Exceptional Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches (e.g. if/else, switch, for, while)
  - Call and return (e.g. function call, return)

  Both react to changes in *program state*

- Insufficient for a useful system:
  - Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

1. **Exceptions :**
   - Low level, implemented with hardware and OS
     a. **interrupts**, e.g. I/O interrupts (caused by disk data arrival, network data arrival, ..)
     b. **traps**, intentional, e.g. system-call, break-points
     c. **faults**, unintentional but possibly recoverable, e.g. page fault
     d. **aborts**, unintentional and unrecoverable

Shell Lab

2. **Process context switch**
   - Implemented by OS software and hardware timer
   - control flow from one process to another, scheduled by OS kernel

3. **Signals**
   - Implemented by OS software
   - a small message sent to a process from kernel

4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
   - Implemented by C runtime library

Process

- What is a *process?*
  - A process is an instance of a running program
  - 2 key abstractions:
    - each process seems to have exclusive use of the CPU
    - each process seems to have exclusive use of main memory
  - The reality is:
    - Process executions interleaved, (alternatively executed)
    - Address spaces managed by virtual memory system
  - difference with a program:
    - program can be an executable (e.g. binary file), saved in disk
    - process is an instance of a program running in memory

Process basic states

From a simple perspective, we can think of a process as being in one of three states

- Running (Runnable)
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- Stopped
  - Process execution is *suspended* and will not be scheduled until further notice (i.e. SIGCONT signal)
- Terminated
  - Process is stopped permanently

Four basic process control functions

- `fork()`
- `exec()` (and other variants such as `execve()`)
- `exit()`
- `wait()` (and variants like `waitpid()`)

Standard on all UNIX-based systems

Running Processes

- **`int exec()`**
  - to load and run **another** program in the **current** process
    - program is changed but process ID is not
  - Never returns on successful execution
  - More useful variant is **`int execve()`**

  `int execve(char *filename, char *argv[], char *envp[])`
  - `filename: executable file to run`
  - `argv[]: argument list, argv[0] == filename`
  - `envp[]: environment variable list`

      `-- in format of` "name=value" strings (e.g., `USER=droh`)

      The global variable, "environ", which saves all environment variables.

Returning from Processes

- **`void exit(int status)`**
  - Terminates the current process with a return status (error code)
  - Normal return with status 0 (other numbers indicate an error)
  - OS frees resources such as heap memory and open file descriptors and so on…
  - Reduce to a zombie state
    - process is terminated but there's still its entry in the process table (managed by OS)
    - Must wait to be reaped by the parent process (or the init process if the parent died)
    - To reap a child, process, use wait() or waitpid()
      - `exit` is called once but never returns!

Waiting for Processes

- **`pid_t wait(int *child_status)`**
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
    - When wait returns a pid > 0, child process has been reaped
  - update child_status, can be used to check the exit status and reason:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED …`
  - More useful variant is `waitpid()`

    `pid_t waitpid(pid_t` **`pid`**`, int *status, int` **`options`**`)`

    can specify a child by **pid**, can use more **options**
  - search online for more

Process Example (1)

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{


   printf("Parent!\n");
}
```

- What are the possible output (assuming fork succeeds) ?

  - ```
    Child!
    Parent!
    ```
  - ```
    Parent!
    Child!
    ```

- How to get the child to always print first?

How to guarantee the order of execution?

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else {
   waitpid(child_pid, &status, 0);

   printf("Parent!\n");
}
```

- Waits until the child has terminated.
- Parent can inspect exit status of child using 'status'
  - WEXITSTATUS(status)

- Output always:
  `Child!`
  `Parent!`

Process Example (2)

```
int status;
pid_t child_pid = fork();
char* argv[] = {“/bin/ls”, “-l”, NULL};
char* env[] = {…, NULL};

if (child_pid == 0){
   /* only child comes here */

   execve(“/bin/ls”, argv, env);
   printf(“Child”);
   /* will child reach here? */
}
else{
   waitpid(child_pid, &status, 0);

   … parent continue execution…
}
```

- Will child reach here?

