Week 13 Lecture 1

XMUT-NWEN 241 - 2024 T2

Systems Programming

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

Content

- System calls for Process Management

Process management system calls

The following system calls are used for basic process management.

- fork()
- exec()

Defined in unistd.h

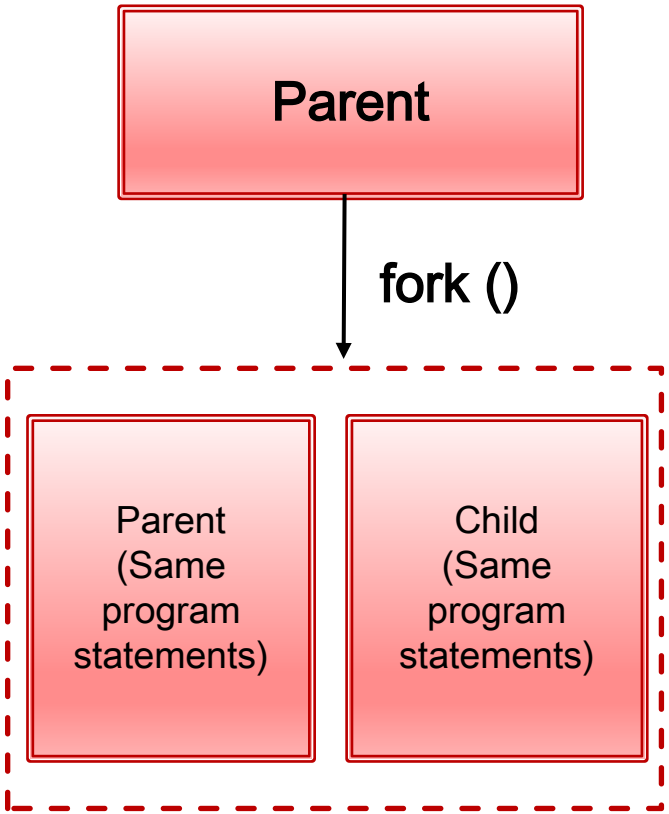
- wait()

Defined in sys/wait.h

- exit()

Defined in stdlib.h

Process creation with fork()



- In Linux all processes are created with the system call **fork()**.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

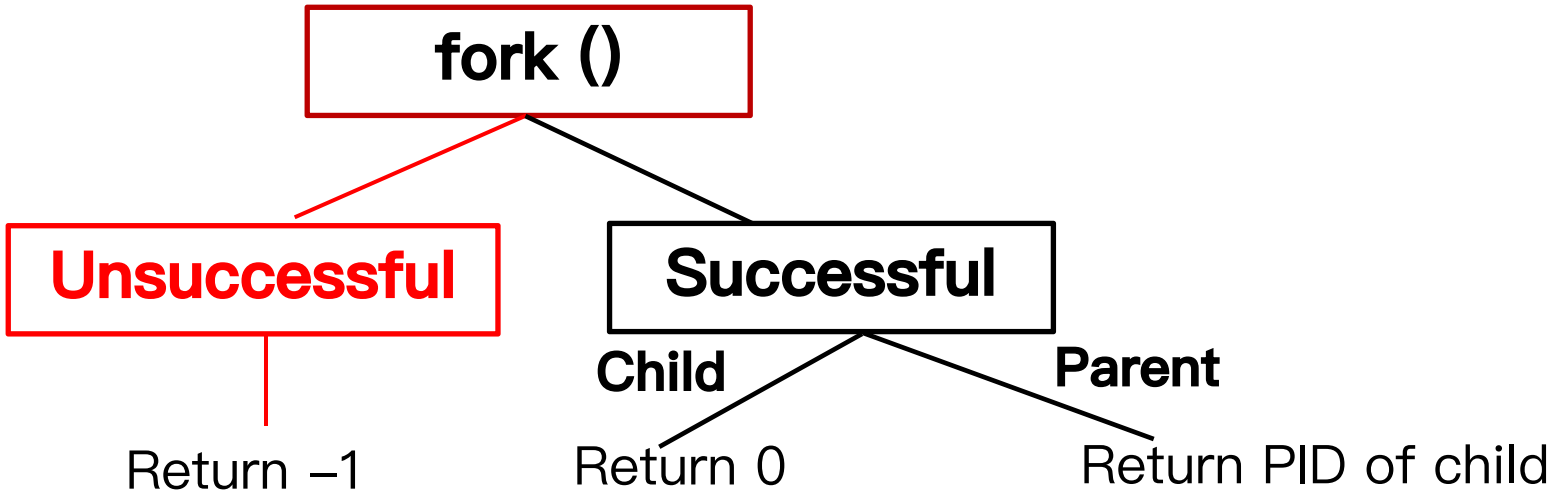
- A process calling **fork()** spawns a new process (child), which is a copy of the calling process.

