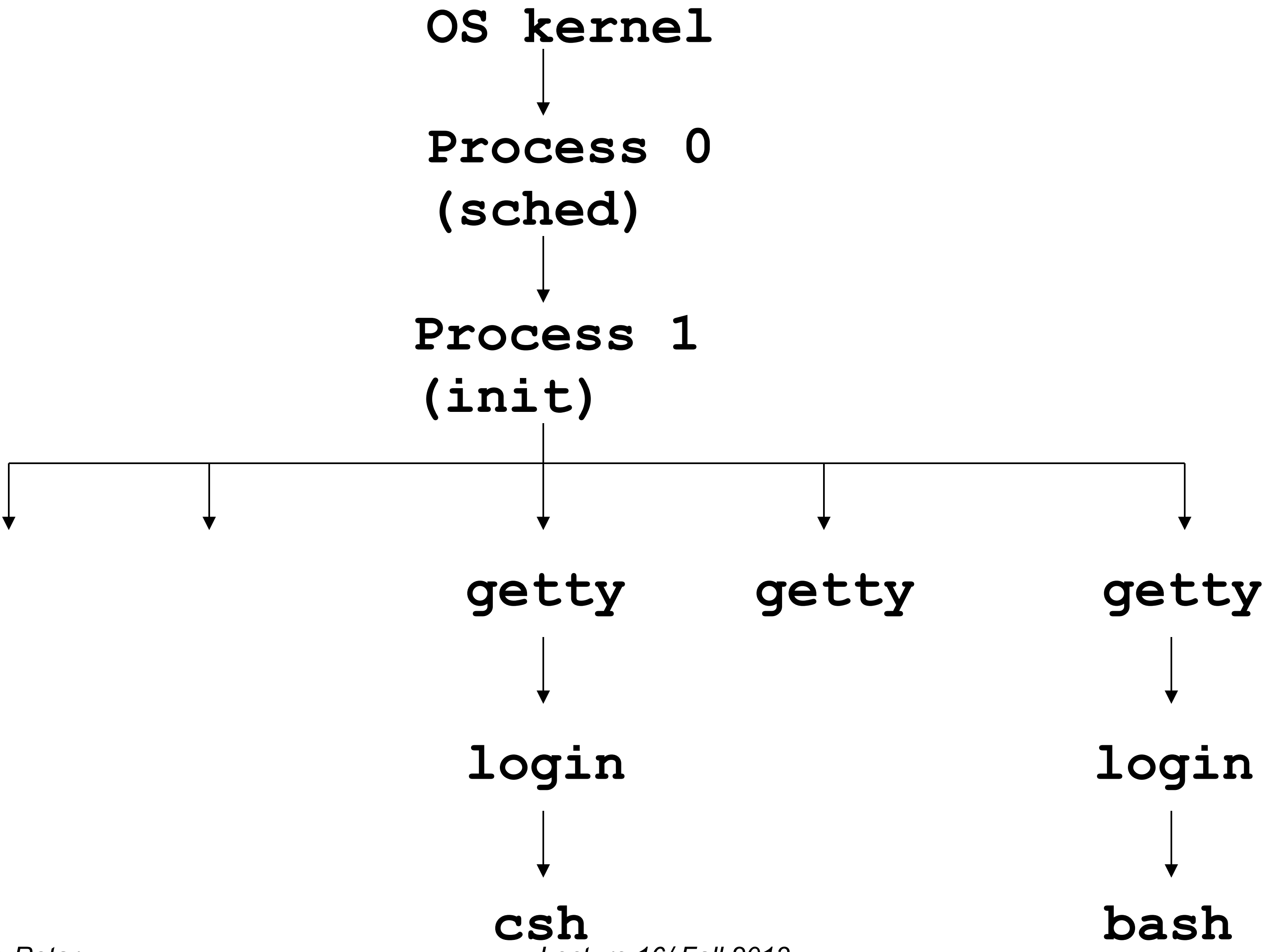
CS240: Programming in C

Lecture 16: Process 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



OS kernel

Process 0 (sched)

Process 1 (init)

getty

getty

getty

login

login

csh

bash

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

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 wait for the child

Fork Example

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

int main() {
   pid_t x;
   x = fork();
   if(x == 0) {
      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;
}
```

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()

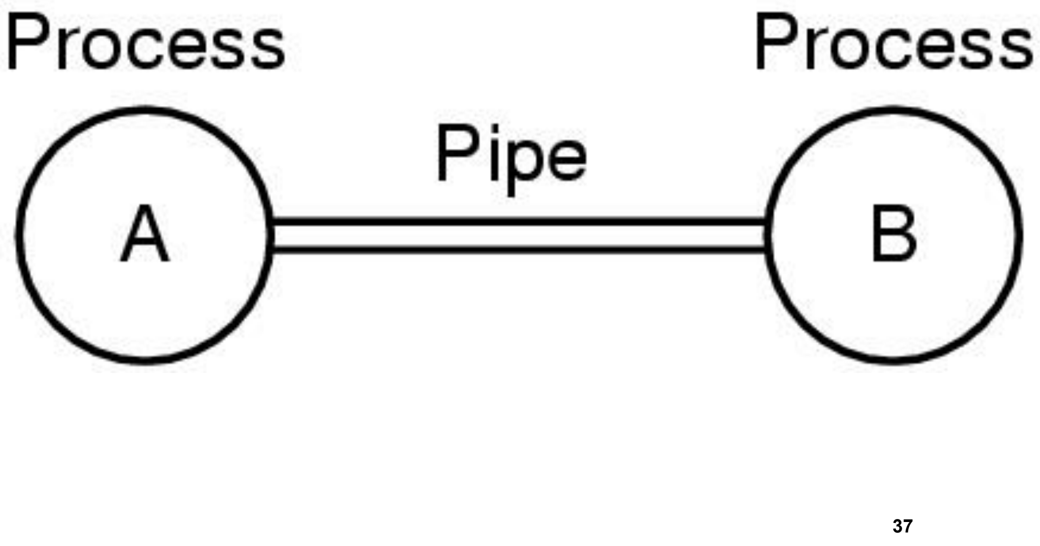
Wait and waitpid

```
#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.



pipe

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

- Creates a pipe
- 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   100

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

  pipe(fd);    /* create pipe */
  if (fork() != 0) { /* parent */
    printf("Sending to child: %s\n", parent_send);
    write(fd[1], parent_send, strlen(parent_send) + 1) ;
  }
  else { /* child */
    read(fd[0], child_recv, 1024) ;
    printf("Received from parent: %s\n", child_recv) ;
  }
   return 0;
}
```

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.