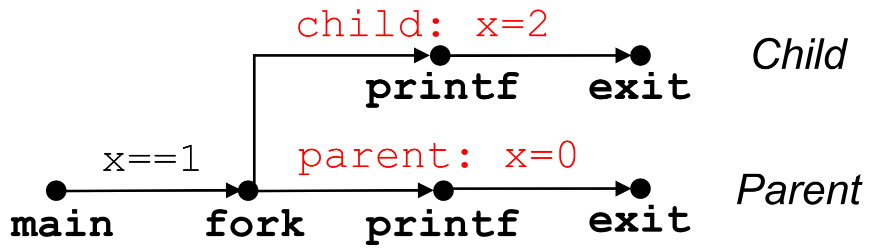
Process Graphs

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
       exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c



events in child/parent are partially ordered!
arrows determine relative orders,
if no arrow, order is unknown

Signals

- A *signal* is a small message that notifies a process that an event
  - sent from the kernel (sometimes at the request of another process)
  - sent to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
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

SIGTSTP default action: Stop Ctrl-Z from keyboard
SIGCONT default action: continue run for a stopped process

Sending signals

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as Ctrl-C (SIGINT), divide-by-zero (SIGFPE), or the termination of a child process (SIGCHLD)
  - The user used a `kill` command (in terminal)
  - Another program called the `kill()` function

```
int kill(pid_t pid, int sig);    //send signal "sig" to process pid
    // if send signal to a process group, use –pid, pid means the PGID
```

Receiving signals

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal

- Three possible ways to react:
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***

Receiving signals

- Signal Handler
  - a user-level function to catch a specific signal
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired
  - implement in the form: `void  handler(int signum){ … }`
  - install the handler by:

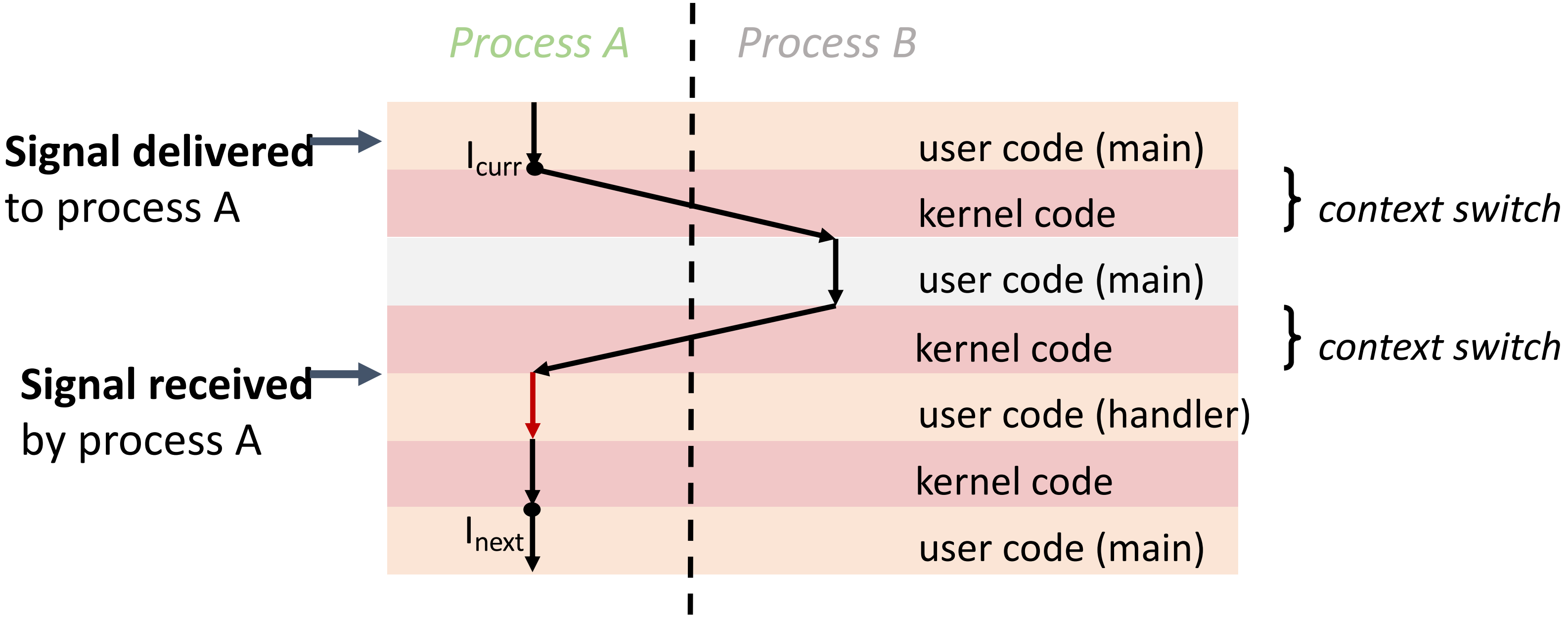
```
handler_t *signal(int signum, handler_t *handler)
```

