

System Programming Lab #2

2019-03-26

sp-tas

realloc() functionality

7.22.3.5 The realloc function

Synopsis

```
1      #include <stdlib.h>
      void *realloc(void *ptr, size_t size);
```

Description

- 2 The **realloc** function deallocates the old object pointed to by **ptr** and returns a pointer to a new object that has the size specified by **size**. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.
- 3 If **ptr** is a null pointer, the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if **ptr** does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

Returns

- 4 The **realloc** function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

```
// book-keeping
n_realloc++;
n_allocb += size;

i = find(list, ptr);
if (i != NULL) n_freeb += i->size;
dealloc(list, ptr);
alloc(list, res, size);
```

Lab Assignment #2 – Shell Lab

- Download skeleton code from eTL
 - `shlab.tar`
- Hand In
 - Upload your files **eTL**
 - A zip file should include your implementation (`tsh.c`) and a report
- PLEASE, **READ** the Hand-out!!!
 - Hints section provides helpful information to implement your shell
- Assigned: Mar. 26
- Deadline: Apr. 8, 11:59:59 PM
- Lab #4 (4/2) will be Q&A session

Shell Lab

- To become more familiar with the concepts of process control and signaling
- Writing a simple Unix shell program that supports job control

Shell Lab

- There's a lot of starter code
 - Look over it so you don't needlessly repeat work
- Don't be afraid to write your own helper functions; this is not a simple assignment
- SIGCHLD handler may have to reap multiple children per call
- Try actually using your shell and seeing if/where it fails
 - Can be easier than looking at the driver output

You will implement

```
/* Here are the functions that you will implement */  
void eval(char *cmdline);  
int builtin_cmd(char **argv);  
void do_bgfg(char **argv);  
void waitfg(pid_t pid);  
  
void sigchld_handler(int sig);  
void sigtstp_handler(int sig);  
void sigint_handler(int sig);
```

- 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 SIGCHLD signals. 80 lines]
- sigint_handler: Catches SIGINT (ctrl-c) signals. [15 lines]
- sigtstp_handler: Catches SIGTSTP (ctrl-z) signals. [15 lines]

Shell Lab

- Read man pages. You may find the following functions helpful:
 - `sigemptyset()`
 - `sigaddset()`
 - **`sigprocmask()`**
 - `sigsuspend()`
 - `waitpid()`
 - `open()`
 - `dup2()`
 - `setpgid()`
 - `kill()`

In `eval`, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list by calling `addjob`

Shell Lab Testing

- Run your shell
 - This is the *fun* part!
- tshref
 - How should the shell behave?
- runtrace
 - Each trace tests one feature.

Let's start the fun part!

```
# tar xvf shlab.tar
```

```
root@sp3:/home/ta/hkim/shlab/lab4-demo# ls
shlab.tar
root@sp3:/home/ta/hkim/shlab/lab4-demo# tar xvf shlab.tar
shlab/
shlab/trace04.txt
shlab/trace14.txt
shlab/trace15.txt
shlab/trace12.txt
shlab/trace02.txt
shlab/sdriver.pl
shlab/trace10.txt
shlab/tshref
shlab/myspin.c
shlab/trace16.txt
shlab/README
shlab/trace06.txt
shlab/trace05.txt
shlab/mysplit.c
shlab/Makefile
shlab/trace08.txt
shlab/myint.c
shlab/trace11.txt
shlab/tsh.c
shlab/trace13.txt
shlab/trace03.txt
shlab/trace09.txt
shlab/trace01.txt
shlab/tshref.out
shlab/mystop.c
shlab/trace07.txt
root@sp3:/home/ta/hkim/shlab/lab4-demo#
```

