Outline

- Review

- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO

- Spotlights
Review

Inter process communication
- Signals
- Fork
- Pipes
- FIFO

Spotlights
Review: multithreading

- Race conditions
  - non-determinism in thread order.
  - can be prevented by synchronization
  - atomic operations necessary for synchronization
- Mutex: Allows a single thread to own it
- Semaphores: Generalization of mutex, allows $N$ threads to acquire it at a time.
  - $P(s)$: acquires a lock
  - $V(s)$: releases lock
  - $\text{sem\_init}()$, $\text{sem\_destroy}()$
  - $\text{sem\_wait}()$, $\text{sem\_trywait}()$, $\text{sem\_post}()$
- Other problems: deadlock, starvation
Sockets

- `<sys/socket.h>`
- enables client-server computing
- **Client:** `connect()`
- **Server:** `bind()`, `listen()`, `accept()`
- **I/O:** `write()`, `send()`, `read()`, `recv()`
Review

- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO

- Spotlights
Preliminaries

• Each process has its own address space. Therefore, individual processes cannot communicate unlike threads.
• Interprocess communication: Linux/Unix provides several ways to allow communications
  • signal
  • pipes
  • FIFO queues
  • shared memory
  • semaphores
  • sockets
Unix/Linux allows us to handle exceptions that arise during execution (e.g., interrupt, floating point error, segmentation fault etc.).

A process receives a *signal* when such a condition occurs.

```c
void (*signal (int sig, void (*handler)(int)))( int )
```

determines how subsequent signals will be handled.

- **pre-defined behavior:** `SIG_DFL (default), SIG_IGN (ignore)`
- returns the previous handler.
Valid signals:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>abnormal termination</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>floating point error</td>
</tr>
<tr>
<td>SIGILL</td>
<td>illegal instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>interrupt</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>termination request</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>bus error</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>quit</td>
</tr>
</tbody>
</table>

The two signals `SIGSTOP, SIGKILL` cannot be handled.
int raise(int sig) can be used to send signal sig to the program.

Notes:

- There can be race conditions.
- signal handler itself can be interrupted.
- use of non-reentrant functions unsafe.
- sigprocmask can be used to prevent interruptions.
- handler is reset each time it is called.
```c
#include <stdio.h>

void sigproc() {
    signal(SIGINT, sigproc); /* */
    printf("you have pressed ctrl-c \n");
}

void quitproc() {
    printf("ctrl-\ disabled use ctrl-\ to quit");
    exit(0); /* normal exit status */
}

main() {
    signal(SIGINT, sigproc);
    signal(SIGQUIT, quitproc);
    printf("'\ disabled use ctrl-\ to quitn'");
    for(;;); /* infinite loop */
}
```
Fork

pid_t fork(void)

- `fork()` is a system call to create a new process
- In the child process, it returns 0
- In the parent process, it returns the PID (process id) of the child.
- The child PID can be used to send signals to the child process.
- returns -1 on failure (invalid PID)
Example

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

int main() {
    /*some code*/
    pid_t pid=fork();
    int i;
    if (pid) {
        for (i=0; i<5; i++){
            sleep(2);
            printf("parent process:%d\n", i);
        }
    }
    else {  
        for (i=0; i<5; i++){
            sleep(1);
            printf("child process:%d\n", i);
        }
    }
    }/*end child*/
}/*end main*/
```

parent process:0
child process:1
child process:2
parent process:1
child process:3
child process:4
parent process:2
parent process:3
parent process:4
Fork

- `fork()` makes a full copy of the parents address space.
- `pid_t getpid()` returns PID of the current process.
- `pid_t getppid()` returns PID of the parent process.
- `wait(int*)` is used to wait for the child to finish.
- `waitpid()` is used to wait for a specific child.

Zombies:

- The child process can exit before the parent
- Stray process is marked as `<defunct>`
- `preap` can be used to reap zombie processes.
Pipes

Pipes are used in unix to redirect output of one command to another. Pipes also allow parent processes to communicate with its children. Examples

- `ls | more` - displays results of `ls` one screen at a time
- `cat file.txt | sort` - displays contents of `file.txt` in sorted order

```c
int pipe(int FILEDES[2])
```

- A pipe can be thought of as a pair of file descriptors
- no physical file is associated with the file descriptor
- one end is opened in write mode.
- other end is opened in read mode.
Example

/* source: http://beej.us/guide */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h> /* ipc */

int main(void)
{
    int pfds[2];
    char buf[30];
    pipe(pfds);
    if (fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
    return 0;
}
FIFO queues may be thought of as named pipes.  
Multiple processes can read and write from a FIFO.  
Unlike pipes, the processes can be unrelated.  
FIFOs can be created using `mknod` system call.

```c
int mknod (const char *path, mode_t mode, dev_t dev)
```

- `<sys/stat.h>` contains the declaration for `mknod`.  
- `mknod` used to create *special* files - devices, fifos etc.  
- mode can have special bits such as `S_IFIFO | 0644`  
- dev is interpreted based on the mode.

**Example:** `mknod("myfifo", S_IFIFO | 0644 , 0);`
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#define FIFO_NAME "fifo"

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

Review

Inter process communication
- Signals
- Fork
- Pipes
- FIFO

Spotlights
Project spotlights

- Face finding with openCV
- Barcode scanner
- ImageIC
- Image2DXF
- Library database
- Simple Audio Visualizer
- Non-linear oscillator
- NoteDeluxe
- CUDA
- Visual mouse
- Wallpaper downloader