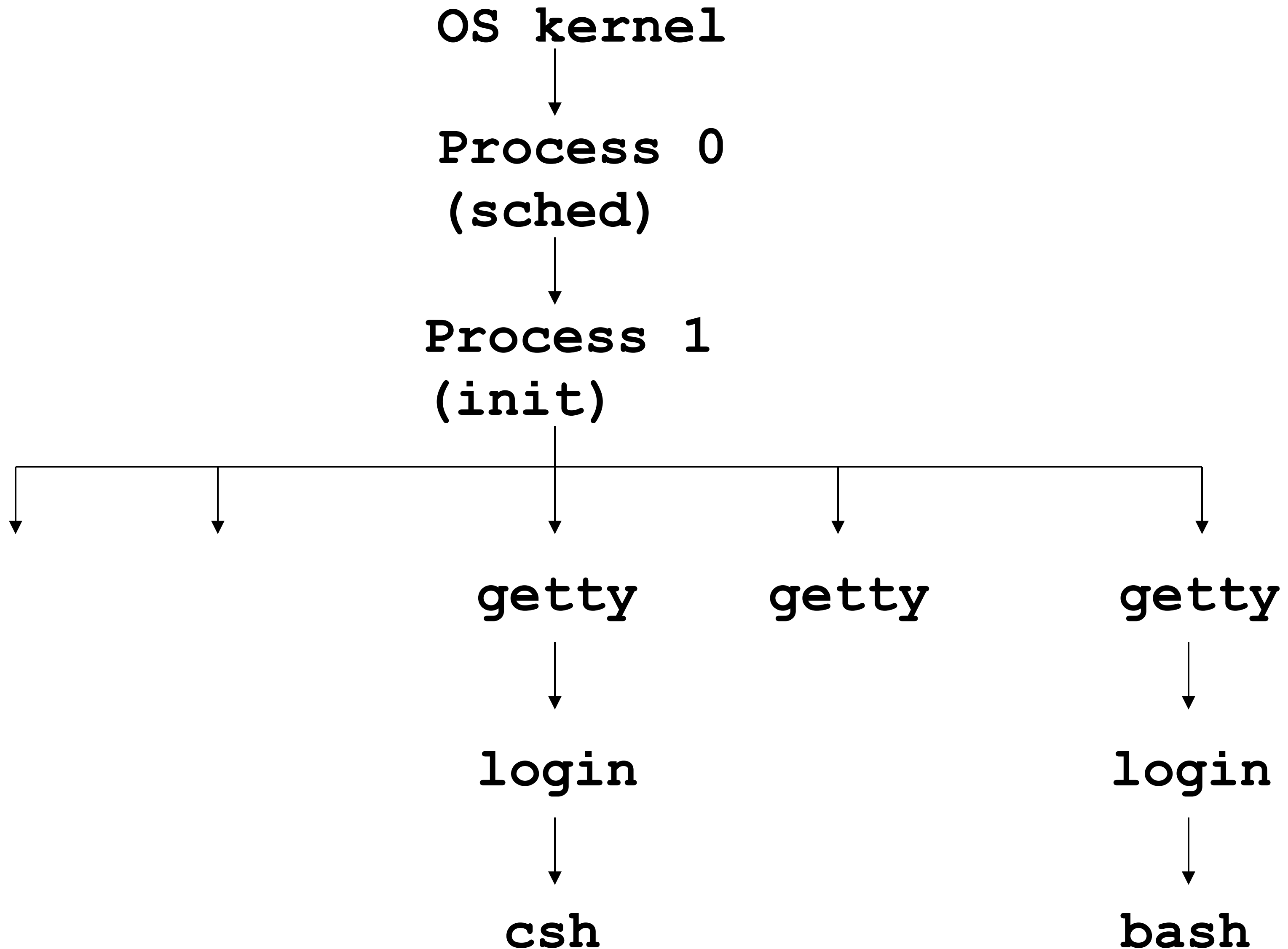
CS240: Programming in C

Lecture 17: Processes, Pipes, and Signals

Processes in UNIX

- UNIX identifies processes via a unique Process ID
  - Each process also knows its parent process ID since each process is created from a parent process.
  - Root process is the 'init' process
- **getpid** and **getppid** functions to return process ID (PID) and parent process ID (PPID)

Unix Start Up Processes



Process ID

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

int main () {

  printf(“I am  process %ld\n”, (long)getpid());
  printf(“My parent id is %ld\n”, (long)getppid());

  return 0;
}
```

Creating Processes

- Fork
  - Creates a new process, called child, by duplicating the calling process called parent
- Exec
  - Replacing process's program with the one inside the exec() call.

fork