After decompression

```
root@sp3:/home/ta/hkim/shlab/lab4-demo# ls -ahil
total 92K
419840 drwxr-xr-x 3 root root 4.0K Mar 25 20:50 .
407219 drwxr-xr-x 8 root root 4.0K Mar 25 20:49 ..
419732 drwxr-xr-x 2 root root 4.0K Mar 25 20:33 shlab
419760 -rw-r--r-- 1 root root 80K Mar 25 20:34 shlab.tar
root@sp3:/home/ta/hkim/shlab/lab4-demo# cd shlab
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# ls -ahil
total 144K
419732 drwxr-xr-x 2 root root 4.0K Mar 25 20:33 .
419840 drwxr-xr-x 3 root root 4.0K Mar 25 20:50 ..
419851 -rw-r--r-- 1 root root 2.3K Apr 3 2018 Makefile
419853 -rw-r--r-- 1 root root 618 Apr 3 2018 myint.c
419845 -rw-r--r-- 1 root root 418 Apr 3 2018 myspin.c
419850 -rw-r--r-- 1 root root 622 Apr 3 2018 mysplit.c
419861 -rw-r--r-- 1 root root 624 Apr 3 2018 mystop.c
419847 -rw-r--r-- 1 root root 761 Apr 3 2018 README
419842 -rw-r-xr-x 1 root root 5.1K Apr 3 2018 sdriver.pl
419859 -rw-r--r-- 1 root root 58 Apr 3 2018 trace01.txt
419841 -rw-r--r-- 1 root root 60 Apr 3 2018 trace02.txt
419857 -rw-r--r-- 1 root root 67 Apr 3 2018 trace03.txt
419734 -rw-r--r-- 1 root root 89 Apr 3 2018 trace04.txt
419849 -rw-r--r-- 1 root root 171 Apr 3 2018 trace05.txt
419848 -rw-r--r-- 1 root root 108 Apr 3 2018 trace06.txt
419862 -rw-r--r-- 1 root root 187 Apr 3 2018 trace07.txt
419852 -rw-r--r-- 1 root root 189 Apr 3 2018 trace08.txt
419858 -rw-r--r-- 1 root root 230 Apr 3 2018 trace09.txt
419843 -rw-r--r-- 1 root root 227 Apr 3 2018 trace10.txt
419854 -rw-r--r-- 1 root root 173 Apr 3 2018 trace11.txt
419813 -rw-r--r-- 1 root root 203 Apr 3 2018 trace12.txt
419856 -rw-r--r-- 1 root root 253 Apr 3 2018 trace13.txt
419736 -rw-r--r-- 1 root root 448 Apr 3 2018 trace14.txt
419756 -rw-r--r-- 1 root root 456 Apr 3 2018 trace15.txt
419846 -rw-r--r-- 1 root root 256 Apr 3 2018 trace16.txt
419855 -rw-r--r-- 1 root root 12K Apr 3 2018 tsh.c
419844 -rw-r-xr-x 1 root root 19K Apr 3 2018 tshref
419860 -rw-r--r-- 1 root root 6.0K Apr 3 2018 tshref.out
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab#
```