- After a successful **fork()** call, two copies of the original code will be running.

Process creation with fork()

- After the **fork()**, both processes not only run the same program, but they resume execution as though both had called the system call.

- **fork()** returns an integer value to both parent and child process.



Illustration

Prior to fork() system call:

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

Illustration

After fork() system call:

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 13434

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 0

Illustration

After fork() system call:

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 13434

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 0

In parent, fork() will return PID of child

Illustration

After fork() system call:

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 13434

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

p 0

In parent, fork() will return PID of child
In child, fork() will return 0

Output (if fork() is successful)

```
void main(void)
{
  printf("Before fork\n");
  pid_t p = fork();
  printf("p = %d\n", p);
}
```

```
Before fork
p = 13434
p = 0
```

From parent (and the only process)

From parent

From new child

Order of appearance not predictable

Using the return value

Return value can be used to determine what to do in parent and child

```
void main(void)
{
    printf("Before fork\n");
    pid_t p = fork();
    if(p < 0) {
        /* Failed to fork */
    } else if(p == 0) {
        /* Child process will execute this part */
    } else if(p > 0){
        /* Parent process will execute this part */
    }
}
```

Process ID

- To obtain the process ID of a process:

pid_t getpid(void);

```
void main(void)
{
    pid_t p = fork();
    if(p == 0) { /* Child */
        printf("My PID: %d\n", getpid());
    } else if(p > 0) { /* Parent */
        printf("My PID: %d, child PID: %d\n", getpid(), p);
    }
}
```

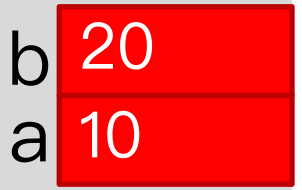
Variables

- After a successful **fork()** call, two copies of the original code will be running
- Parent and child will have their **own** copies of variables
- Variable changes in one process will not affect the variables in the other process

Illustration

Prior to fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```



Illustration

After fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	13434
b	20
a	10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	0
b	20
a	10

Illustration

After fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	13434
b	21
a	10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	0
b	20
a	11

Illustration

After fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	13434
b	21
a	10

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Child */
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	0
b	20
a	11

Illustration

After fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Chil
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	13434
b	21
a	10

```
10 21
11 20
```

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */

                ) { /* Child */

  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	0
b	20
a	11

Illustration

After fork() system call:

```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */
    exit(0);
  } else if(p == 0) { /* Chil
    a++;
  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	13434
b	21
a	10

