SUNG KYUN KWAN UNIVERSITY

Signals

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Multitasking

- Programmer's model of multitasking
  - **fork()** spawns new process
    - Called once, returns twice
  - **exit()** terminates own process
    - Called once, never returns
    - Puts it into "zombie" status
  - **wait()** & **waitpid()** wait for and reap terminated children
  - **execve()** runs new program in existing process
    - Called once, never returns

Signal

- Definition
  - A signal is a small message that notifies a process that an event of some type has occurred in the system
    - Kernel abstraction for exceptions and interrupts
    - Sent from kernel to a process
      - Sometimes at the request of another process
    - Different signals are identified by small integer ID's
    - The only information in a signal is its ID and the fact that it arrived

Signal List

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

- The signals **SIGKILL** & **SIGSTOP** cannot be caught, blocked, or ignored

Signal Concepts (1)

- Sending a signal
  - 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:
    - Generated internally:
      - Divide-by-zero (**SIGFPE**)
      - Termination of a child process (**SIGCHLD**), ...
    - Generated externally:
      - **kill** system call by another process to request signal to the destination process

Signal Concepts (2)

- Receiving a signal
  - 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:
    - Explicitly ignore the signal
    - Execute the default action
    - Catch the signal by invoking signal-handler function
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt

Signal Concepts (3)

- Default actions
  - Abort
    - The process is destroyed
  - Dump
    - The process is destroyed & core dump
  - Ignore
    - The signal is ignored
  - Stop
    - The process is stopped
  - Continue
    - If the process is stopped, it is put into running state

Signal Concepts Example

- Or you can see it from 'man 7 signal'

Standard signals

Linux supports the standard signals listed below. The second column of the table indicates which standard (if any) specified the signal: "P1990" indicates that the signal is described in the original POSIX.1-1990 standard; "P2001" indicates that the signal was added in SUSv2 and POSIX.1-2001.

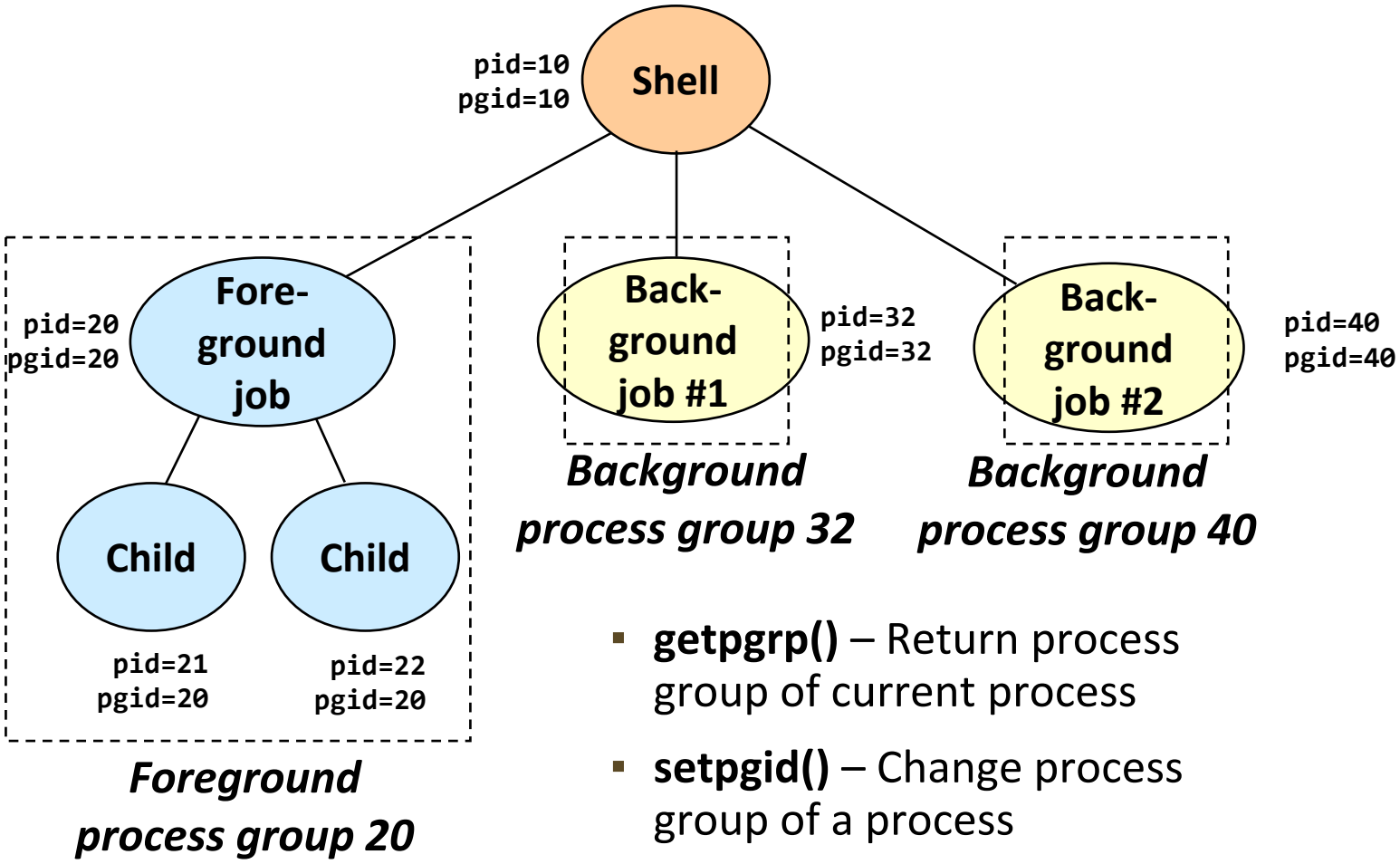
Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from **abort**(3)
SIGALRM	P1990	Term	Timer signal from **alarm**(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see **pipe**(7)
SIGPOLL	P2001	Term	Pollable event (Sys V). Synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also **seccomp**(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see **setrlimit**(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see **setrlimit**(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