```
#include <unistd.h>
pid_t fork(void);
```

- Creates a new process, called child, by duplicating the calling process called parent
- On success, in child it returns 0 and in the parent returns the PID of the child process
- On failure, in parent returns -1 and and *errno* is set appropriately; no child process is created
- Child can always obtain id of the parent with getppid.

Fork Details

- Duplication means:
  - Child gets exact copy of code, stack, file descriptors, heap, global variables, and program counter
  - BUT new pid
- Execution of parent and child:
  - In parallel
  - Parent waits for the child before finishing
  - After fork, scheduler dictates if child starts executing before parent or vice versa

Fork Example

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

int main() {
   pid_t x;
   x = fork();
   if (x < 0) {
        perror("Fork failed ");
        exit(EXIT_FAILURE);
   }
   else if(x == 0) {  /* child */
      printf("I am the child: fork returned %ld\n", (long) x);
      printf("Child and my ID is : %ld\n", (long)getpid());
   }
   else {
      printf("I am the parent: fork returned %ld\n", (long) x);
   }
   return 0;
}
```

exec

```
#include <unistd.h>
int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execle( const char *path, const char *arg  , ...,
char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv [],
char *const envp[] );
```

- Family of functions for replacing process's program with the one inside the exec() call.

Exec example

```
#include <unistd.h>

int main () {

  execl("/bin/ls", "ls", NULL);

  return 0;
}
```

exec vs system

- **`system`**: creates a child process and invokes another shell
  - the return value tells whether the command shell was invoked, but provides no information about the command itself.
- **`exec`**: does not create a child process, but replaces the current process

Process Termination

- A process can terminate voluntary or involuntary
- Voluntary
  - Normal termination: exit(0)
  - Error termination exit(2) or abort()
- Involuntary:
  - Fatal error: divide by 0, segmentation fault
  - Killed by another process kill(procID)

What happens when a process terminates?

- All open files are flushed and closed
- Temporary files are deleted
- Resources are de-allocated
- Parent process is notified via a signal
- Exit status is available to parent via wait()

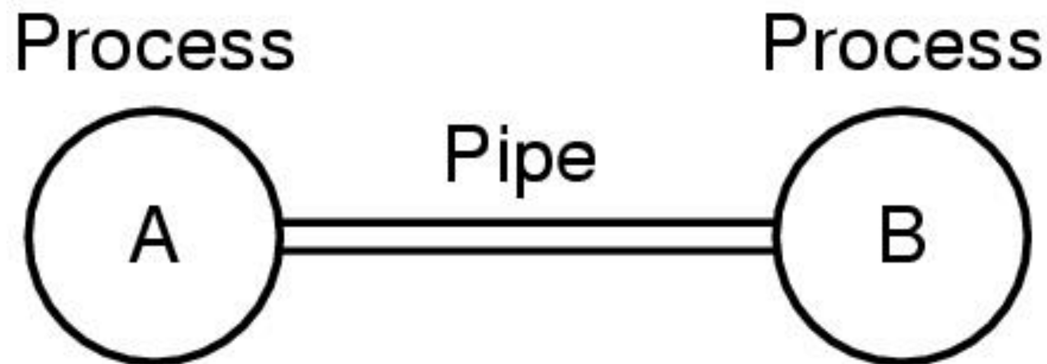
Wait and waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *status, int opts)
```

- **`wait()`**
  - Makes the parent process to wait (block) until some child finishes
  - Returns child's pid and exit status to parent
- **`waitpid()`**
  - Makes the parent to wait (block) for a specific child

Interprocess Communication

- Pipe sets up a communication channel between two (related) processes, usually child - parent



pipe

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- Creates a pipe, it is UNIDIRECTIONAL or half-duplex
- pipefd is used to return two file descriptors referring to the ends of the pipe.
  - **pipefd[0] refers to the read end of the pipe.**
  - **pipefd[1] refers to the write end of the pipe.**
- Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.
- Returns 0 on success and -1 on error

Pipe Example

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#define BUF_SIZE   1024

int main(){
  char child_recv[BUF_SIZE] ;
  char *parent_send = "Hello world!";
  int fd[2];

  pipe(fd);    /* create pipe */
  if (fork() != 0) { /* parent */
   close(fd[0]); /* parent will write */
    printf("Sending to child: %s\n", parent_send);
    write(fd[1], parent_send, strlen(parent_send) + 1) ;
  }
  else { /* child */
    close(fd[1]); /* child will read */
    read(fd[0], child_recv, BUF_SIZE) ;
    printf("Received from parent: %s\n", child_recv) ;
  }
   return 0;
}
```

Specifying how the Pipe is Used

- If a process wants to use the pipe to write should close the read fd
- If a process wants to used to pipe to read should close the write fd
- If both are open in a process, that process can both read and write
- If write is still open at the reading end, the reader does not see EOF because the OS assumes that a write might occur (from the reader)
- If writer overfills the buffer and there is a read open (even if it is the same process writing) the write will block
- Closing ends makes the logic easier and cleaner.

