- `SIGABRT`, `SIGSEGV`, `SIGILL`, `SIGURG`, `SIGBUS`, `SIGUSR1/2`, `SIGTERM`, **`SIGKILL`**, **`SIGSTOP`**, ...

SIGNALS

Name	Description	ISO	C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Default action
SIGABRT	abnormal termination (abort)	•	•		•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•		•	•	•	•	terminate
SIGBUS	hardware fault		•		•	•	•	•	terminate+core
SIGCANCEL	threads library internal use							•	ignore
SIGCHLD	change in status of child		•		•	•	•	•	ignore
SIGCONT	continue stopped process		•		•	•	•	•	continue/ignore
SIGEMT	hardware fault				•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•		•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze							•	ignore
SIGHUP	hangup		•		•	•	•	•	terminate
SIGILL	illegal instruction	•	•		•	•	•	•	terminate+core
SIGINFO	status request from keyboard				•		•		ignore
SIGINT	terminal interrupt character	•	•		•	•	•	•	terminate
SIGIO	asynchronous I/O				•	•	•	•	terminate/ignore
SIGIOT	hardware fault				•	•	•	•	terminate+core
SIGKILL	termination		•		•	•	•	•	terminate
SIGLWP	threads library internal use							•	ignore
SIGPIPE	write to pipe with no readers		•		•	•	•	•	terminate
SIGPOLL	pollable event (poll)			XSI		•		•	terminate
SIGPROF	profiling time alarm (setitimer)			XSI	•	•	•	•	terminate

UNIX SIGNALS

- SIGABRT: Signal from `abort()`, process terminates abnormally
- SIGALRM: Timer set with `alarm()` expires
- SIGBUS: Hardware fault (memory)
- SIGCANCEL: Used by Solaris thread library, not for general use
- SIGCHLD: Child terminated, default action ignore, catch using `waitpid()`
- SIGCONT: Sent to a stopped process to continue (default action), vi catches this signal to redraw terminal screen
- SIGEMT: Hardware fault (emulator trap)
- SIGFPE: Arithmetic exception (divide by 0, floating point error, ...)
- SIGFREEZE: Used by Solaris to notify processes to take special actions before system freeze
- SIGHUP: disconnect detected, session leader terminates, daemon processes reread configuration files
- SIGILL: Illegal hardware instruction
- SIGINFO: Terminal driver generates signal when we type status key (CTRL-T)

UNIX SIGNALS

- SIGINT: Terminal driver generates signal when we type interrupt key (CTRL-C)
- SIGIO: Asynchronous I/O event
- SIGIOT: Hardware fault (I/O trap)
- SIGKILL: Kill process by admin (cannot be caught or ignored)
- SIGLWP: Used by Solaris thread library
- SIGPIPE: Upon pipe write after pipe reader has terminated (same for socket)
- SIGPOLL: Specific event on pollable device
- SIGPROF: Profiling interval timer (set by setitimer) has expired
- SIGPWR: Uninterruptible power supply (UPS) notifies a process of low power,
 - process then sends SIGPWR to init,
 - init handles system shutdown,
 - 2 entries in inittab: powerfail, powerwait

UNIX SIGNALS

- SIGQUIT: Terminal driver generates signal when typing quit (CTRL-\)
- SIGSEGV: Invalid memory reference
- SIGSTOP: Job-control signal to stop process, cannot be caught or ignored
- SIGSYS: Invalid system call
- SIGTERM: Termination signal sent by `kill`
- SIGTHAW: used by Solaris to notify processes to take action after resuming from suspension
- SIGTRAP: hardware fault (trap to debugger)
- SIGTSTP: Terminal driver generates signal when typing suspend (CTRL-Z)

UNIX SIGNALS

- SIGTTIN: background process tries to read from controlling terminal
- SIGTTOU: background process tries to write to controlling terminal
- SIGURG: urgent condition has occurred (out-of-band network data)
- SIGUSR1: user-defined for API
- SIGUSR2: user-defined for API
- SIGVTALRM: virtual interval timer set by `setitimer` expired
- SIGWAITING: used by Solaris thread library
- SIGWINCH: `ioctl()` changes window size
- SIGXCPU: process exceeds soft CPU time limit
- SIGXFSZ: process exceeds soft file size limit
- SIGRES: used only by Solaris for resource control

SIGNALS USING SIGACTION (DETAILED VERSION)

Supersedes `signal`

```
#include <signal.h>
```

```
• int sigaction(int signo, const struct sigaction * act,  
                struct sigaction *oact)
```

```
struct sigaction {  
    void (*sa_handler)(); // signal handler  
    sigset_t sa_mask; // additional signal to be blocked  
    int sa_flags; // various options for handling signal  
};
```

<https://notes.shichao.io/apue/ch10>



BLOCKING CERTAIN SIGNALS (SIGPROCMASK)

- Manipulate signal mask of a process (block/unblock)

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oset);`
how: {SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK}

- Manipulate signal sets

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

- Example

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
// critical region (no signals to worry about)
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```