make

```
419860 -rw-r--r-- 1 root root 6.0K Apr 3 2018 tshref.out
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# make
gcc -Wall -O2 tsh.c -o tsh
gcc -Wall -O2 myspin.c -o myspin
gcc -Wall -O2 mysplit.c -o mysplit
gcc -Wall -O2 mystop.c -o mystop
gcc -Wall -O2 myint.c -o myint
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# ls -ahil
total 208K
419732 drwxr-xr-x 2 root root 4.0K Mar 25 20:52 .
419840 drwxr-xr-x 3 root root 4.0K Mar 25 20:50 ..
419851 -rw-r--r-- 1 root root 2.3K Apr 3 2018 Makefile
419879 -rw-r-xr-x 1 root root 8.8K Mar 25 20:52 myint
419853 -rw-r--r-- 1 root root 618 Apr 3 2018 myint.c
419871 -rw-r-xr-x 1 root root 8.7K Mar 25 20:52 myspin
419845 -rw-r--r-- 1 root root 418 Apr 3 2018 myspin.c
419872 -rw-r-xr-x 1 root root 8.8K Mar 25 20:52 mysplit
419850 -rw-r--r-- 1 root root 622 Apr 3 2018 mysplit.c
419878 -rw-r-xr-x 1 root root 8.8K Mar 25 20:52 mystop
419861 -rw-r--r-- 1 root root 624 Apr 3 2018 mystop.c
419847 -rw-r--r-- 1 root root 761 Apr 3 2018 README
419842 -rw-r-xr-x 1 root root 5.1K Apr 3 2018 sdriver.pl
419859 -rw-r--r-- 1 root root 58 Apr 3 2018 mystop.c
419841 -rw-r--r-- 1 root root 60 Apr 3 2018 trace02.txt
419857 -rw-r--r-- 1 root root 67 Apr 3 2018 trace03.txt
419857 -rw-r--r-- 1 root root 67 Apr 3 2018 trace03.txt
419734 -rw-r--r-- 1 root root 89 Apr 3 2018 trace04.txt
419849 -rw-r--r-- 1 root root 171 Apr 3 2018 trace05.txt
419848 -rw-r--r-- 1 root root 108 Apr 3 2018 trace06.txt
419862 -rw-r--r-- 1 root root 187 Apr 3 2018 trace07.txt
419852 -rw-r--r-- 1 root root 189 Apr 3 2018 trace08.txt
419858 -rw-r--r-- 1 root root 230 Apr 3 2018 trace09.txt
419843 -rw-r--r-- 1 root root 227 Apr 3 2018 trace10.txt
419854 -rw-r--r-- 1 root root 173 Apr 3 2018 trace11.txt
419813 -rw-r--r-- 1 root root 203 Apr 3 2018 trace12.txt
419856 -rw-r--r-- 1 root root 253 Apr 3 2018 trace13.txt
419736 -rw-r--r-- 1 root root 448 Apr 3 2018 trace14.txt
419756 -rw-r--r-- 1 root root 456 Apr 3 2018 trace15.txt
419846 -rw-r--r-- 1 root root 256 Apr 3 2018 trace16.txt
419855 -rw-r--r-- 1 root root 12K Apr 3 2018 tsh.c
419844 -rw-r-xr-x 1 root root 19K Apr 3 2018 tshref
419860 -rw-r--r-- 1 root root 6.0K Apr 3 2018 tshref.out
```

Compare your shell with a reference solution (tshref)

```
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# ./tsh
tsh> quit
tsh>
```

```
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# ./tshref
tsh> quit
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab#
```

Let's start the fun part!

trace08.txt

```
1 #
2 # trace08.txt - Forward SIGTSTP only to foreground job.
3 #
4 /bin/echo -e tsh> ./myspin 4 \046
5 ./myspin 4 &
6
7 /bin/echo -e tsh> ./myspin 5
8 ./myspin 5
9
10 SLEEP 2
11 TSTP
12
13 /bin/echo tsh> jobs
14 jobs
15
```

Your Output

make test{NN}

make test08

```
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab#
```

Reference Output
- the solution

make rtest{NN}

make rtest08

```
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab# make rtest08
./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (3829) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3831) stopped by signal 20
tsh> jobs
[1] (3829) Running ./myspin 4 &
[2] (3831) Stopped ./myspin 5
root@sp3:/home/ta/hkim/shlab/lab4-demo/shlab#
```

Let's start the fun part!

trace11.txt

```
1 #
2 # trace11.txt - Forward SIGINT to every process in foreground process group
3 #
4 /bin/echo -e tsh> ./mysplit 4
5 ./mysplit 4
6
7 SLEEP 2
8 INT
9
10 /bin/echo tsh> /bin/ps a
11 /bin/ps a
12
13
```

```
./sdriver.pl -t trace11.txt -s ./tsh -a -p
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (26298) terminated by signal 2
tsh> /bin/ps a
  PID TTY          STAT       TIME COMMAND
25181 pts/3        S           0:00 -usr/local/bin/tcsh -i
26239 pts/3        S           0:00 make tshrefout
26240 pts/3        S           0:00 /bin/sh -c make tests > tshref.out 2>&1
26241 pts/3        S           0:00 make tests
26295 pts/3        S           0:00 perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p
26296 pts/3        S           0:00 ./tsh -p
26301 pts/3        R           0:00 /bin/ps a
```

- The output of the `/bin/ps` commands in `trace11.txt`, `trace12.txt`, and `trace13.txt` will be different from run to run. However, the running states of any `mysplit` processes in the output of the `/bin/ps` command should be identical.

