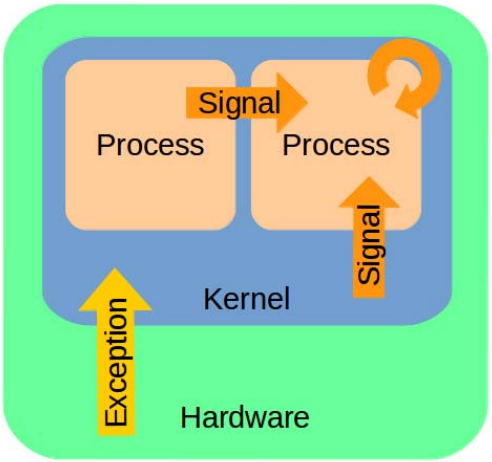
CSE 320 System Fundamentals II

Signals: software generated interrupts

YoungMin Kwon

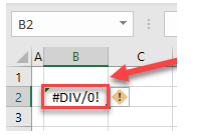
Signals

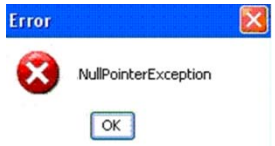
- Signals:
  - Software generated interrupts
  - A small message that notifies a process that an event of some type has occurred
  - A signal handler can handle the message
- Low-level hardware exceptions
  - Processed by the kernel's exception handlers



Signals

- Some examples of signals
  - SIGFPE (division by zero)
  - SIGALRM (registered time has passed)
  - SIGKILL (to kill a process)
  - SIGSEGV (illegal memory reference)
  - SIGINT (Ctrl-C)
  - SIGCHLD (stop/termination of child process)
  - SIGUSR1, SIGUSR2 (user defined signals)





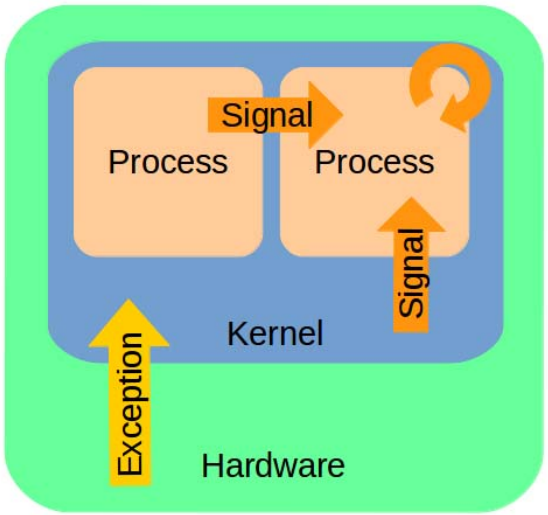
SUNY Korea

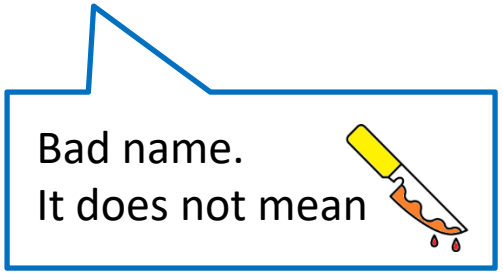
Signal Example

```c
//sigfpe.c
#include "common.h"
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void handler(int sig) {
    prnmsg("Caught division by zero!\n");
    _exit(1);
}

int main() {
    int zero = 0;
    ON_FALSE_EXIT(signal(SIGFPE, handler) != SIG_ERR, "signal");
    printf("%d\n", 1 / zero);
    return 0;
}
```

Signals

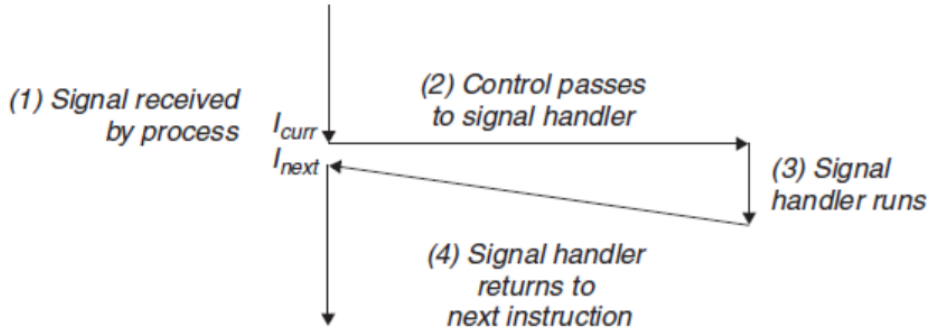
- Sending a signal
  - The kernel sends (delivers) a signal to a process by updating some state of the process
  - Invoked by a system event or by the `kill` function