Signal Concepts (4)

- Signal semantics
  - A signal is pending if it has been sent but not yet received
    - There can be at most one pending signal of any type
    - Signals are not queued!
  - A pending signal is received at most once
    - Kernel uses a bit vector for indicating pending signals
  - A process can block the receipt of certain signals
    - Blocked signals can be delivered, but will not be received until the signal is unblocked
    - There are signals that can not be blocked by the process
      - e.g. **SIGKILL**, **SIGSTOP**

Process Groups

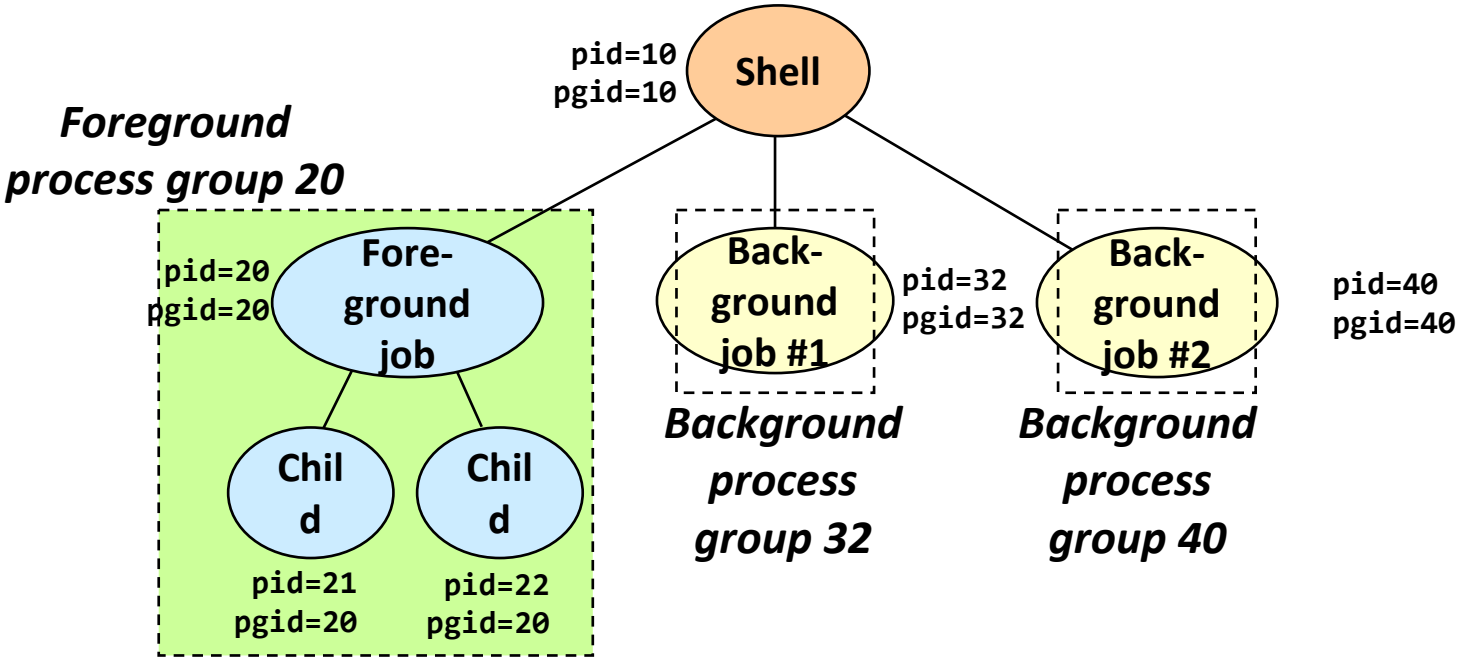
- Every process belongs to exactly one process group