Checking Your Work

- Reference Solution
 - The linux executable `tshref` is the reference solution for the shell
 - Your shell should emit output that is *identical* to the reference solution
- Shell driver
 - The `sdriver.pl` program
 - executes a shell as a child process,
 - sends it commands and signals as directed by a trace file,
 - and captures and displays the output from the shell

Your solution shell will be tested for correctness on a Linux machine, using the same shell driver and trace files that were included in your lab directory. Your shell should produce **identical** output on these traces as the reference shell, with only two exceptions:

- The PIDs can (and will) be different.
- The output of the `/bin/ps` commands in `trace11.txt`, `trace12.txt`, and `trace13.txt` will be different from run to run. However, the running states of any `mysplit` processes in the output of the `/bin/ps` command should be identical.

Guide to start your implementation

```
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return. Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
 */
void eval(char *cmdline)
{
    return;
}
```

0. parse & check cmd
1. block signals
2. create child process
3. do the job

3.1 <child process>

- 1) Unblock signal
- 2) Get new process group ID
- 3) Load & run new program

3.2 <parent process>

- 1) Addjob
- 2) Unblock signal
- 3) (if bg) print log message

```
*****
 * Signal handlers
*****

/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 * a child job terminates (becomes a zombie), or stops because it
 * received a SIGSTOP or SIGTSTP signal. The handler reaps all
 * available zombie children, but doesn't wait for any other
 * currently running children to terminate.
 */
void sigchld_handler(int sig)
{
    return;
}

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 * user types ctrl-c at the keyboard. Catch it and send it along
 * to the foreground job.
 */
void sigint_handler(int sig)
{
    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 * the user types ctrl-z at the keyboard. Catch it and suspend the
 * foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    return;
}
```

4. Implement signal handler

Errors from last semester

```
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground process
#
tsh> ./myspin 4 &
[1] (26274) ./myspin 4 &
tsh> ./myspin 5
Job [2] (26276) stopped by signal 20
tsh> jobs
[1] (26274) Running ./myspin 4 &
[2] (26276) Stopped ./myspin 5
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
```

```
root@sysprog:/home/exetest/shlab_tmp# make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (14749) ./myspin 4 &
tsh> ./myspin 5
Job [2] (14751) stopped by signal 20
tsh> jobs
[1] (14749) Running ./myspin 4 &
[2] (14751) Stopped ./myspin 5
tsh> bg %2
[2] (14751) ./myspin 5
tsh> jobs
[1] (14749) Running ./myspin 4 &
[2] (14751) Running ./myspin 5
root@sysprog:/home/exetest/shlab_tmp#
```

```
root@sysprog:/home/exetest/shlab_tmp# make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (14760) ./myspin 4 &
tsh> ./myspin 5
Job [2] (14762) stopped by signal 20
tsh> jobs
[1] (14760) Running ./myspin 4 &
[2] (14762) Stopped ./myspin 5
tsh> bg %2
```

Questions from last semester

- verbose 옵션 대응까지 과제의 범위 안에 있는지 궁금합니다
 - 채점 시에 verbose 옵션 구현 여부 체크 하지 않습니다
- nonlocaljump 사용하여 구현해도 괜찮은가요?
 - Jump를 사용하지 않고 구현하시기 바랍니다
- 과제 스펙 혹은 test 들에 나오지 않은 잘못된 입력은 들어오지 않는다고 가정해도 되나요? 과제 스펙과 test 에 주어진 형태의 입력들만 처리하면 될까요?
 - 배포해드린 trace 파일로 채점 진행 예정입니다
- 채점 기준의 check the return value of every system call (5pt) 관련 sigemptyset, sigaddset, sigprocmask, kill 등의 함수들의 return value 도 모두 체크해줘야 하나요?
 - 네 체크하시기 바랍니다.
- tshref/ref 내부에서 ./myspin: Command not found 에러
 - make 실행하셔서 myspin / myint / mysplit / mystop 실행파일 생성하신 후 실행하시기 바랍니다.