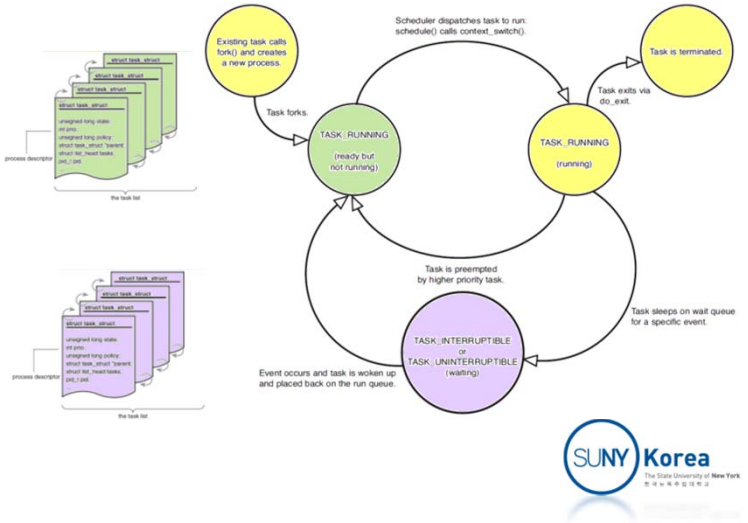




Signals

- Receiving a signal
  - A process receives a signal when it is forced by the kernel to react to the delivery of the signal
  - In the next context switch, if there is an unblocked pending signal for the process, its handler will be invoked first





SUNY Korea

Sending Signals

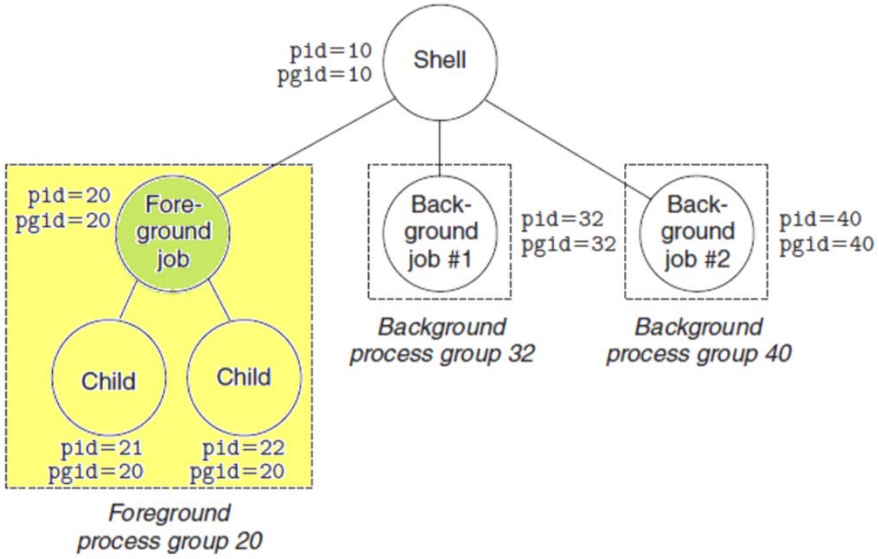
- Process Group
  - Each process belongs to one process group

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

  - `getpgrp:` returns the process group id
  - `setpgid:` sets the process group id of a process

Sending Signals

- kill program sends an arbitrary signal to another process
  - $ kill -9 20 sends 9 (SIGKILL) to process 20
  - $ kill -9 -20 sends 9 to all processes in process group 20