- **getpgrp()** – Return process group of current process
- **setpgid()** – Change process group of a process

Sending Signals (1)

- Sending signals from the keyboard
  - Typing ctrl-c (ctrl-z) sends a SIGINT (SIGTSTP) to every job in the foreground process group
    - SIGINT: default action is to terminate each process
    - SIGTSTP: default action is to stop (suspend) each process



Sending Signals (2)

- **int kill (pid_t pid, int sig)**
  - Can be used to send any signal to any process group or process
    - **pid** > 0, signal **sig** sent to **pid** process
    - **pid** == 0, **sig** is sent to every process in the process group of the current process
    - **pid** == -1, **sig** is sent to every process except for process 1
    - **pid** < -1, **sig** is sent to every process in the process group whose ID is **-pid**
    - **sig** == 0, no signal is sent, but error checking is performed

Example for Sending Signals

```c
int main(void)
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            while(1); /* Child infinite loop */
        }
    }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>

#define N 10
```

```
jaehyun@csl:~/SSE/week7$ ./test1
Killing process 12267
Killing process 12268
Killing process 12269
Killing process 12270
Killing process 12271
Killing process 12272
Killing process 12273
Killing process 12274
Killing process 12275
Killing process 12276
Child 12267 terminated abnormally
Child 12268 terminated abnormally
Child 12269 terminated abnormally
Child 12270 terminated abnormally
Child 12271 terminated abnormally
Child 12272 terminated abnormally
Child 12273 terminated abnormally
Child 12274 terminated abnormally
Child 12275 terminated abnormally
Child 12276 terminated abnormally
jaehyun@csl:~/SSE/week7$
```

Installing Signal Handlers

- **sighandler_t signal (int sig, sighandler_t handler)**
  - **typedef void (*sighandler_t)(int);**
  - The signal function modifies the default action associated with the receipt of signal **sig**

- Different values for handler:
  - **SIG_IGN**: ignore signals of type **sig**
  - **SIG_DFL**: revert to the default action
  - Otherwise, handler is the address of a signal handler
    - Called when process receives signal of type **sig**
    - Referred to as "installing" the signal handler
    - Executing handler is called "catching" or "handling" the signal
    - When the handler executes its return statement, control passes back to the instruction in the control flow of the process that was interrupted by receipt of the signal

Example for Installing Signal Handler

```c
void int_handler(int sig)
{
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(22);
}

int main(void)
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            while(1); /* Child infinite loop */
        }
    }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }

    return 0;
}
```

```
jaehyun@csl:~/SSE/week7$ ./test2
Killing process 12356
Killing process 12357
Killing process 12358
Killing process 12359
Killing process 12360
Killing process 12361
Killing process 12362
Killing process 12363
Killing process 12364
Killing process 12365
Process 12356 received signal 2
Process 12357 received signal 2
Process 12358 received signal 2
Process 12359 received signal 2
Child 12356 terminated with exit status 22
Child 12357 terminated with exit status 22
Child 12358 terminated with exit status 22
Child 12359 terminated with exit status 22
Process 12362 received signal 2
Process 12360 received signal 2
Child 12360 terminated with exit status 22
Child 12362 terminated with exit status 22
Process 12363 received signal 2
Process 12361 received signal 2
Child 12361 terminated with exit status 22
Child 12363 terminated with exit status 22
Process 12365 received signal 2
Process 12364 received signal 2
Child 12365 terminated with exit status 22
Child 12364 terminated with exit status 22
jaehyun@csl:~/SSE/week7$
```