Exceptional Control Flow

Exceptional Control Flow

- Up to now: two mechanisms for changing control flow:

- Jumps and branches
- Call and 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”

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
 - Indicated by setting the processor's interrupt pin
 - Handler returns to “next” instruction

- Examples:
 - I/O interrupts
 - hitting Ctrl-C at the keyboard
 - arrival of a packet from a network
 - arrival of data from a disk
 - Hard reset interrupt
 - hitting the reset button
 - Soft reset interrupt
 - hitting Ctrl-Alt-Delete on a PC

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
 - **Traps**
 - Intentional
 - Examples: **system calls**, breakpoint traps, special instructions
 - Returns control to “next” instruction
 - **Faults**
 - Unintentional but possibly recoverable
 - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
 - Either re-executes faulting (“current”) instruction or aborts
 - **Aborts**
 - unintentional and unrecoverable
 - Examples: parity error, machine check
 - Aborts current program

Processes

- What is a *program*?
 - A bunch of data and instructions stored in an executable binary file
 - Written according to a specification that tells users what it is supposed to do
 - Stateless since binary file is static

Processes

- Definition: A *process* is an instance of a running program.
- Process provides each program with two key abstractions:
 - Logical control flow
 - Each program seems to have exclusive use of the CPU
 - Private virtual address space
 - Each program seems to have exclusive use of main memory
 - Gives the running program a *state*
- How are these Illusions maintained?
 - Process executions interleaved (multitasking) or run on separate cores
 - Address spaces managed by virtual memory system
 - Just know that this exists for now; we'll talk about it soon

Processes

- Four basic States
 - Running
 - Executing instructions on the CPU
 - Number bounded by number of CPU cores
 - Runnable
 - Waiting to be running
 - Blocked
 - Waiting for an event, maybe input from STDIN
 - Not runnable
 - Zombie
 - Terminated, not yet reaped

Processes

- Four basic process control function families:
 - `fork()`
 - `exec()`
 - And other variants such as `execve()`
 - `exit()`
 - `wait()`
 - And variants like `waitpid()`
- Standard on all UNIX-based systems
- Don't be confused:
Fork(), Exit(), Wait() are all wrappers provided by CS:APP

Processes

■ `int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- OS creates an exact duplicate of parent's state:
 - Virtual address space (memory), including heap and stack
 - Registers, except for the return value (%eax/%rax)
 - File descriptors but files are shared
- **Result → Equal but separate state**
- Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Processes

■ `int fork(void)`

- returns 0 to the child process
- returns child's **pid** (process id) to the parent process
- Usually used like:

```
pid_t pid = fork();

if (pid == 0) {
    // pid is 0 so we can detect child
    printf("hello from child\n");
}

else {
    // pid = child's assigned pid
    printf("hello from parent\n");
}
```

Processes

■ `int exec()`

- Replaces the current process's state and context
 - But keeps PID, open files, and signal context
- Provides a way to load and run **another** program
 - Replaces the current running memory image with that of new program
 - Set up stack with arguments and environment variables
 - Start execution at the entry point
- Never returns on successful execution
- The newly loaded program's perspective: as if the previous program has not been run before
- More useful variant is **`int execve()`**
- More information? `man 3 exec`