Sending Signals

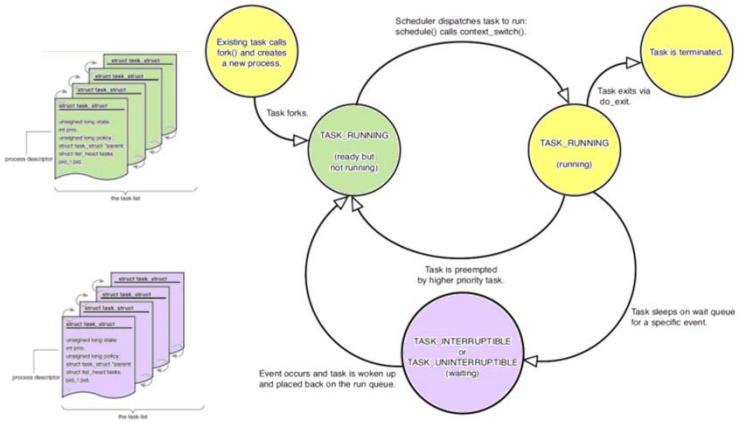
- A process can send signals to other processes

```
int kill(pid_t pid, int sig);
unsigned int alarm(unsigned int seconds);
```

  - kill:
    - If pid > 0: send sig to the process pid
    - If pid = 0: send sig to every process in the calling process' process group
    - If pid < 0: send sig to every process in the process group |pid|
  - alarm:
    - Send SIGALRM to itself after sec seconds

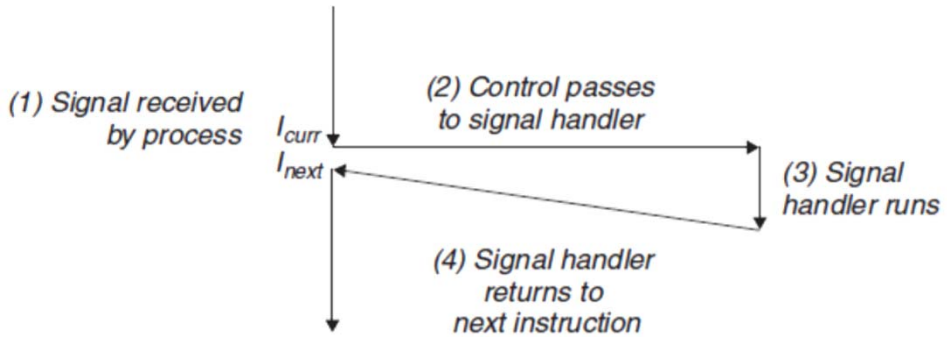
Sending Signals

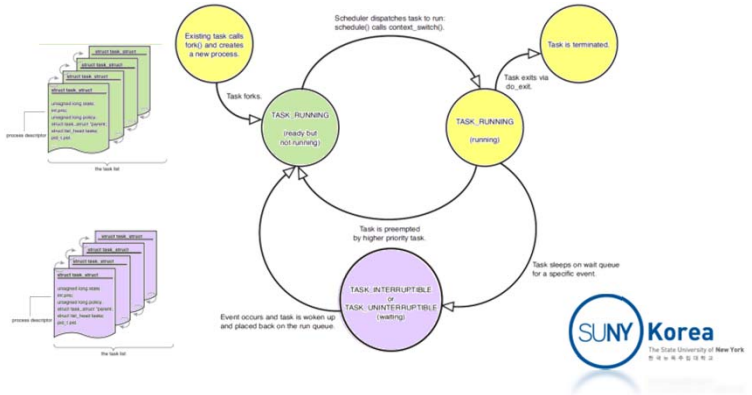
```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if(pid == 0 /*child*/) {
        printf("before pause\n");
        pause();
        printf("after pause\n");
    }
    else /*parent*/ {
        sleep(1);
        printf("sending SIGUSR1 to child\n");
        kill(pid, SIGUSR1);
    }
    return 0;
}
```



Receiving Signals

- When the kernel switches from kernel mode to user mode
  - It checks the set of unblocked pending signals (`pending & ~blocked`)
  - If this set is not empty the kernel forces the process to receive the signal
  - Once an action for the signal is complete the control passes back to the next instruction (I_{next})





SUNY Korea

Receiving Signals

- Default actions
  - Terminate the process
  - Terminate and dump core
  - Stop (suspend) until restarted by a SIGCONT signal
  - Ignores the signal

Receiving Signals

- signal function can change the action associated with a signal signum

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

  - handler is a user-defined signal handler function
  - If handler is SIG_IGN, the signal is ignored
  - If handler is SIG_DFL, the default action for the signal