Handling Signals

- Things to remember
  - Pending signals are not queued
    - For each signal type, just have signal bit indicating whether signal is pending
    - Even if multiple processes have sent this signal
  - A newly arrived signal is blocked while the handler of the signal is running
  - Sometimes system calls such as **read()** are not restarted automatically after they are interrupted by the delivery of a signal
    - They return prematurely to the calling application with an error condition **(errno == EINTR)**

Example for Handling Signals #1

- Deal with non-queueing signals

```
#define N 10

pid_t pid[N];
int ccount = 0;

void handler (int sig) {
   pid_t id = wait(NULL);
   ccount--;
   printf("Received signal %d from pid %d\n", sig, id);
}

int main(void) {
   int i;
   ccount = N;

   signal (SIGCHLD, handler);
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) {exit(0); /* child */}
   }
   while (ccount > 0) {sleep(5);}

   return 0;
}
```

Example for Handling Signals #2

- Deal with non-queueing signals

```
#define N 10

pid_t pid[N];
int ccount = 0;

void handler (int sig) {
   pid_t id;
   while ((id = waitpid(-1, NULL, WNOHANG)) > 0) {
      ccount--;
      printf("Received signal %d from pid %d\n", sig, id);
   }
}

int main(void) {
   int i;
   ccount = N;

   signal (SIGCHLD, handler);
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) {exit(0); /* child */}
   }
   while (ccount > 0) {sleep (5);}

   return 0;
}
```

options	Description
WNOHANG	return immediately if no child has exited
0	meaning wait for any child process

pid	Description
-1	meaning wait for any child process
0	meaning wait for any child process whose process group ID is equal to that of the calling process
> 0	meaning wait for the child whose process ID is equal to the value of pid

Exercise

- Make queued signal
  - Sender is parent process and receiver is child process
  - The program get the number of signals to send by argument
  - Use **SIGINT, SIGALRM, SIGUSR1** signal
  - Use **alarm()** function

```
ALARM(2)                                                Linux Programmer's Manual

NAME
       alarm - set an alarm clock for delivery of a signal

SYNOPSIS
       #include <unistd.h>

       unsigned int alarm(unsigned int seconds);

DESCRIPTION
       alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds seconds.

       If seconds is zero, any pending alarm is canceled.

       In any event any previously set alarm() is canceled.
```