Processes

■ `void exit(int status)`

- Normally return with status 0 (other numbers indicate an error)
- Terminates the current process
- OS frees resources such as heap memory and open file descriptors and so on...
- Reduce to a zombie state
 - Must wait to be reaped by the parent process (or the init process if the parent died)
 - Signal is sent to the parent process notifying of death
 - Reaper can inspect the exit status

Processes

- `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
 - When `wait` returns a `pid > 0`, child process has been reaped
 - All child resources freed
 - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated
 - More useful variant is `int waitpid()`
 - For details: `man 2 wait`

Process Examples

```
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?

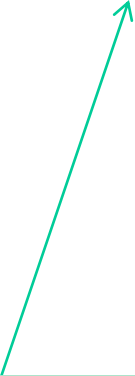
Process Examples

```
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 til the child has terminated.
Parent can inspect exit status of child using 'status'
 - WEXITSTATUS(status)
- Output always:
Child!
Parent!

Process Examples

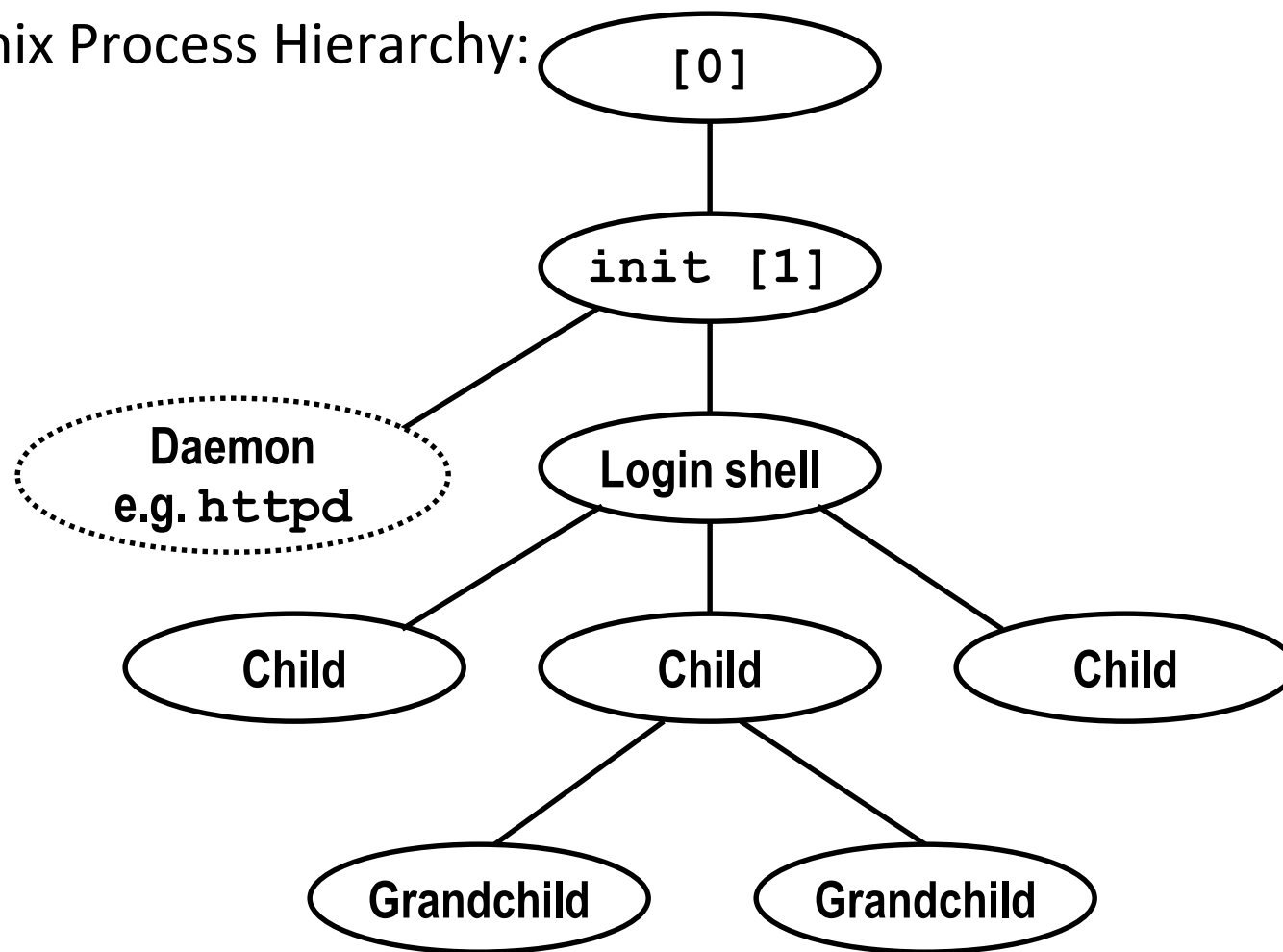
```
int status;  
pid_t child_pid = fork();  
char* argv[] = {"/bin/ls", "-l", NULL};  
char* env[] = {..., NULL};
```

```
if (child_pid == 0){  
    /* only child comes here */  
  
    execve("/bin/ls", argv, env);  
  
    /* will child reach here? */  
}  
else{  
    waitpid(child_pid, &status, 0);  
  
    ... parent continue execution...  
}
```