```
11 20
10 21
```

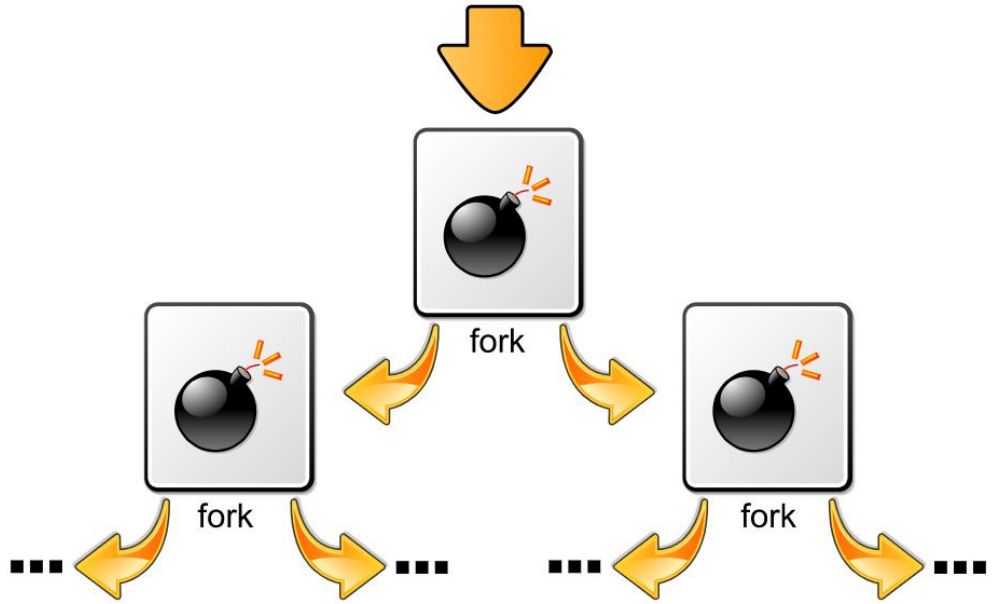
```
void main(void)
{
  int a = 10, b = 20;
  pid_t p = fork();
  if(p < 0) { /* Failed */

                ) { /* Child */

  } else {  /* Parent */
    b++;
  }
  printf("%d %d\n", a, b);
}
```

p	0
b	20
a	11

Fork bomb

What will happen in this code?

```
void main(void)
{
    while(1)
        fork();
}
```



Fork bomb (aka *wabbit* or *rabbit virus*): a form of denial of service attack to Linux based systems

Process creation with exec()

- **exec()** call replaces a current process' image with a new one (i.e. loads a new program within current process)

- Upon success, **exec() does not** return to the caller
  - If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.

- The process id **PID is not changed**, this is because we are not creating a new process, we are just replacing a process with another process

- The new process is executed from the entry point.

Process creation with exec()

- There is no system call specifically by the name **exec()**
- By **exec()** we usually refer to a family of calls:
  - **int execl(char *path, char *arg, ...);**
  - **int execv(char *path, char *argv[]);**
  - **int execle(char *path, char *arg, ...,**
    char *envp[]);
  - **int execve(char *path, char *argv[],**
    char *envp[]);
  - **int execlp(char *file, char *arg, ...);**
  - **int execvp(char *file, char *argv[]);**
- The various options *l*, *v*, *e*, and *p* mean:
  - ***l*** : an argument list,
  - ***v***: an argument vector,
  - ***e***: an environment vector, and
  - ***p***: a search path.

Illustration

int execl(char *path, char *arg, ...);

Path of the executable binary

List of one or more pointers of argument list to the program to be executed. End with NULL pointer

The first argument, by convention, should point to the filename associated with the file being executed.

```
void main(void)
{
  printf("Before exec\n");
  int r = execl("/bin/ls", "ls", NULL);
  printf("r = %d\n", r);
}
```

Illustration

int execl(char *path, char *arg, ...);

Path of the executable binary

List of one or more pointers of argument list to the program to be executed.
End with NULL pointer