Reading/Writing from a Pipe

- The data is handled in a first-in, first-out (FIFO) order.
- Pipes do not allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.
- Reading or writing pipe data is *atomic* if the size of data written is not greater than PIPE_BUF.
- Once PIPE_BUF bytes have been written, further writes will block until some bytes are read.

Reading/Writing from a One-end Pipe

- If we read from a pipe whose write end has been closed, after all the data has been read, read function returns 0 to indicate the end of file.
- If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns an error with errno set to EPIPE

Signals

- **Signal**: notification from one process (user process or OS) to another process about an event
- **Handler**: code ran in response to a signal
- Handling signals:
  - can be ignored
  - ran the default handler
  - ran the user handler

Asynchronous or synchronous

- **Asynchronous**
  - Poll: ask the OS, did the event took place
  - Handle: tell OS what to do when the event occurs (through the handler)
- **Synchronous**
  - The process that generated the signal blocks till the handler of the signal is executed and returns

Types of signals

- **Interrupts**
  - (SIGINT, Ctrl-C); Environment-triggered (SIGINT, Ctrl-C)
- **Hardware**
  - (SIGSEGV); divide by 0, invalid memory reference
- **Software**
  - (SIGPIPE, SIGALRM). Timeout on network connection, a broken pipe, ...

Generating a signal

```
#include <signal.h>
int kill(pid_t pid, int sig);
int raise(int sig);
```

- **`kill`** can send any signal to any process group or process.
  - If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.
  - If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.
- **`raise`** generates a signal handled by the program that contains the call to raise;
  - In a single-threaded program it is same as **`kill`**

List of signals

- UNIX has a fixed set of signals (Linux has 32 of them)
- **`signal.h`** defines the signals in the OS
- Applications programs can use **`SIGUSR1`** & **`SIGUSR2`** for arbitrary signaling

Signal.h

`SIGABRT`:

- Abnormal termination, such as instigated by the abort function (Abort)

`SIGFPE`:

- Erroneous arithmetic operation, such as divide by 0 or overflow (Floating point exception)

`SIGILL`:

- An 'invalid object program' has been detected. This usually means that there is an illegal instruction in the program (Illegal instruction)

Signal.h cont.

`SIGINT`:

- Interactive attention signal; on interactive systems this is usually generated by typing some ‘break-in’ key at the terminal (Interrupt)

`SIGSEGV`:

- Invalid storage access; most frequently caused by attempting to store some value in an object pointed to by a bad pointer (Segment violation)

`SIGTERM`:

- Termination request made to the program (Terminate)

signal

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

- signal installs a new handler for the supplied signal
  - It returns the previous value of the handler as its result
  - If no such value exists, it returns SIG_ERR and sets errno appropriately

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

- signal is a function pointer to a function that
  - takes as arguments a signal (represented as an int) and a handler
  - returns a function that takes an int and returns void
- The handler is a function pointer to a function that takes an int and returns void.

Example with signal

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

long  prev, i;

void  SIGhandler(int sig) {
  printf("\nGot SIGUSR1. %ld!=%ld\n", i-1, prev);
  exit(0);
}
void  main(void) {
    long  fact;
    signal(SIGUSR1, SIGhandler);
    for (prev = i = 1; ; i++, prev = fact) {
         fact = prev*i;
         if  (fact < 0) raise(SIGUSR1);
         else if (i % 3 == 0)
         printf("     %ld! = %ld\n", i, fact);
    }
}
```

Example program handling two signals

```
static void sig_usr(int signo) {
   if (signo == SIGUSR1)
      printf("received SIGUSR1\n");
   else if (signo == SIGUSR2)
      printf("received SIGUSR2\n");
   else
      printf("received signal %d\n", signo);
   return;
}

int main () {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR2");
    for(;;) pause();
}
```

Example

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

FILE *temp_file;
void leave(int sig);

int main() {
 signal(SIGINT,leave);
 temp_file = fopen("tmp","w");
 for(;;) {  printf("Ready...\n"); getchar(); }
 exit(EXIT_SUCCESS);
}
void leave(int sig) {
   fprintf (temp_file,"\nInterrupted.");
   fclose(temp_file);
   exit(sig);
}
```

Readings and exercises for this lecture

Read man/info pages for all the functions mentioned in the lecture

Code all the examples in the lecture.