IT628: Systems Programming

Process termination, wait/waitpid, exec

When does a process finish?

- **A process terminates for one of 3 reasons:**
  - It calls exit();
  - It returns (an int) from main
  - It receives a signal (from the OS or another process) whose default action is to terminate

- **Key observation: the dying process *produces status information*.**
  - Who looks at this? The parent process!

- `void exit(int status);`
  - Terminates a process with a specified status
  - By convention, status of 0 is normal exit, non-zero indicates an error of some kind

```
void foo() {
  exit(1); /* no return */
}

int main() {
  foo();   /* no return */
  return 0;
}
```

Reaping Children

- **wait(): parents reap their dead children**
  - Given info about why child died, exit status, etc.

- **Two variants**
  - wait(): wait for and reap next child to exit
  - waitpid(): wait for and reap specific child

`pid_t wait(int *stat_loc);`

when called by a process with >=1 children:

- ***waits*** **(if needed) for a child to terminate**
- ***reaps*** **a terminated child (if >= 1 terminated children, arbitrarily pick one)**
- ***returns*** **reaped child's pid and exit status info via pointer (if non-NULL)**

when called by a process with no children:

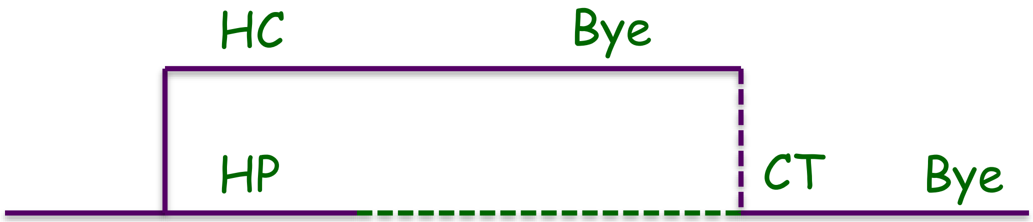
- **return -1 immediately**

```
int main() {
   pid_t cpid;
   if (fork()== 0)
      exit(0);              /* terminate child */
   else
      cpid = wait(NULL); /* reaping parent */

   printf(“Parent pid = %d\n”, getpid());
   printf(“Child pid = %d\n”, cpid);

   while (1);               /* Infinite loop */
}
```

```
int main() {
   if (fork()== 0){
      printf(“HC: hello from child\n”);
   } else {
      printf(“HP: hello from parent\n”);
      wait(NULL);
      printf(“CT: child has terminated\n”);
   }
   printf(“Bye\n”);
}
```



```
int main() {
   if (fork()== 0){
      printf(“HC: hello from child\n”);
   } else {
      printf(“HP: hello from parent\n”);
      wait(NULL);
      printf(“CT: child has terminated\n”);
   }
   printf(“Bye\n”);
}
```

A.

HP
CT
HC
Bye
Bye

B.

HP
HC
CT
Bye
Bye

C.

HP
HC
Bye
CT
Bye

D.

HC
Bye
HP
CT
Bye

E.

HC
HP
Bye
CT
Bye

```
void wait4() {
  int stat;
  if (fork() == 0)
     exit(1);
  else
     wait(&stat);
  printf(“%d\n”, stat);
}
```

```
linux> ./wait4
256
```

Child status information

- **status information about the child reported by wait is more than just the exit status of the child**
  - normal/abnormal termination
  - termination cause
  - exit status

WIF... macros

- **`WIFEXITED(status)`: child exited normally**
  - **`WEXITSTATUS(status)`**: return code when child exits

- **`WIFSIGNALED(status)`: child exited because a signal was not caught**
  - **`WTERMSIG(status)`**: gives the number of the terminating signal

- **`WIFSTOPPED(status)`: child is stopped**
  - **`WSTOPSIG(status)`**: gives the number of the stop signal

```
/* prints information about a signal */
```

- **`void psignal(unsigned sig, const char *s);`**

```c
void wait5() {
  int stat;
  if (fork() == 0)
     exit(1);
  else
     wait(&stat);
  if (WIFEXITED(stat))
     printf(“Exit status: %d\n”, WEXITSTATUS(stat));
  else if (WIFSIGNALED(stat))
     psignal(WTERMSIG(stat), “Exit signal”);
}
```

```
linux> ./wait5
Exit status: 1
```

```
void wait6() {
  int stat;
  if (fork() == 0)
     *(int *)NULL = 0;
  else
     wait(&stat);
  if (WIFEXITED(stat))
     printf(“Exit status: %d\n”, WEXITSTATUS(stat));
  else if (WIFSIGNALED(stat))
     psignal(WTERMSIG(stat), “Exit signal”);
  return 0;
}
```

```
linux> ./wait6
Exit signal: Segmentation fault
```