//signal function will modify the default action associated with signum

Receiving signals

- Receiving a signal is non-queuing
  - There is only one bit in the context per signal
  - Receiving 1 or 300 SIGINTs looks the same to the process

- Signals are received at a context switch



Receiving signals

- Blocking/Unblocking signals
  - Sometimes code needs to run through a section that can't be interrupted
  - use `sigprocmask()` to block/unblock/set blocking masks
  - support functions:
    - `sigemptyset()`, `sigaddset()`, `sigfillset()`, `sigdelset()`

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    /* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

Tsh functionalities and specifications (1/7)

- **command line**, consists of a “name” and 0-or-more “arguments”

  e.g. “`./myspin 10`”. name: `./myspin`, 1 argument: “`10`”

  if name is a **built-in command**:

  tsh executes the command immediately in “tsh” process

  a built-in command corresponds to a function in tsh.c -- you need to implement these functions

  if name is a pathname of an **executable**:

  tsh forks a child process, and load/run the executable in the child

Hints:

1. This part is left for you to implement.
2. **parseline**() is already implemented, to parse the cmdline into char** argv
3. use **fork**() to fork a child process, and use **execve**() to load/run an executable

Tsh functionalities and specifications (2/7)

- **background** job & **foreground** job

  If command line ends with “&”,

  it is a **background** job, i.e. shell process won’t wait the child process

  otherwise, it’s a **foreground** job,

  i.e. shell process will wait the child process to terminate/stopped

Hints:

1. **parseline**() will return you a “bg/fg” boolean indicator.
2. implement your waiting function in **waitfg**()

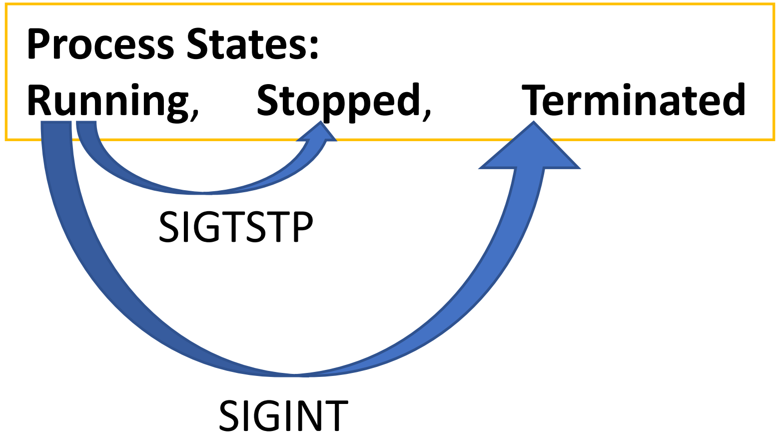
   A Recommendation (provided in the writeup) :

   - In `waitfg`, use a busy loop around the `sleep` function.

   loop iterates until this process is no longer the foreground process, e.g terminated/stopped/move to background

Tsh functionalities and specifications (3/7)

- **Catch Signals**
  - Typing CTRL+C should cause a SIGINT signal to be sent to the current foreground job (i.e. the foreground process and its descendent processes), which is to terminate these processes.
  - Typing CTRL+Z should cause a SIGTSTP signal to be sent to the current foreground job , which is to stop these processes.