SIGNALS INTERRUPTING BLOCKED SYSTEM CALLS

- Impact of signals on system calls
 - A system call may return prematurely
 - How to deal with this problem? Sometimes automatically done
 - Check the return value of the system call and act accordingly
 - Check `errno` to decide the error type (see `man -a read`)
 - Restart (system call using `sa_flags SA_RESTART`)



EXAMPLE: SENDING SIGNALS FROM THE KEYBOARD

Typing Ctrl-c (Ctrl-z) sends a SIGTERM (SIGTSTP) to every job in the foreground process group

- SIGTERM – default is to terminate each process
- SIGTSTP – default is to stop (suspend) each process

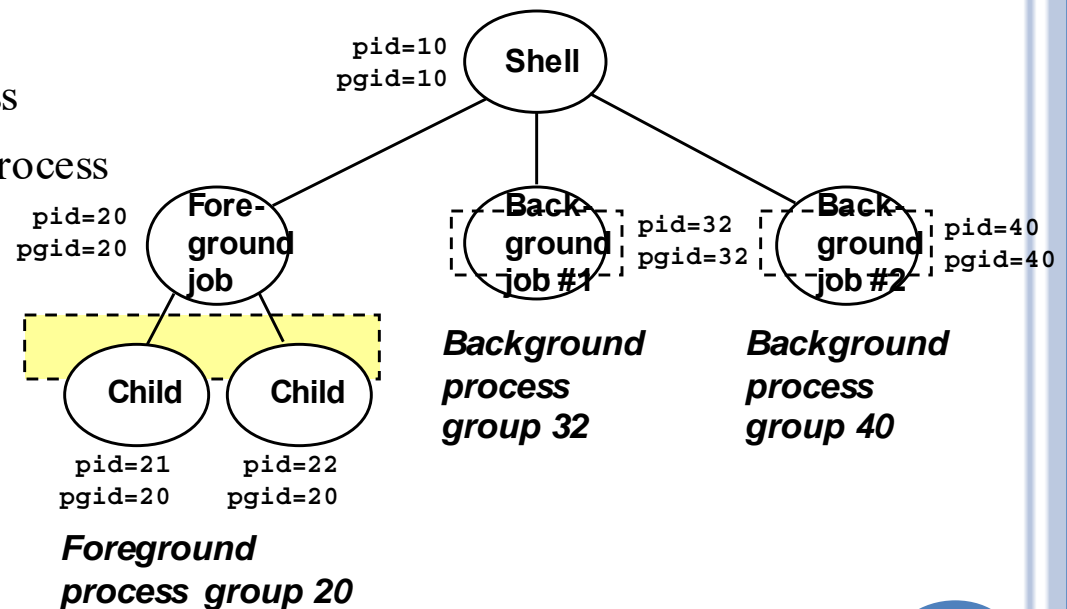
Process group-related Functions

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

```
int setpgid(pid_t pid, pid_t pgid);  
pid_t getpgid(pid_t pid);
```

```
pid_t getpgrp(void); // POSIX.1  
pid_t getpgrp(pid_t pid); // BSD
```

```
int setpgrp(void); // System V  
int setpgrp(pid_t pid, pid_t pgid); // BSD
```



Q &A

LIVING WITH NONQUEUEING SIGNALS

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

```
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);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

REVIEW

- Name a few commonly used signals.
- What is the typical behavior when a signal is received?
- Name different ways a program can deal with a signal
- Which routine can be used to block a signal?
- Which routine can be used to ignore a signal
- Which routine can be used to establish a user-defined signal handler?
- What to do to set up a code region that no-signals are allowed?
- Give two routines where signals are sent to a process
- What is the potential impact of signal on system calls?
- Is tcsh using canonical mode of input?
- How to turn on non-canonical mode of input?
- In non-canonical mode of input, how to decide when to return from a input routine?



SIGNAL CONCEPTS

- Kernel maintains `pending` and `blocked` bit vectors in the context of each process.
 - `pending` – represents the set of pending signals
 - Kernel sets bit `k` in `pending` whenever a signal of type `k` is delivered.
 - Kernel clears bit `k` in `pending` whenever a signal of type `k` is received
 - `blocked` – represents the set of blocked signals
 - Can be set and cleared by the application using the `sigprocmask` function.



RECEIVING SIGNALS

- Suppose kernel is returning from exception handler and is ready to pass control to process p .
- Kernel computes $\text{pnb} = \text{pending} \ \& \ \sim\text{blocked}$
 - The set of pending nonblocked signals for process p
- If $(\text{pnb} == 0)$
 - Pass control to next instruction in the logical flow for p .
- Else
 - Choose least nonzero bit k in pnb and force process p to **receive** signal k .
 - The receipt of the signal triggers some **action** by p
 - Repeat for all nonzero k in pnb .
 - Pass control to next instruction in logical flow for p .



NON-RE-ENTRANT FUNCTION