- **If multiple children completed, will reap in arbitrary order**

```
void wait7() {
  int i, stat;
  pid_t pid[5];
  for (i=0; i<5; i++)
     if ((pid[i] = fork()) == 0){
         sleep(1);
         exit(100+i);
     }
  for (i=0; i<5; i++) {
     pid_t cpid = wait(&stat);
     if (WIFEXITED(stat))
        printf(“Child %d terminated with status: %d\n”,
                cpid, WEXITSTATUS(stat));
  }
}
```

waitpid(): waiting for a specific process

- Useful when parent has more than one child, or you want to check for exited child but not block

```
pid_t  result =
     waitpid(child_pid,
             &status,
             options);
```

The child to wait for/check on
-1 means any child

0 = no options, wait until child exits
WNOHANG = don't wait, just check

- Return value
  - pid of child, if child has exited
  - 0, if using WNOHANG and child hasn't exited

- **Can use waitpid() to reap in order**

```
void wait8() {
  int i, stat;
  pid_t pid[5];
  for (i=0; i<5; i++)
     if ((pid[i] = fork()) == 0){
         sleep(1);
         exit(100+i);
     }
  for (i=0; i<5; i++) {
     pid_t cpid = waitpid(pid[i], &stat, 0);
     if (WIFEXITED(stat))
        printf(“Child %d terminated with status: %d\n”,
                cpid, WEXITSTATUS(stat));
  }
}
```

- **Can use WNOHANG to avoid busy waiting**

```
void wait9() {
  int i, stat;
  pid_t cpid;
  if (fork() == 0) {
     printf(“Child pid = %d\n”, getpid());
     sleep(3);
     exit(1);
  } else {
  /* use with -1 to wait on any child (with options) */
     while ((cpid = waitpid(-1, &stat, WNOHANG)) == 0) {
         sleep(1);
         printf(“No terminated children\n”);
     }
     printf(“Reaped %d with exit status: %d\n”,
                cpid, WEXITSTATUS(stat));
  }
}
```

exec(): Loading and Running Programs

- **After fork, the child process is an identical duplicate of the parent process**
- **How do we start a new program, instead of copying the parent?**
  - Use the exec() system call

```
int execve( char *filename, char *argv[ ],
char *envp );
```

- `filename`: **name of the executable file to run**
- `argv`: **command line arguments**
- `envp`: **environment variable settings (e.g.** `$PATH`**,** `$HOME`**, etc.)**
- **returns -1 if error, otherwise doesn’t return**