```
void main(void)
{
  printf("Before exec\n");
  int r = execl("/bin/ls", "ls", NULL);
  printf("r = %d\n", r);
}
```

Illustration

int execl(char *path, char *arg, ...);

Path of the executable binary

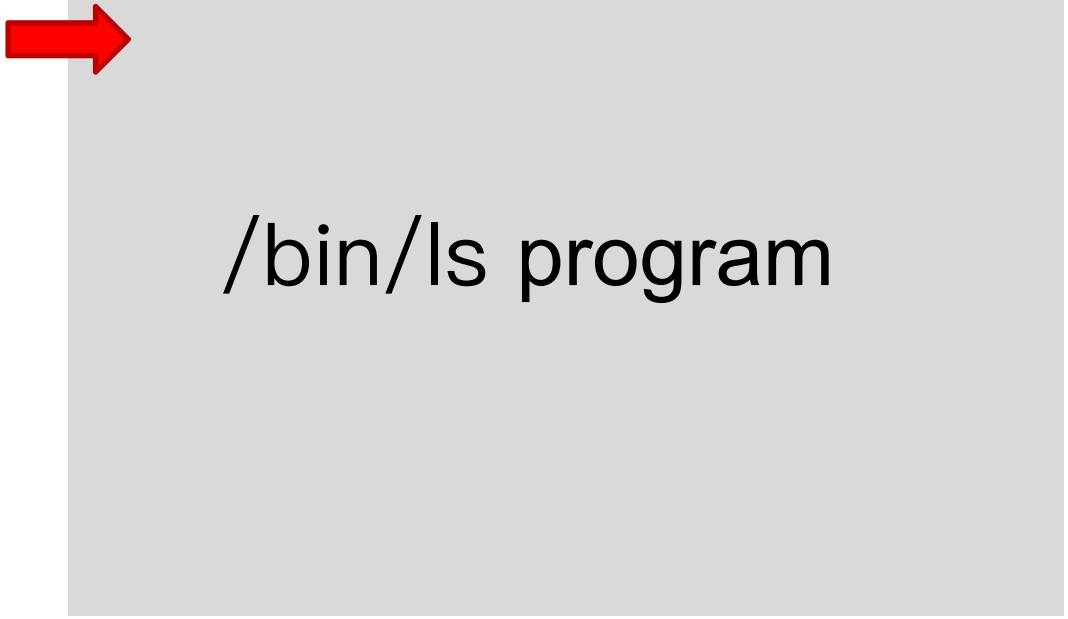
List of one or more pointers of argument list to the program to be executed. End with NULL pointer