Tsh functionalities and specifications (4/7)

- **Catch Signals**

Hints:

1. you need implement handlers for SIGINT, SIGTSTP to catch the SIGINT/SIGTSTP signals
   - implement: `sigint_handler, sigtstp_handler`.
   - default action for SIGINT/SIGTSTP is to terminate/stop its own process (tsh), but here you need to terminate/stop the foreground processes.

2. to send signals to some process, use **kill()**

   int kill(pid_t pid, int sig); //pass -pid to send sig to the process group whose GID==pid

3. create a separate process group for the child process, otherwise all the children processes will be in the same group with "tsh".

   call "setpgid(0,0)" between fork() and execve() in the child process.

   setpgid(pid_t pid, pid_t pgid) // sets the PGID of the process specified by pid to pgid

Tsh functionalities and specifications (5/7)

- Built-in Commands
  - **quit**: terminates "tsh"
  - **jobs**: list the existing (running/stopped) jobs
  - **bg** <job>: restart <job> by sending it SIGCONT signal, then run in background
  - **fg** <job>: restart <job> by sending it SIGCONT signal, then run in foreground

    <job> can be process id, or job id (with a "J" prefix, e.g. bg J1)

Tsh functionalities and specifications (6/7)

- Tsh should reap all of its zombie children.
  - A child process terminated but not reaped becomes a zombie process
  - So tsh should keep records of all its existing jobs and reap them when they terminates
  - Besides, if any job terminates/stops because of signals (e.g. SIGINT/SIGTSTP), the tsh should recognize the event and print a message with JobID, PID and signal number which terminates/stops the job.

Tsh functionalities and specifications (7/7)

- Tsh should reap all of its zombie children.

Hints:

1. the global var “jobs” is used to keep the list of jobs, you can use **addjob**(), **deletejob**() to update this “jobs”.
2. SIGCHLD is a signal which is sent to a process when any of its children terminated/stopped
   You can catch SIGCHLD by implement : `sigchld_handler()`
3. You can put a **waitpid**() in sigchld_handler() to reap terminated processes.
   `pid_t waitpid(pid_t pid, int* status, int options)`
   if pid==-1, means wait for any child
   options:
   WUNTRACED: returns also when the process is stopped. (Otherwise, only return for terminated)
   WNOHANG: returns 0 immediately is no stopped/terminated child process. (Otherwise, will wait)
4. the **status** output from **waitpid**() can be used to test the exit status/signals
   e.g. WIFSTOPPED(status), WIFEXITED(status), WIFSIGNALED(status), …

Other hints

- In eval(), the parent must block SIGCHLD signals before it forks the child, and then unblock these signals, after it calls addjob (to add job in the "jobs").

  Also remember to unblock SIGCHLD signals for the child before it calls execve().

in Parent:	in Child:

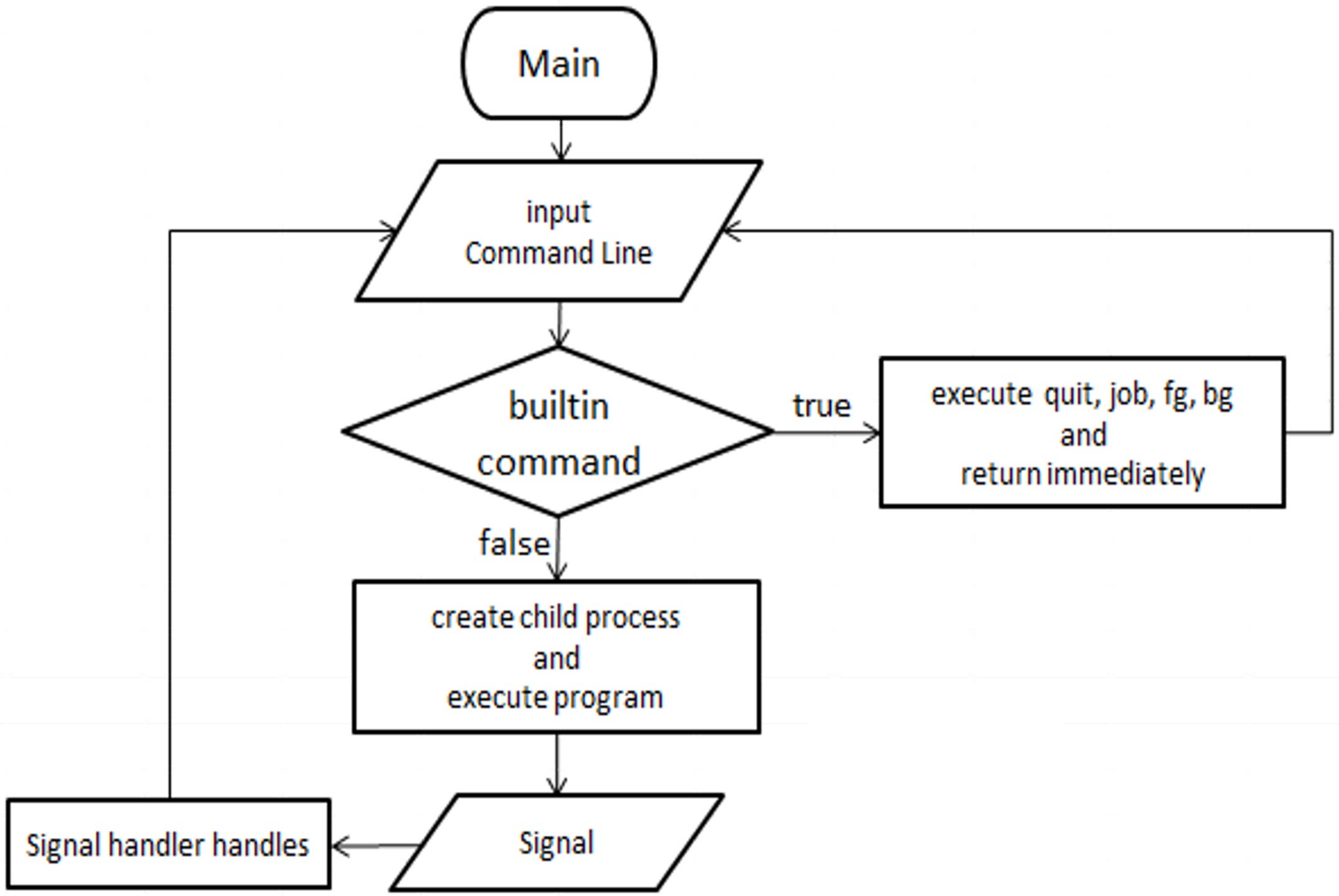
//block SIGCHLD
fork()
…
addjob()
//unblock SIGCHLD
	//block SIGCHLD
	fork()
	…
	//unblock SIGCHLD
	execve()

- This is to avoid the race condition where the child is reaped by sigchld_handler (and thus removed from the job list) *before* the parent calls addjob()
- related function: **sigprocmask**(), **sigemptyset**(), **sigaddset**()..