Receiving Signals

```
//common.h
//
...
//it's not safe to use printf, strlen, etc. in a signal handler
extern int prnmsg(char *msg);

//common.c
//
...
//fwd decl
static int msglen(char *msg);
int prnmsg(char *msg) {
    write(1, msg, msglen(msg));
}

static int msglen(char *msg) {
    int i = 0;
    while (msg[i])
        i++;
    return i;
}
```

Receiving Signals

```c
//sigint.c
#include "common.h"
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    prnmsg("Caught SIGINT!\n");
}

int main() {
    ON_FALSE_EXIT(signal(SIGINT, handler) != SIG_ERR, "signal");
    printf("before pause\n");
    pause();
    printf("after pause\n");
    return 0;
}
```

Blocking and Unblocking Signals

- Applications can explicitly block and unblock selected signals using sigprocmask

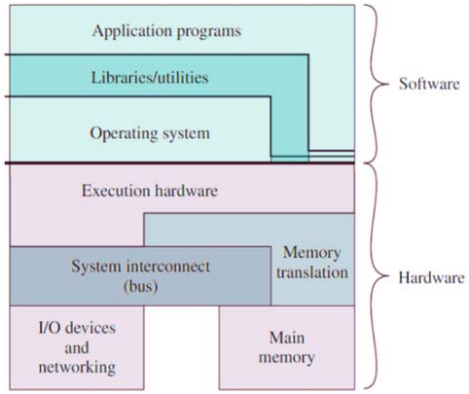
```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
```

  - how:
    SIG_BLOCK add signals in set to be blocked,
    SIG_UNBLOCK remove signals in set from blocked,
    SIG_SETMASK make blocked equal to set

  - oldset: if not NULL, the previous set of blocked signals is returned

Blocking and Unblocking Signals

- Helper functions




```
//make set the empty set
//  set: to batch system calls
int sigemptyset(sigset_t *set);

//make set the universal set
int sigfillset(sigset_t *set);

//add signum to set
int sigaddset(sigset_t *set, int signum);

//delete signum from set
int sigdelset(sigset_t *set, int signum);

//whether set has signum
int sigismember(const sigset_t *set, int signum);
```

```c
//mask.c
#include "common.h"
#include <signal.h>
#include <unistd.h>
void handler(int sig) {
    prnmsg("Caught SIGINT!\n");
}

int main() {
    ON_FALSE_EXIT(signal(SIGINT, handler) != SIG_ERR, "signal");

    sigset_t mask, oldmask;
    sigemptyset(&mask);          //mask = {}
    sigaddset(&mask, SIGINT);    //mask = { SIGINT }
#if 1
    //try Ctrl-C with and without this line
    sigprocmask(SIG_BLOCK, &mask, &oldmask);
#endif

    printf("before sleep\n");
    sleep(3);
    printf("after sleep\n");

    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return 0;
}
```

Safe Signal Handling

- Keep handlers simple
  - Handlers set global flags and let main program check the flags periodically
- Call only async-signal-safe functions in your handler
  - printf, sprintf, malloc, exit , … are not safe
  - write, _exit, … are safe
- Save and restore errno
  - On entering a handler save errno to a local variable and restore it before returning from the handler

Safe Signal Handling

- Protect access to shared global data
  - Temporarily block signals while accessing the data
- Declare global variables with volatile to update memory

```
volatile int g;
```

  - Compiler optimization may cache variables in registers and not reading them from the memory
  - Updates from handlers may be ignored without volatile
- Declare flags with `sig_atomic_t`

```
sig_atomic_t flag;
```

  - Reads and writes are guaranteed to be atomic (uninterruptable)

Correct Signal Handling

: Signals are not queued

```c
//noqueue.c
#include "common.h"
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

void handler(int sig) {
    int olderrno = errno; //save and restore errno

    ON_FALSE_GOTO(waitpid(-1, NULL, 0) >= 0, done, "waitpid");
    prnmsg("Reaped a child\n");

    sleep(1);
done:
    errno = olderrno;
}
```

Correct Signal Handling

: Signals are not queued

```
int main() {
    int i;
    char buf[100];

    ON_FALSE_EXIT(signal(SIGCHLD, handler) != SIG_ERR, "signal");

    for(i = 0; i < 10; i++) {
        if(fork() == 0 /*child*/) {
            prnmsg("child\n");
            return 0;
        }
    }

    ON_FALSE_EXIT(read(0, buf, sizeof(buf)) >= 0, "read");
    ON_FALSE_EXIT(signal(SIGCHLD, SIG_IGN) != SIG_ERR, "signal");
    return 0;
}
```

