Process Termination, Zombies

Systems Software

When does a process die?

- **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**
  - Reaping: cleaning up and removing the entries of terminated processes from the Process Table.
  - 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:

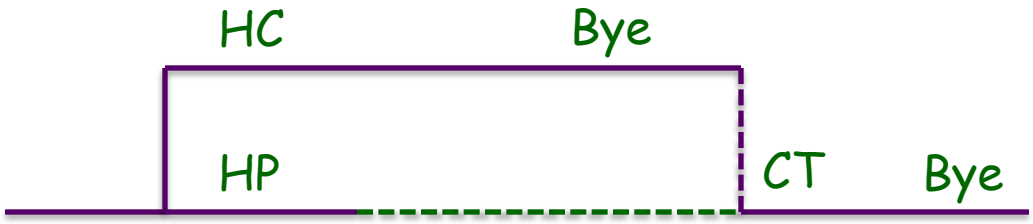
- **return -1 immediately**

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

   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
```

- **256 means child process exited with status code of 1**

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);`**

```
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));
  }
}
```

What should happen if dead child processes are never reaped? (That is, the parent has not waited() on them.)

1. **The OS should remove them from the process table**
2. **The OS should leave them in the process table**
3. **The neglected processes seek revenge as undead in afterlife**

Zombies

(Bet you didn't expect to see THAT title on a slide for a programming course?)

- **Zombie: A process that has terminated but not been reaped by its parent (AKA defunct process) (<u>parent is live</u>)**

- **"dead" but still tracked by the OS**
  - Parent may still reap them, want to know status
  - Don't want to re-use the process ID

- **Does not respond to signals (can't be killed)**

Example

```
void fork7() {
  if (fork() == 0) {
     printf("Terminating Child, PID=%d\n",getpid());
  } else {
     printf("Running Parent, PID=%d\n", getpid());
     while (1);   /* Infinite loop */
  }

}
```

Reaping children

- Parents are responsible for reaping their children
- What should happen if parent terminates without reaping its children?
- Who reaps the children?

Orphaned Processes

- **Orphan: A process that has not been reaped by its <u>terminated parent</u>**
- **Orphaned processes are adopted by the OS kernel**
- **… and the kernel always reaps its children**

Example

```
int main() {
  int i;
  for (i=0; i < 3; i++) {
      if (fork() == 0)
          exit(0);
  }
  printf("Parent pid = %d\n", getpid());
  return 0;   /* parent exits */
}
```