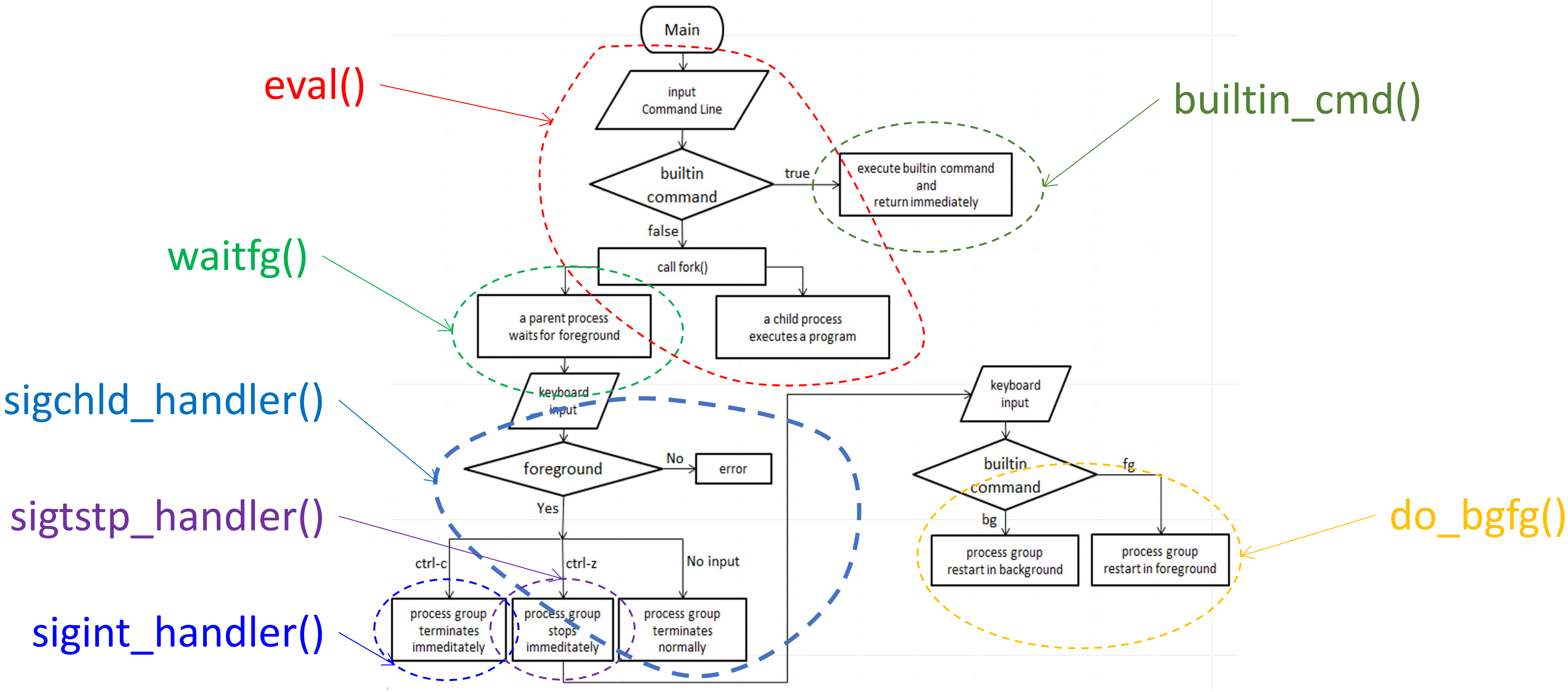
Introduction to Shell Lab

- Implementation of a simple Unix shell program (mainly job control): tsh.cc
  - eval: Main routine that parses and interprets the command line. [70 lines]
  - builtin_cmd: Recognizes and interprets the built-in commands: quit, fg, bg, and jobs. [25 lines]
  - do_bgfg: Implements the bg and fg built-in commands. [50 lines]
  - waitfg: Waits for a foreground job to complete. [20 lines]
  - sigchld_handler: Catches SIGCHILD signals. 80 lines]
  - sigint_handler: Catches SIGINT (ctrl-c) signals. [15 lines]
  - sigtstp_handler: Catches SIGTSTP (ctrl-z) signals. [15 lines]

Logical Flowchart of TSH (1/2)



Logical Flowchart of TSH (2/2)



* Figure 8.24 in the textbook is a good reference to start the shell lab

Example Codes in the Textbook

- Figure 8.24

code/ecf/shellex.c

```c
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

code/ecf/shellex.c

Figure 8.24 eval evaluates the shell command line.