Correct Signal Handling

: Signals are not queued

```
void handler(int sig) {
    int olderrno = errno;

    while(waitpid(-1, NULL, 0) > 0)
        prnmsg("Reaped a child\n");

    sleep(1);
done:
    errno = olderrno;
}
```

Synchronizing Flows

```
//task: add the pid of a new child process to a queue and
//      remove it when the child process is terminated.
//
static int num_jobs = 0;

void add_to_queue(pid_t pid) {
    // assume that pid is added to an imaginary queue
    num_jobs++;
}

void delete_from_queue(pid_t pid) {
    // assume that pid is removed from an imaginary queue
    ON_FALSE_EXIT(num_jobs > 0, "nonexistent job");
    num_jobs--;
}

void handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    while((pid = waitpid(-1, NULL, 0)) > 0)
        delete_from_queue(pid);
    errno = olderrno;
}
```

Synchronizing Flows

```
int main() {
    char *argv[] = { "/usr/bin/cal", "-m", "12", NULL };
    pid_t pid;
    sigset_t mask, prev;
    sigfillset(&mask);  //universal set: masks all signals

    signal(SIGCHLD, handler);
    while(1) {
        if((pid = fork()) == 0 /*child*/)
            execve(argv[0], argv, NULL);

        //block concurrent access to the job queue
        sigprocmask(SIG_BLOCK, &mask, &prev);

        //BUGBUG: if a child exited before the parent adds pid
        //to the queue, delete_from_queue cannot find pid and
        //the pid added later won't be removed
        add_to_queue(pid);

        sigprocmask(SIG_SETMASK, &prev, NULL);
    }
    return 0;
}
```

```c
int main() {
    char *argv[] = { "/usr/bin/cal", "-m", "12", NULL };
    pid_t pid;
    sigset_t mask, prev;
    sigfillset(&mask);  //set universe: masks all signals

    signal(SIGCHLD, handler);
    while(1) {
        //block before fork to access the queue exclusively
        sigprocmask(SIG_BLOCK, &mask, &prev);

        if((pid = fork()) == 0) {
            //unblock signals for child
            sigprocmask(SIG_SETMASK, &prev, NULL);
            execve(argv[0], argv, NULL);
        }

        add_to_queue(pid);

        //unblock for parent after adding pid to queue
        sigprocmask(SIG_SETMASK, &prev, NULL);
    }

    return 0;
}
```

Explicit Waiting for Signals

```c
#include "common.h"
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

volatile sig_atomic_t pid;

void sigchld_handler(int sig) {
    int olderrno = errno;
    //main will wait until pid is set to a nonzero value
    pid = waitpid(-1, NULL, 0);
    errno = olderrno;
}

void sigint_handler(int sig) {
    _exit(0);
}
```

```c
int main() {
    char *argv[] = {"/usr/bin/cal", "-m", "12", NULL};
    sigset_t mask, prev;
    ON_FALSE_EXIT(sigfillset(&mask) == 0, "sigfillset");

    ON_FALSE_EXIT(signal(SIGINT,  sigint_handler)  != SIG_ERR, "signal");
    ON_FALSE_EXIT(signal(SIGCHLD, sigchld_handler) != SIG_ERR, "signal");

    while(1) {
        ON_FALSE_EXIT(sigprocmask(SIG_BLOCK, &mask, &prev) == 0, "sigprocmask");
        if((pid = fork()) == 0 /*child*/) {
            ON_FALSE_EXIT(sigprocmask(SIG_SETMASK, &prev, NULL) == 0, "sigprocmask");
            execve(argv[0], argv, NULL);
            ON_FALSE_EXIT(0, "execve");
        }

        pid = 0;
        ON_FALSE_EXIT(sigprocmask(SIG_SETMASK, &prev, NULL) == 0, "sigprocmask");
        while(!pid)         //OK but wasteful spin
            ;/*spin*/

        //while(!pid)       //BUGBUG: SIGCHLD can be received after
        //  pause();        //        !pid check but before pause()

        //while(!pid)       //OK but up to 1 sec delay
        //  sleep(1);

        printf("child pid: %d\n", pid);
    }
    return 0;
}
```

```c
int main() {
    char *argv[] = {"/usr/bin/cal", "-m", "12", NULL};
    sigset_t mask, prev;
    ON_FALSE_EXIT(sigfillset(&mask) == 0, "sigfillset");

    ON_FALSE_EXIT(signal(SIGINT,  sigint_handler)  != SIG_ERR, "signal");
    ON_FALSE_EXIT(signal(SIGCHLD, sigchld_handler) != SIG_ERR, "signal");

    while(1) {
        ON_FALSE_EXIT(sigprocmask(SIG_BLOCK, &mask, &prev) == 0, "sigprocmask");
        if((pid = fork()) == 0 /*child*/) {
            ON_FALSE_EXIT(sigprocmask(SIG_SETMASK, &prev, NULL) == 0, "sigprocmask");
            execve(argv[0], argv, NULL);
            ON_FALSE_EXIT(0, "execve");
        }

        pid = 0;
        while(!pid)
            sigsuspend(&prev);
            //equivalent to an atomic version of
            //    sigprocmask(SIG_SETMASK, &prev, &tmp);
            //    pause();
            //    sigprocmask(SIG_SETMASK, &tmp, NULL);
        ON_FALSE_EXIT(sigprocmask(SIG_SETMASK, &prev, NULL) == 0, "sigsuspend");
        printf("child pid: %d\n", pid);
    }
    return 0;
}
```