```
void main(void)
{
  printf("Before exec\n");
  int r = execl("/bin/ls", "ls", NULL);
  printf("r = %d\n", r);
}
```

Illustration

After exec() system call:

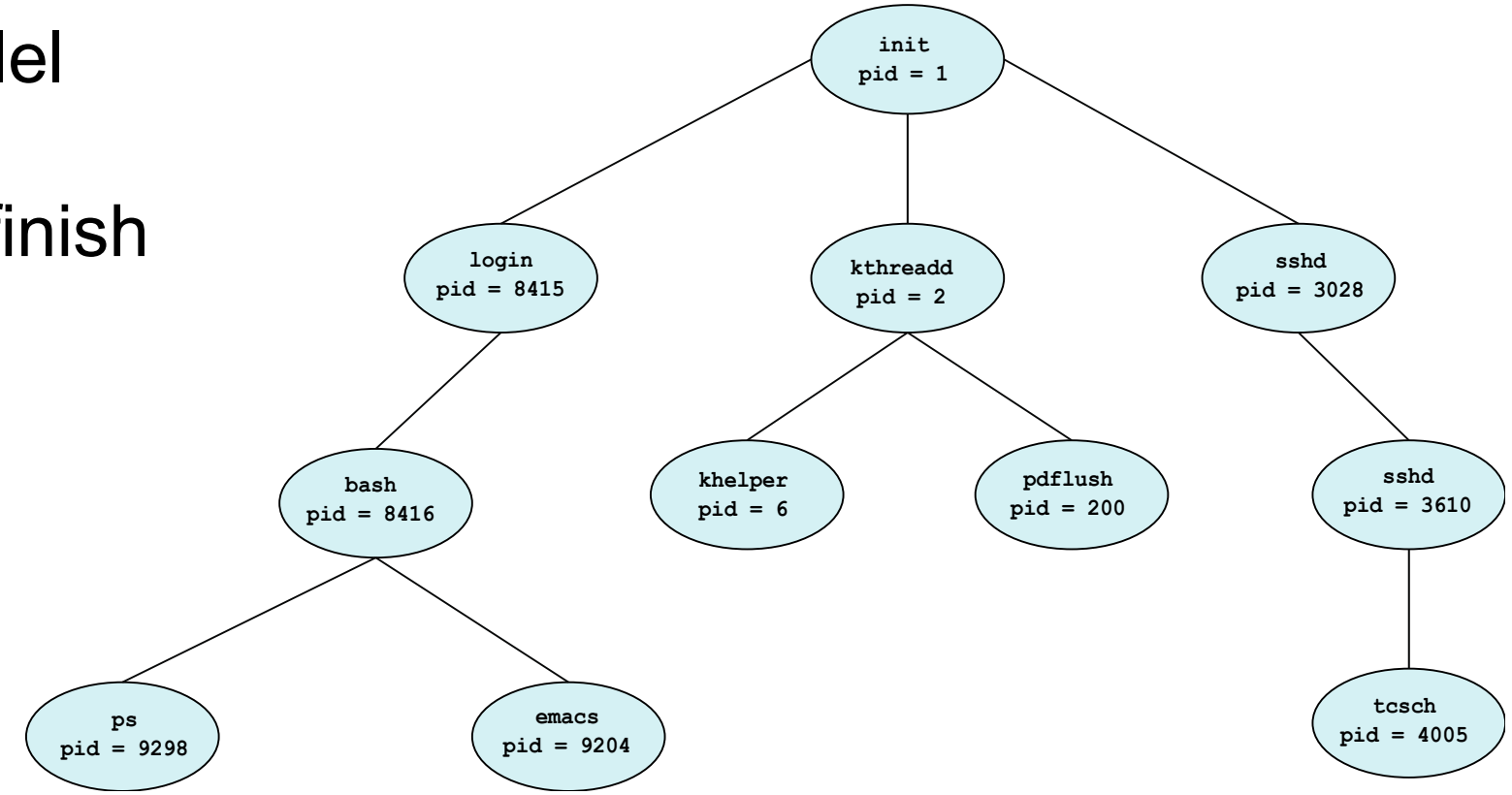
Image will be replaced with /bin/ls