- An example of something useful.
- Why is the first arg `"/bin/ls"`?
- Will child reach here?

Process Examples

■ Unix Process Hierarchy:



Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
 - akin to exceptions and interrupts (asynchronous)
 - sent from the kernel (sometimes at the request of another process) 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

<i>ID</i>	<i>Name</i>	<i>Default Action</i>	<i>Corresponding Event</i>
2	SIGINT	Terminate	Interrupt (e.g., ctrl-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

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)
 - Another program called the `kill()` function
 - The user used a `kill` utility

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
- 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
- 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*
 - Akin to a hardware exception handler being called in response to an asynchronous interrupt

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
- Blocking signals
 - Sometimes code needs to run through a section that can't be interrupted
 - Implemented with `sigprocmask()`
- Waiting for signals
 - Sometimes, we want to pause execution until we get a specific signal
 - Implemented with `sigsuspend()`
- Can't modify behavior of SIGKILL and SIGSTOP

Signals

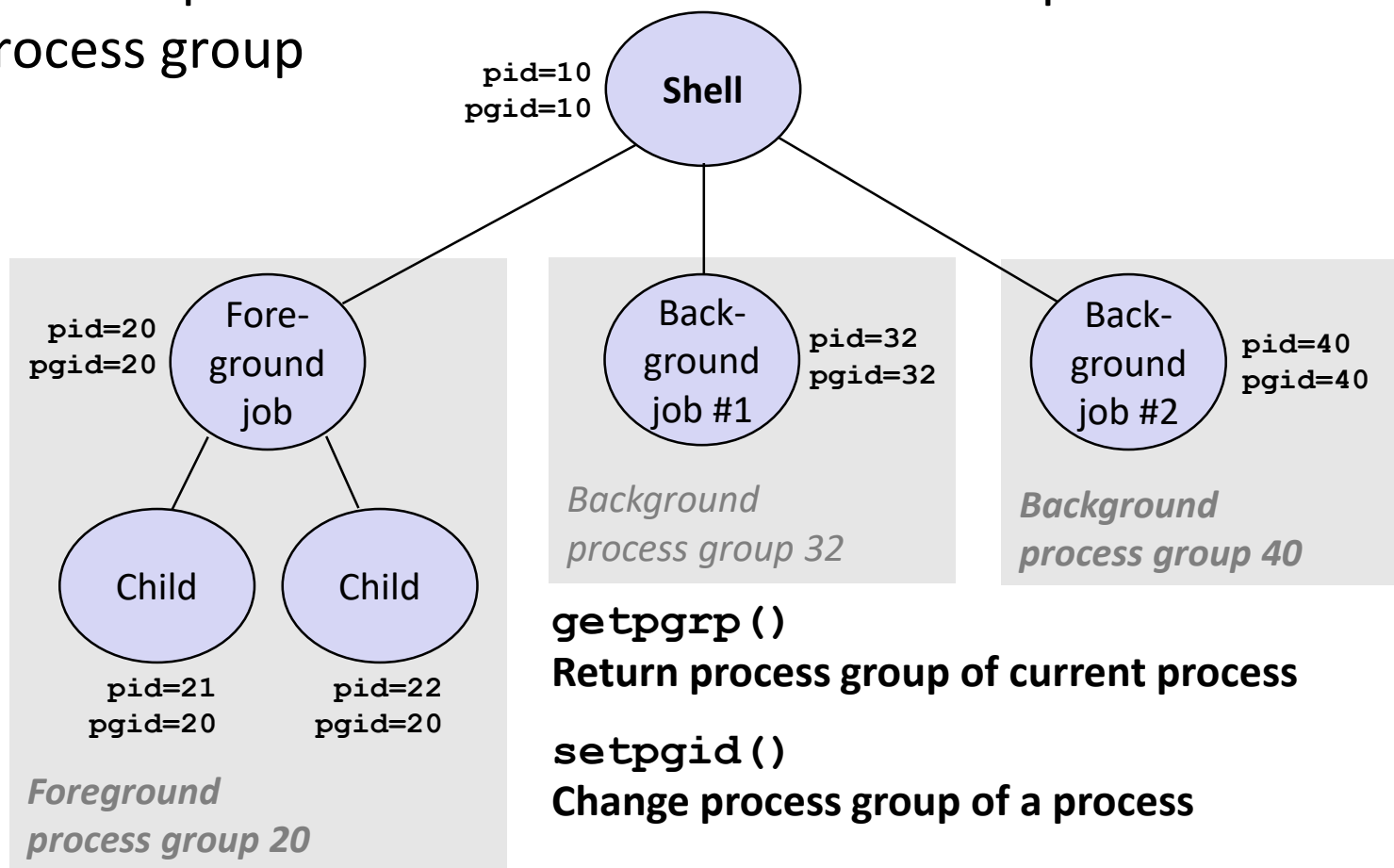
- Signal handlers
 - Can be installed to run when a signal is received
 - The form is `void handler(int signum){ ... }`
 - **Separate** flow of control in the same process
 - Resumes normal flow of control upon returning
 - Can be called **anytime** when the appropriate signal is fired