Programming Assignment 8

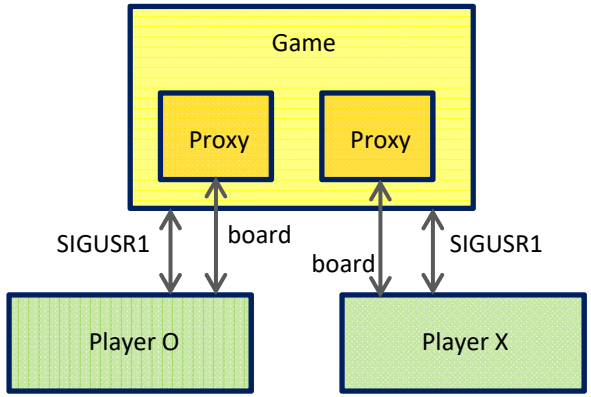
- The assignment is about playing a TicTacToe game between two processes
  - Download signal.zip and implement TODO parts
  - Upload to blackboard in a single zip file.
- Due date: TBD

Programming Assignment 8

- Three processes:
  - 2 actual players of the game.
    - It takes a mark (O or X) as its parameter: (e.g. player O or player X)
    - It reads a board from stdin
    - It puts its mark on the board and prints the board to stdout
  - game: using proxy players,
    - It loads the two actual players
    - Makes the actual players play (read and write the board) in turn through proxy

Programming Assignment 8

- Flow of control
  - Game loads two actual players as it child processes
    - Communication with them through proxy players