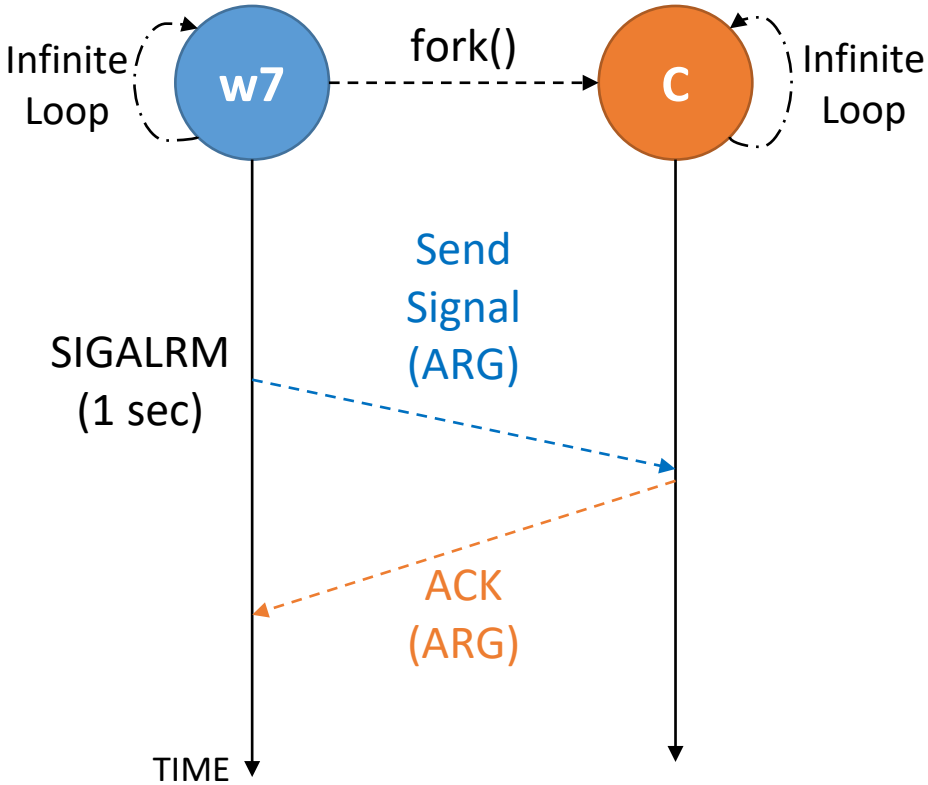
Exercise

- Sender
  - Sender sends signal to receiver and receives acknowledge from receiver
  - Sender checks the number of sending signal and received acknowledge
  - If sender doesn't receive all acks of sending signal, sends back the remaining signals after 1 second
    - Do not use wait() or sleep()
  - If received acks and the number of sending signals are same, send SIGINT to receiver and terminate

Exercise

- Receiver
  - Receiver receives signal and sends acknowledge to sender
  - If receiver receives SIGINT, receiver prints how many signals receive and is terminated

Exercise




```
> ./p7 10
total number of signal(s): 10
sender: total remaining signal(s): 10
receiver: received #1 signal and sending ack
receiver: received #2 signal and sending ack
receiver: received #3 signal and sending ack
sender: total remaining signal(s): 8
receiver: received #4 signal and sending ack
receiver: received #5 signal and sending ack
sender: total remaining signal(s): 6
receiver: received #6 signal and sending ack
receiver: received #7 signal and sending ack
sender: total remaining signal(s): 4
receiver: received #8 signal and sending ack
sender: total remaining signal(s): 3
receiver: received #9 signal and sending ack
sender: total remaining signal(s): 2
receiver: received #10 signal and sending ack
sender: total remaining signal(s): 1
all signals have been sent
receiver: total received signal(s): 10
```

Skeleton code of p7.c

- copy the skeleton code to your directory

$ cp ~swe2024-41_23s/2023s/p7_skeleton.c ./

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int num_send_sig;
int num_recv_sig = 0;
int num_recv_ack = 0;
pid_t pid;
/* DO NOT MODIFY OR ADD CODE ABOVE */

void recv_ack_handler(int sig) {
    num_recv_ack++;
    return;
}

void terminate_handler(int sig) {
    printf("receiver: total received signal(s): %d\n", num_recv_sig);
    exit(0);
}

void sending_handler(int sig) {
    /* your code here */
    alarm(1);
    return;
}

void sending_ack(int sig) {
    /* your code here */
    return;
}
```

```c
int main(int argc, char *argv[]) {
    /* DO NOT MODIFY MAIN */

    num_send_sig = atoi(argv[1]);
    printf("total number of signal(s): %d\n", num_send_sig);

    if ((pid = fork()) == 0) {
        pid = getppid();
        /* In the child process pid, == parent's pid,
           [sending_ack] function is installed as signal [SIGUSR1]'s handler
           [terminate_handler] function is installed as signal [SIGINT]'s handler */
        signal(SIGUSR1, sending_ack);
        signal(SIGINT, terminate_handler);
        while (1)
            ;
    } else {
        /* In the parent process, pid == child's pid,
           [recv_ack_handler] function is installed as signal [SIGUSR1]'s handler
           [sending_handler] function is installed as signal [SIGINT]'s handler */
        signal(SIGUSR1, recv_ack_handler);
        signal(SIGALRM, sending_handler);
        alarm(1);
        while (1)
            ;
    }
    return 0;
}
/* How the program works----------------------------------------------------------------------------|
 1. SIGALARM signal is sent from parent process when alarm(1) is called.                            |
 2. sending_handler function is called                                                              |
 3. sending_handler function should send SIGUSR1s to child while there are remaining signals to be sent |
   3-1. sending_handler should send SIGINT when all signals are sent                                |
   3-2. sending_handler calls alarm(1) and repeats from 2                                           |
 4. sending_ack function is called from child process                                               |
 5. sending_ack function should send SIGUSR1 to parent and recv_ack_handler is called from parent  |
 6. When all signals are sent and received, SIGINT from 3-1 calls terminate_handler               |
 7. done                                                                                            |
-------------------------------------------------------------------------------------------------*/
```

Exercise

- Submit your exercise source code
  - To InUiYeJi Cluster
  - Put your Makefile and *.c files in p7 folder
  - Submit using

  $ ~swe2024-41_23s/bin/submit p7 p7
  - We will compile by using command *make*
    - When compilation fails, your get zero points
    - Compiled binary name should be "*p7*"

- Due 2023/4/12 23:59