```
char *strtoupper(char *string) {  
    static char buffer[MAX_STRING_SIZE];  
    int index;  
    for (index = 0; string[index]; index++)  
        buffer[index] = toupper(string[index]);  
  
    buffer[index] = 0;  
    return buffer;  
}
```



RE-ENTRANT FUNCTION (POOR)

```
char *strtoupper(char *string) {  
    char *buffer;  
    int index; /* error-checking needed! */  
    buffer = malloc(MAX_STRING_SIZE);  
    for (index = 0; string[index]; index++)  
        buffer[index] = toupper(string[index]);  
    buffer[index] = 0;  
    return buffer;  
}
```



RE-ENTRANT VERSION

```
char *strtoupper_r(char *i n_str, char
*out_str) {
    int i ndex;
    for (i ndex = 0; i n_str[i ndex]; i ndex++)
        out_str[i ndex] =
toupper(i n_str[i ndex]);
    out_str[i ndex] = 0;
    return out_str;
}
```

ΜΕΡΙΚΑ ΣΗΜΑΤΑ

- `#include <signal.h>` Δείτε πίνακα 10.1 για άλλα σήματα
- • SIGHUP (1) terminal line got hung-up (θυμηθείτε nohup)
- • SIGINT (2) interrupt (Control^C)
- • SIGKILL (9) kill
- • **SIGUSR1 (10) user-defined signal 1**
- • SIGSEGV (11) Segm. Viol. – invalid memory reference
- • **SIGUSR2 (12) user-defined signal 2**
- • SIGALRM (14) alarm clock
- • SIGTERM (15) software termination signal (like 2)
- • SIGCONT (18) continue after stop
- • SIGSTOP (19) stop (**cannot be caught or ignored**).
- • SIGTSTP (20) stop signal from keyboard (Control^Z)



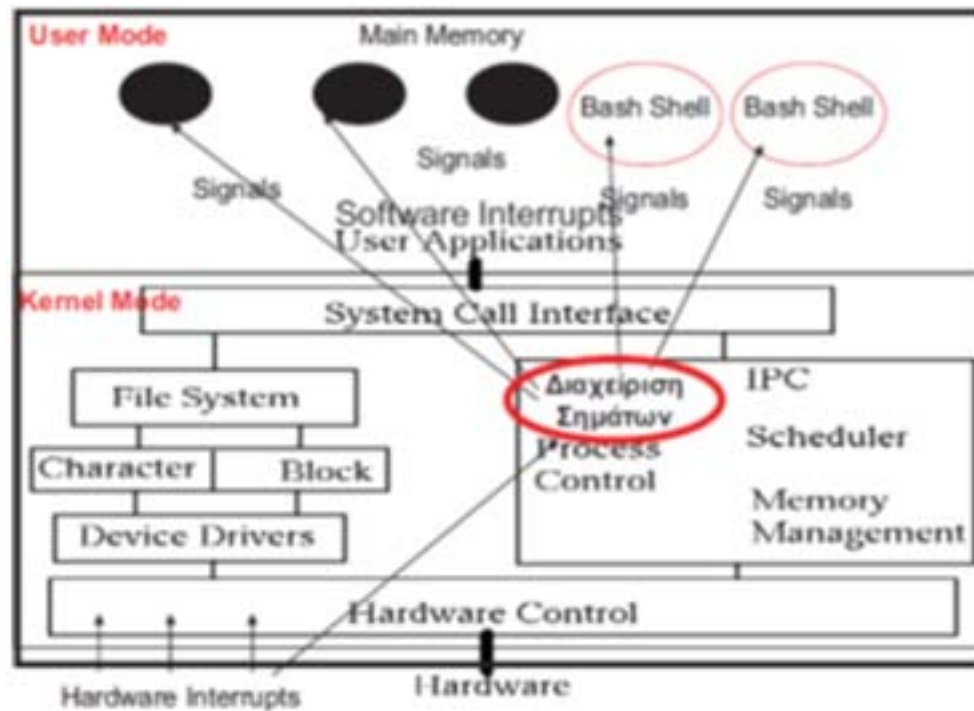
EXAMPLE: ALARM()

`unsigned int alarm(unsigned int s);`

- generates SIGALRM after s seconds
- returns time to next alarm
- only one pending alarm
- s=0 cancels alarm
- `pause()` waits until signal



ΔΙΑΧΕΙΡΙΣΗ ΣΗΜΑΤΩΝ (SIGNALS)



ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctrl-c)
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

COMMON SIGNAL NAMES AND NUMBERS

0	SIGNULL	Null Check access to pid
1	SIGHUP	Hangup Terminate
2	SIGINT	Interrupt Terminate
3	SIGQUIT	Quit Terminate with core dump
9	SIGKILL	Kill Forced termination; cannot be trapped
15	SIGTERM	Terminate Terminate; can be trapped
24	SIGSTOP	Stop Pause the process; cannot be trapped
25	SIGTSTP	Terminal stop Pause the process; can be
26	SIGCONT	Continue Run a stopped process



SENDING A SIGNAL: KILL()SYSTEM CALL