  - Proxy connects the stdin/stdout of the players to its fds created by pipe
  - Game and players are synchronized by sending/receiving SIGUSR1 signals
- Due date TBD

```
//board manager
//
#ifndef __BOARD_MGR__
#define __BOARD_MGR__

//mark for empty spot
#define EMPTY       '.'
//mark of the other player
#define OTHER(m)    ((m) == 'O' ? 'X': 'O')

//board manager
//
typedef struct board_mgr {
    void (*destroy       )(struct board_mgr *self);
    int  (*has_winner    )(char *board);
    int  (*has_empty_spot)(char *board);
    void (*print_board   )(char *mark, char *board);
    void (*read_board    )(int fd, char *board);
    void (*write_board   )(int fd, char *board);
} board_mgr_t;

extern board_mgr_t *make_board_mgr();

#endif
```

```
//
// Actual player program
//
typedef struct player {
    char *mark;          //either "O" or "X"
    board_mgr_t *bmgr;   //board manager

    void (*destroy      )(struct player *self);
    void (*read_board   )(struct player *self, char *board);
    void (*write_board  )(struct player *self, char *board);
    int  (*simulate_play)(struct player *self, char mark,
                          char *board, int *inx);
    int  (*put_mark     )(struct player *self, char *board);
    void (*play         )(struct player *self);
} player_t;
```

```c
// signal handler
//
static volatile int my_turn = 0;
static void sig_handler(int sig) {
    //TODO: if sig is SIGUSR1, set my_turn to 1

}


//play the game
static void player_play(player_t *self) {
    char board[10] = ".........";

    //block SIGUSR1 signal using sigprocmask
    sigset_t mask, prev;
    //TODO: empty mask,
    //      add SIGUSR1 to mask,
    //      block signals in mask and get the previous set in prev


    //TODO: set sig_handler as a signal handler for SIGUSR1
```

```
    while(!self->bmgr->has_winner(board) &&
            self->bmgr->has_empty_spot(board)) {

        //wait for my turn
        //TODO: until my_turn is set, wait for a signal
        //hint: use sigsuspend


        //set my turn to 0
        my_turn = 0;

        //read board
        //TODO: read the board from stdin (actually from a proxy player)


        if(self->bmgr->has_winner(board))
            break;
        if(!self->bmgr->has_empty_spot(board))
            break;

        //mark on the board
        self->put_mark(self, board);

        //write board
        //TODO: write the board to stdout (actually to a proxy player)

    }
}
```

```
//Proxy for the actual palyer
//
typedef struct player_proxy {
    char *fname;        //executable file name for the player
    char *mark;         //mark of the player
    pid_t pid;          //the pid of the player
    int fd[2];          //pipe connected to this player
    board_mgr_t *bmgr;  //board manager

    void (*load       )(struct player_proxy *self);
    void (*end        )(struct player_proxy *self);
    void (*read_board )(struct player_proxy *self, char *board);
    void (*write_board)(struct player_proxy *self, char *board);
    void (*print_board)(struct player_proxy *self, char *board);
} player_proxy_t;

//The game
//
typedef struct game {
    //whose turn is it now (either 0 or 1)
    int turn;
    //proxy for the two players
    player_proxy_t *proxy[2];

    void (*load_players)(struct game *self);
    void (*end         )(struct game *self);
    void (*play        )(struct game *self);
} game_t;
```

```c
// signal handler
//
volatile int switch_turn;
static void sig_handler(int sig) {
    //TODO: if sig is SIGUSR1, set switch_turn to 1
}

//load the actual player
//   - the actual player is fname (an executable program)
//   - N.B. this method is called from a CHILD process
static void player_load(player_proxy_t *self) {
    //TODO: connect fd[0] to stdin and fd[1] to stdout using dup2,
    //      close unnecessary fds

    //TODO: execute fname using execvp
    //      - fname needs mark as its only parameter
}

//terminate the actual player and destroy the proxy
static void player_end(player_proxy_t *self) {
    //TODO: send SIGKILL to the actual player,
    //      reap the player,
    //      close unnecessary fds

    //TODO: destroy the board manager,
    //      free self
}
```

```
//load the players
static void game_load_players(game_t *self) {
    for(int i = 0; i < 2; i++) {
        player_proxy_t *player = self->proxy[i];

        //TODO: create a child process using fork
        //      - child: load the player
        //      - parent: set the player's pid
    }
…
}

//play the game
static void game_play(game_t *self) {
    char board[10] = ".........";

    //block SIGUSR1 signal using sigprocmask
    sigset_t mask, prev;
    //TODO: empty mask,
    //      add SIGUSR1 to mask,
    //      block signals in mask and get the previous set in prev


    //TODO: set sig_handler as a signal handler for SIGUSR1
```

```
    proxy->write_board(proxy, board);
    while(!proxy->bmgr->has_winner(board) &&
           proxy->bmgr->has_empty_spot(board)) {

        //wait until the player marks
        //TODO: until switch_turn is set, wait for a signal


        //read and print the board
        //TODO: read the board from the actual player using its proxy,
        //      print the board


        //check the result
…

        //switch the turn
        //TODO: switch the self->turn index (0 <-> 1),
        //      set the proxy variable,
        //      set switch_turn to 0


        //let the other player play
        //TODO: write the board to the actual player using its proxy

    }
}
```

To compile:

```
$ make
gcc common.c board_mgr.c game.c -o game
gcc common.c board_mgr.c player.c -o player

$ ls
Makefile  board_mgr.c  board_mgr.h  common.c
common.h  game  game.c  player  player.c
```

To execute:

```
$ ./game ./player ./player
O:
. . .
. . .
. . .
O:
. . .
. . .
. . O
X:
. . .
. . .
. X O
O:
. . .
. . .
O X O
X:
. . .
. . X
O X O
…
```

```
O:
. . .
. O X
O X O
X:
X . .
. O X
O X O
O:
X . O
. O X
O X O
winner: O
```