```
Before exec
01_Prog1.c   01_Prog1
02_Prog2.c   02_Prog2
```

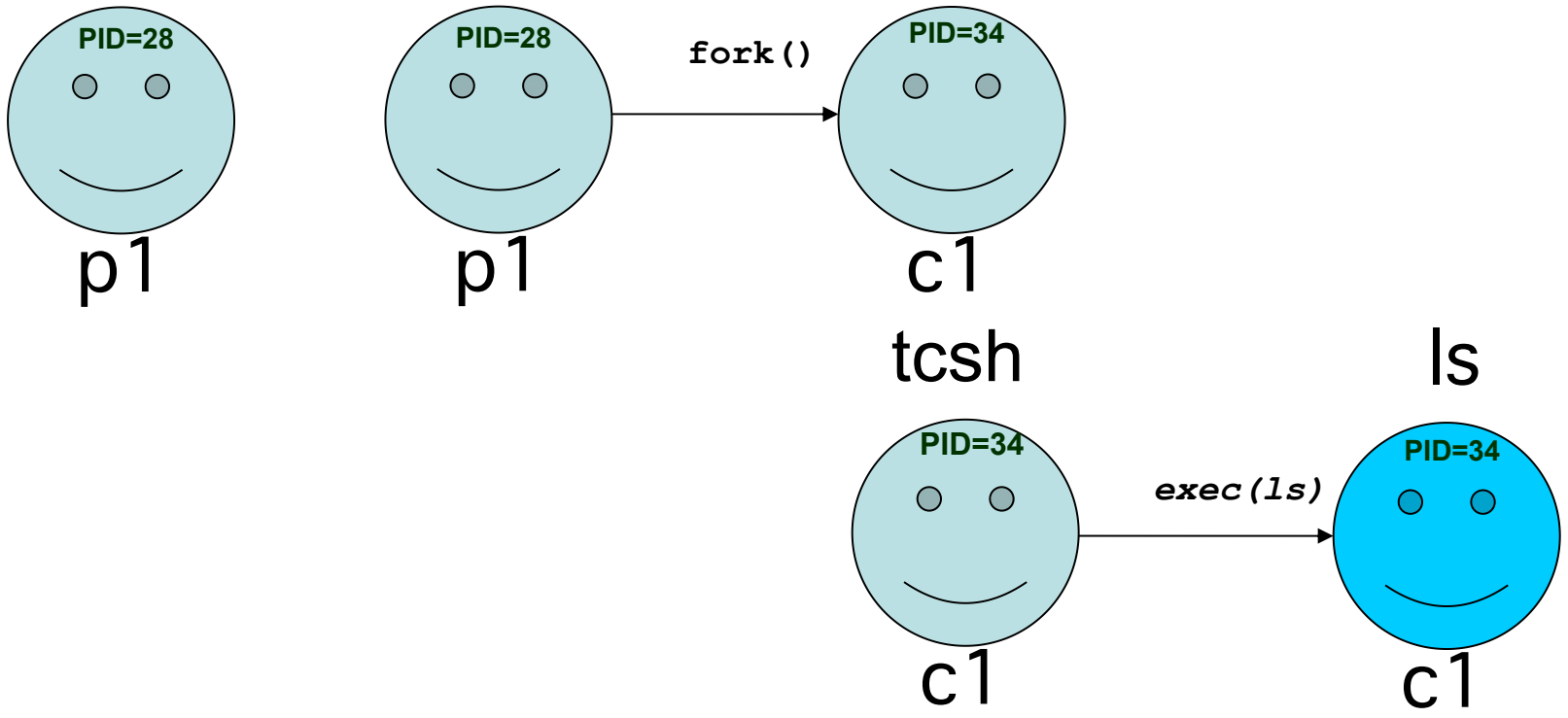
What parent does while the child is executing ??

- **execute** in parallel
- **wait** for child to finish



fork() and exec() together

- Often after doing **`fork()`** we want to load a new program into the child.
- Most common e.g. a shell programs



wait() System Call

- Forces the parent to suspend execution, i.e. wait for the child to terminate.

```
#include<unistd.h>
#include<sys/wait.h>

pid_t wait(int *status);
```

- When the child process terminates, it returns a **termination status** to the operating system, which is then returned to the waiting parent process if the **status** is not NULL. The status can be then be analyzed by the parent.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

Example of wait when forking separate process

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

int main(void)
{
     pid_t pid;
     /* fork a child process */
     pid=fork();

     if(pid<0){
        printf("Error");
        return 1;     }

    else if(pid==0){
        /*child process */
         execlp("/bin/ls","ls", NULL);  }

    else{
       /*Parent process waits for child process to complete*/
        wait(NULL);
        printf("Child Completes");  }
}
```

```
01_Prog1.c   01_Prog1
02_Prog2.c   02_Prog2
Child Completes
```

exit() System Call

- A process can either terminate normally or abnormally (e.g. divide by zero error).
- **exit( )** system call enables explicit call for normal termination and gracefully terminates process execution (it does clean up and release of resources).

  void exit(int status);

- The status argument given to **exit()** defines the exit status of the process. It is an integer value between 0 and 255.
- By convention, when a process exits with a status of zero that means it terminated normally and didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.