Signals

- `int sigsuspend(const sigset_t *mask)`
 - Can't use `wait()` twice – use `sigsuspend`!
 - Temporarily replaces the signal mask of the calling process with the mask given
 - Suspends the process until delivery of a signal whose action is to invoke a signal handler or terminate a process
 - Returns if the signal is caught
 - Signal mask restored to the previous state
 - Use `sigaddset()`, `sigemptyset()`, etc. to create the mask

Signal Examples

- Every process belongs to exactly one process group
- Process groups can be used to distribute signals easily
- A forked process becomes a member of the parent's process group



Signal Examples

```
// sigchld handler installed

pid_t child_pid = fork();

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

    execve(.....);
}
else{

    add_job(child_pid);

}
```

```
void sigchld_handler(int signum)
{
    int status;

    pid_t child_pid =
        waitpid(-1, &status, WNOHANG);

    if (WIFEXITED(status))
        remove_job(child_pid);
}
```

- Does `add_job` or `remove_job()` come first?
- Where can we block signals in this code to guarantee correct execution?

Signal Examples

```

// sigchld handler installed
void sigchld_handler(int signum)
{
    int status;

    pid_t child_pid =
        waitpid(-1, &status, WNOHANG);

    if (WIFEXITED(status))
        remove_job(child_pid);
}

pid_t child_pid = fork();

if (child_pid == 0){
    /* child comes here */
    execve(.....);
}
else{
    add_job(child_pid);
}

```

- Does add_job or remove_job() come first?
- Where can we block signals in this code to guarantee correct execution?