Process Management II

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
Recall: all processes turn into zombies upon termination
- no longer runnable, but still tracked by OS kernel
§ Reaping Processes
(& Synchronization)
All processes are responsible for reaping their own (immediate) children
So what happens if we don’t?
```c
int main() {
    int i;
    for (i=0; i<3; i++) {
        if (fork() == 0)
            exit(0);
    }
    printf("Parent pid = %d\n", getpid());
    while (1); /* non-terminating parent */
}
```

```
$ ./a.out &
Parent pid = 7254

$ ps -g 7254
    PID  STAT  TT  STAT      TIME   COMMAND
    7254  S    s003 S      0:00.01   ./a.out
    7255  Z    s003 Z      0:00.00   (a.out)
    7256  Z    s003 Z      0:00.00   (a.out)
    7257  Z    s003 Z      0:00.00   (a.out)
```
```c
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) */
}
```

```bash
$ ./a.out
Parent pid = 7409

$ ps -g 7409
   PID STAT   TT  STAT      TIME COMMAND
```
Q: How to kill a zombie?
A: By shooting it in the head! (i.e., terminating its parent process)
Orphaned processes (i.e., with terminated parents) are adopted by the OS kernel

... and the kernel always reaps its children
It is especially important for long-running processes to reap their children (why?)
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) */
}

Q: who reaps the parent??
A: The **Shell**!
```c
int main() {
    printf("My parent's pid = %d\n", getppid());
    printf("My own pid = %d\n", getpid());
    return 0; /* terminate -> zombie */
}
```

$ ./a.out
My parent's pid = 7600
My own pid = 7640

$ ps
```
PID    STAT  TT  STAT     TIME COMMAND
7600   Ss    s005 Ss     0:28.32  -bash
```

The **Shell!** (how does it do it?)
pid_t reap(int *stat_loc);

(I wish)
pid_t wait(int *stat_loc);
pid_t wait(int *stat_loc);

when called by a process with ≥ 1 children:
- \textit{waits} (if needed) for a child to terminate
- \textit{reaps} a zombie child (if ≥ 1 zombified children, arbitrarily pick one)
- \textit{returns} reaped child’s pid and exit status info via pointer (if non-NULL)
pid_t wait(int *stat_loc);

when called by a process with no children:
- return -1 immediately & populate errno
int main() {
    pid_t cpid;
    if (fork() == 0)
        exit(0);   /* child -> zombie */
    else
        cpid = wait(NULL); /* reaping parent */

    printf("Parent pid = %d\n", getpid());
    printf("Child pid  = %d\n", cpid);
    while (1);
}

$ ./a.out &
Parent pid = 7505
Child pid  = 7506

$ ps -g 7505
   PID STAT   TT STAT      TIME COMMAND
 7505 R s003 R   0:00.05 ./a.out
void fork9() {
    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");
}
```c
void fork9() {
    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");
}
```
wait allows us to synchronize one process with events (e.g., termination) in another
int main() {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        } else {
            wait(NULL);
            printf("4");
        }
    } else {
        if (fork() == 0) {
            printf("1");
            exit(0);
        }
        printf("2");
    }
    printf("0");
    return 0;
}
int main() {
    int stat;
    if (fork() == 0)
        exit(1);
    else
        wait(&stat);
    printf("%d\n", stat);
    return 0;
}

$ ./a.out
256
“status” reported by wait is more than just the exit status of the child; e.g.,

- normal/abnormal termination
- termination cause
- exit status
/* macros */
WIFEXITED(status)    /* exited normally? */
WEXITSTATUS(status)  /* if so, exit status */
WIFSTOPPED(status)   /* process stopped? */
WIFSIGNALED(status)  /* process signaled? */
WTERMSIG(status)     /* if so, signal number */

/* prints information about a signal */
void psignal(unsigned sig, const char *s);
```c
int main() {
    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");
    return 0;
}
```

```
$ ./a.out
Exit status: 1
```
```c
int main() {
    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;
}
```

```
$ ./a.out
Exit signal: Segmentation fault
```
void fork10() {
    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));
    }
}
/* explicit waiting -- i.e., for a specific child */
pid_t waitpid(pid_t pid, int *stat_loc, int options);

/** Wait options **/

/* return 0 immediately if no terminated children */
#define WNOHANG 0x00000001

/* also report info about stopped children (and others) */
#define WUNTRACED 0x00000002
void fork11() {
    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));
    }
}
int main() {
    int 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));
    }
}

Child pid = 8885
No terminated children!
No terminated children!
No terminated children!
Reaped 8885 with exit status 1
Recap:

- *fork*: create new (duplicate) process
- *exit*: terminate process
- *wait*: reap terminated (zombie) process
§ Running new programs (within processes)
/* the "exec family" of syscalls */

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
Execute a *new program* within the *current process context*
Complements `fork` (1 call $\rightarrow$ 2 returns):

- when called, `exec` (if successful) never returns!
  
- starts execution of new program
```c
int main() {
    execl("/bin/echo", "/bin/echo",
          "hello", "world", (void *)0);
    printf("Done exec-ing...\n");
    return 0;
}
```

```
$ ./a.out
hello world
```
```c
int main() {
    printf("About to exec!\n");
sleep(1);
execl("./execer", ".\execer", (void *)0);
printf("Done exec-ing...\n");
return 0;
}
```

```
$ gcc execer.c -o execer
$ ./execer
About to exec!
About to exec!
About to exec!
About to exec!
About to exec!
...```
```c
int main () {
    if (fork() == 0) {
        execl("/bin/ls", "ls", "-l", (void *) 0);
        exit(0); /* in case exec fails */
    }
    wait(NULL);
    printf("Command completed\n");
    return 0;
}
```

```
$ ./a.out
-rw-r--r--  1 lee  staff      267 Feb 8 01:51 demo.c
Command completed
```
Interesting question:

Why are `fork` & `exec` separate syscalls?

/* i.e., why not: */

fork_and_exec("/bin/ls", ...)


A1: we might really want to just create duplicates of the current process (e.g.?)
A2: we might want to *replace* the current program *without creating* a new process.
A3 (more subtle): we might want to “tweak” a process before running a program in it