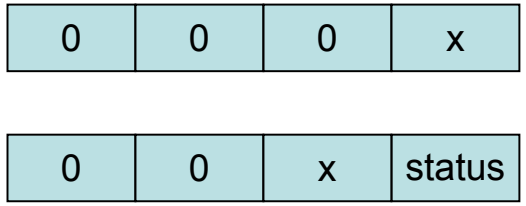
Example of wait() and exit()

```
main()
{
  int pid; int rv;

  pid=fork();
  switch(pid){
    case -1:
      printf("Error -- Something went wrong with fork()\n");
      exit(1); // parent exits
    case 0:
      printf("CHILD: This is the child process!\n");
      printf("CHILD: My PID is %d\n", getpid());
      printf("CHILD: Enter my exit status: ");
      scanf(" %d", &rv);
      printf("CHILD: I'm outta here!\n");
      exit(rv);
    default:
      printf("PARENT: This is the parent process!\n");
      printf("PARENT: My child's PID is %d\n", pid);
      printf("PARENT: I'm now waiting for my child to exit()...\n");
      wait(&rv);
      printf("PARENT: I'm outta here!\n");
  }
}
```

More about wait() and exit()

- Should not interpret the status value of system call **`wait(&status)`** literally. If **`&status`** is not NULL, **`wait()`** stores status information in the **`int`** to which it points.
- Value returned by **`exit(&status)`** is moved to 2nd byte and 1st (lowest) byte is used to store the status information.
- In previous example:

```
   scanf(" %d", &rv); // if value of x is entered
   ...
   exit(rv);
...
   wait(&rv); // the rv contents will be x left shift
              // by 8 bits and additional status
              // written into lowest 8 bit
```