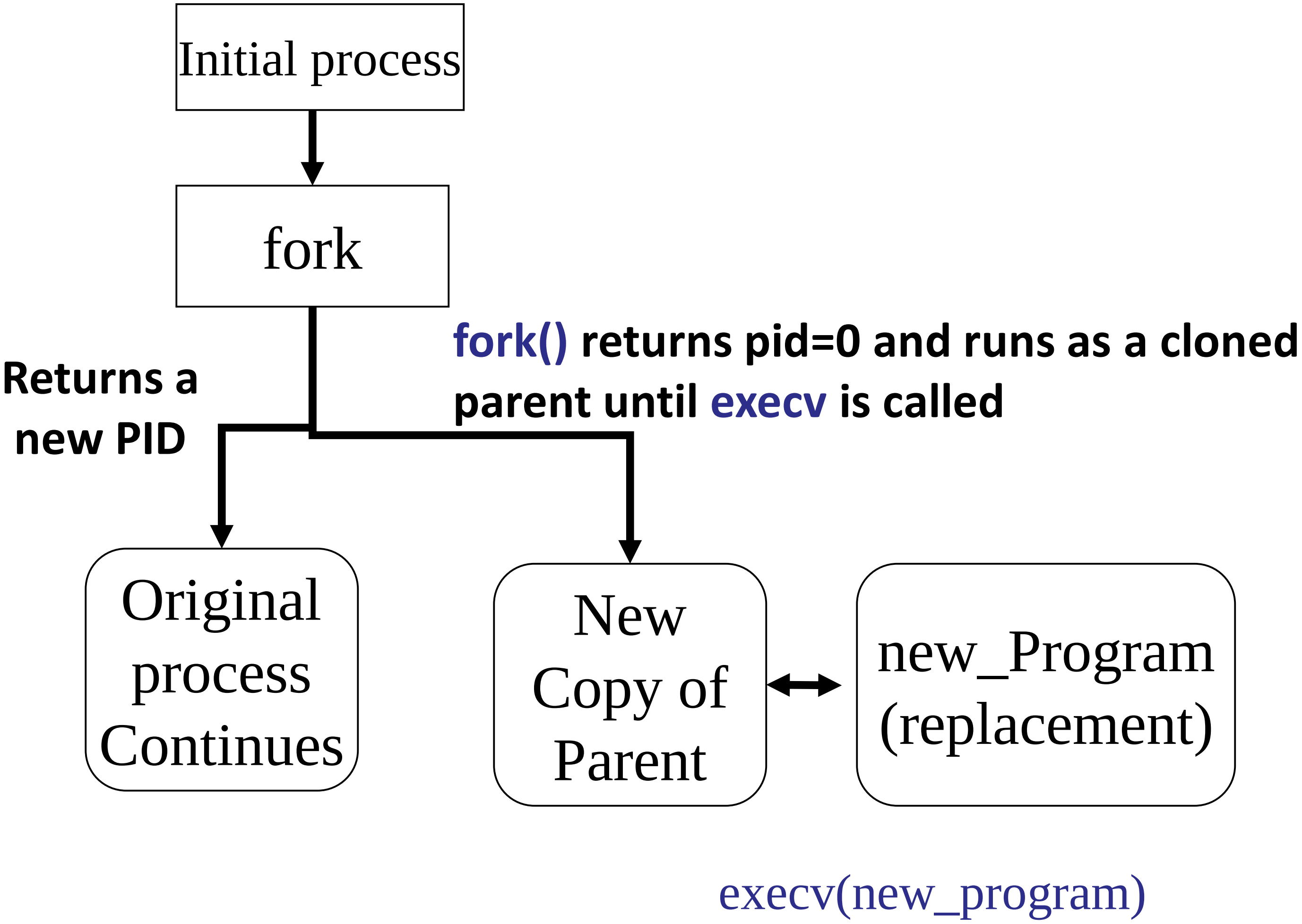
```
int main() {        /* exec1.c */
   char *args[2];
   args[0] = “/bin/echo”;
   args[1] = NULL;
   execv(“/bin/echo”, args);
   return 0;
}
```

```c
int main() {      /* exec2.c */
    char *args[2];
    args[0] = “/bin/echo”;
    args[1] = NULL;
    printf(“About to exec from process %d\n”, getpid());
    sleep(1);
    execv(“./exec2”, args);
    printf(“Done exec-ing ...\n”);
    return 0;
}
```

- **exec() does not create a new process!**
  - Replaces the address space and CPU state of the current process
  - Loads the address space from the executable file and starts it from main()
- On success, exec does not return!

- **UNIX shells use fork-then-exec to run programs**

execv(new_program, argv[ ])



Initial process

fork

Returns a new PID

fork() returns pid=0 and runs as a cloned parent until **execv** is called

Original process Continues

New Copy of Parent

new_Program (replacement)

execv(new_program)

Exercise

Write a program that creates a child process, the child executes /bin/ls, and then the parent prints “done”. Make sure the word “done” is printed after the output of ls.

```
int main() {      /* exec3.c */
   if (fork() == 0) { /* Child process */
      char *args[2];
      args[0] = “/bin/ls”; /* Not required!! */
      args[1] = NULL; /* Indicate end of args array */
      execv(“/bin/ls”, args);
      exit(0);         /* in case exec fails! */
    }
    wait(NULL);
    printf(“Done\n”);
    return 0;
}
```

Waiting for a Child Process

- If a process (the parent) calls fork() to create a process (the child), the parent doesn't automatically wait for the child to finish. The parent must call wait.
- So if wait is not called, which process finishes first?
- Either one could finish first.

Zombies

- **If the parent finishes first, the child becomes an orphan and is *adopted* by a system process called init whose pid is 1.**
- **If the child finishes first, it becomes a *zombie*.**
- **The child is mostly dead, but the parent might call waitpid. So it's termination information must be retained until the parent either terminates or calls waitpid.**

Summary

- **Basic functions**
  - fork spawns new processes
  - exit terminates own process
  - wait and waitpid wait for and reap terminated children
  - execve runs new program in existing process

Lab 1 (22/1) preparation

- **Pre-lab task**
  - before you come to the lab session
  - Setup Linux on your laptop
    - install linux in a VM
    - install Windows Subsystem for Linux (WSL)
  - You will lose points if you come to the lab without finishing this task
- **In-lab task**
  - follow instructions