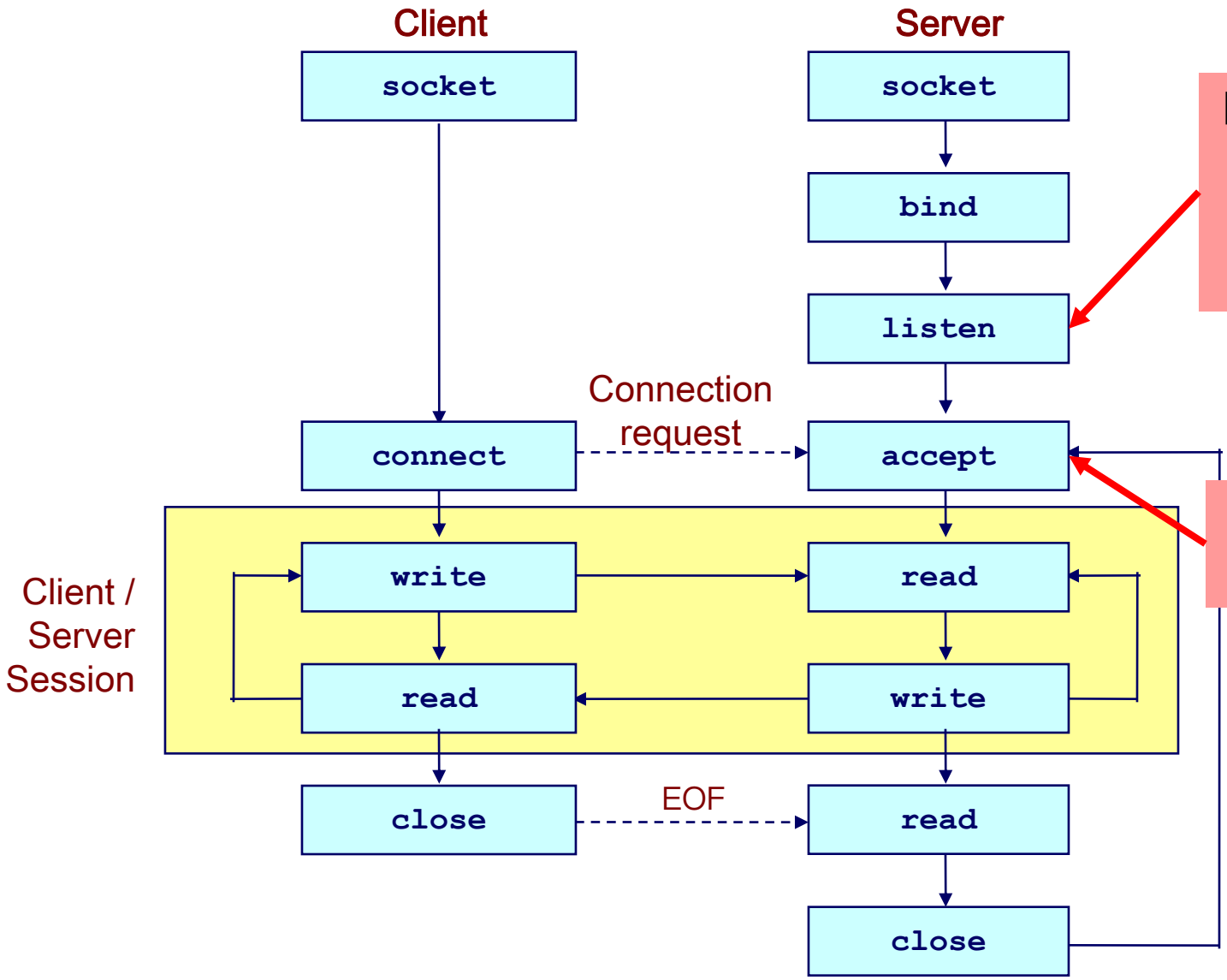


More about wait() and exit()

- This `status` integer can be inspected with macros:
  - `WIFEXITED(status)`
  - `WEXITSTATUS(status)`
  - `WIFSIGNALED(status)`
  - `WTERMSIG(status)`
  - `WCOREDUMP(status)`
  - `WIFSTOPPED(status)`
  - `WSTOPSIG(status)`
  - `WIFCONTINUED(status)`

Socket Programming – Handling Multiple Clients

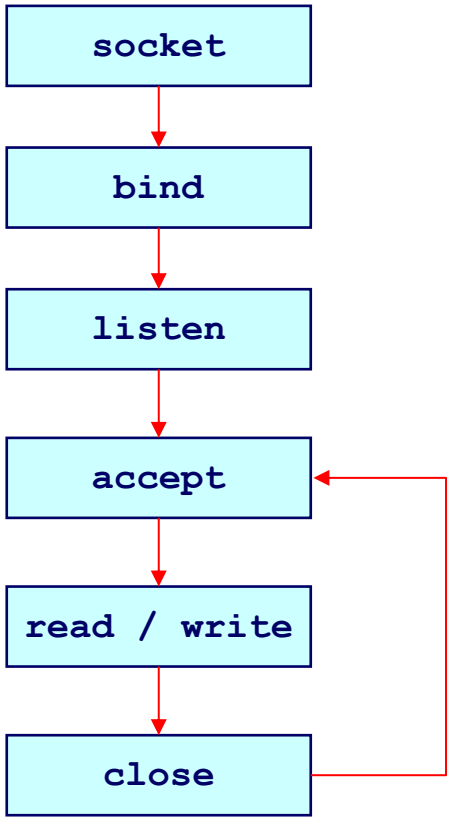
Revisit - listen() and accept()



Non-blocking call: tells OS that process is interested to accept connections and returns immediately

Blocking call: will wait until there is incoming connection

Drawback of our server



- Can only establish at most one client session at any given time
- Iterative server
- Other connecting clients will have to wait
  - If backlog is reached, these clients may get dropped
- How to make server be able to handle more than one client session concurrently?

Socket programming with fork()

- Call fork() after accepting a client connection
- In parent process, go back to accept()
- In child process, handle communication with client