CS241 Systems Programming

Discussion Section Week 2

Original slides by: Stephen Kloder

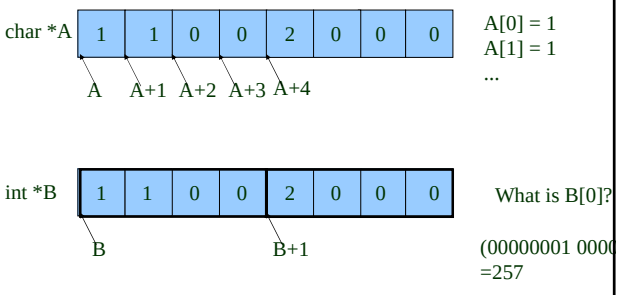
Today's Topic

- MP0 problems
- Topics needed for MP1
  - Process
  - Fork
  - Wait
  - Exec

About MP0

- overlay_overlay(char *s1, char *s2, char *s3, char *s4)
  - Find the overlay between s1 and s2 = o1
  - Find the overlay between s3 and s4 = o2
  - Find the overlay between o1 and o2 = result
- How to find the overlay between “father” and “mother”?
  - Possible approach: generate all possible combination of “mother”
  - *strstr()* with “father” to find the largest match

About MP0 (contd.)



MP1

- Simple Unix shell
- Non built-in commands
- Built-in commands
  - Change Directory
  - Termination
  - History
- Error handling
- Concepts needed: process, fork, exec, wait

Processes

A process is an instance of a running program

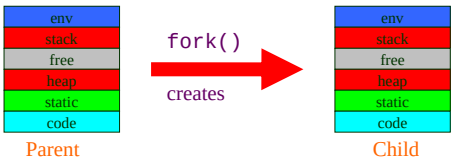
A process contains:

- Instructions (i.e. the program)
- Resources (variables, buffers, links, etc.)
- State (identity, ready/running/locked, etc.)

Processes can create other processes

Process Creation with fork()

`fork()` creates a new process:



- The new (child) process is identical to the old (parent) process, except…

Differences between parent and child

Process ID (`getpid()`)

Parent ID (`getppid()`)

Return value of `fork()`

- In parent, `fork()` returns child pid
- In child, `fork()` returns 0

`fork()` Example 1

What does this do?

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

int main() {
     printf("%d\n", fork());
     return 0;
}
```

Try it!

`fork()` example 2

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

int main() {
   pid_t child_pid = fork();
   if (child_pid < 0) {        // error code
      perror("Fork Failed");
      return -1;
   }
   printf("I'm process %d\n",getpid());

   if (child_pid == 0) {       // child code
      printf("I'm the child of parent process %d.\n", getppid());
   } else { /* child_pid > 0 */     // parent code
      printf("I'm the parent of child process %d.\n", child_pid);
   }
   return 0;
}
```

Example 2 cont'd

This exits too quickly; let's slow it down:

```
if (child_pid == 0) {     // child code
    sleep(15);
    printf("I'm the child of parent process %d.\n",
    getppid());
} else {                  // parent code
    sleep(20);
    printf("I'm the parent of child process %d.\n",
           child_pid);
}
```

Example 2 cont'd

In a second window, run `ps -a`, and look for the pids from the program output.

Periodically run `ps -a` again, as the program in the first window executes.

- What happens when the program runs?
- What happens when the child finishes?
- What happens when the parent finishes?
- What happens when you switch the parent and child sleep statements?

Orphans and Zombies

When a process finishes, it becomes a *zombie* until its parent cleans up after it.

If its parent finishes first, the process becomes an *orphan*, and the `init` process (id 1) adopts it.

How can a parent know when its children are done?

Solution: `wait(…)`

`wait()` allows a parent to wait for its child process, and save its return value

```
pid= wait(&status);
pid= waitpid(pid, &status, options);
```

`wait()` waits for any child;

`waitpid()` waits for a specific child.

`wait` cont'd

`wait()` blocks until child finishes

- `wait()` does not block if the option `WNOHANG` is included. When would we want to use this?

The child's return value is stored in `*status`

`wait(...)` macros

`WIFEXITED(status)` is true iff the process terminated normally.

`WEXITSTATUS(status)` gives the last 8 bits of the process's return value (assuming normal exit)

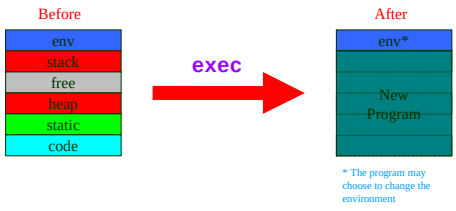
Example 3: `wait`

```
#include <sys/wait.h>
  …
  …

    // add this to parent code
    if (waitpid(child_pid, &result, 0) == -1) {
      perror("Wait failed");
      return -1;
    }
    if(WIFEXITED(result)) {
    fprintf(stdout, "child %d returned %d\n",
        child_pid, WEXITSTATUS(result));
    }
```

`exec`

`exec` replaces the current process image(code, variables, etc.) with that of a new program:



`exec` variations

There are 6 different ways of calling `exec`. Which one to use depends on three conditions:

1. How arguments are passed
   - execl, execv
2. How the path is specified
   - execlp, execvp
3. Whether a new environment is used
   - execle, execve

exec variations: passing parameters

exec can have parameters passed to it two different ways:

List of parameters:

```
execl("/bin/ls", "ls", "-l", NULL);
```

Argument Vector (like argv):

```
execv("/bin/ls", argv); // char *argv[]
```

Q: When would you use execl?

When would you use execv?

exec variations: command path

Adding a "p" to an exec call tells the system to look for the command in the environment's path.

Compare:

- `execl("/bin/ls", "ls", "-l", NULL);`
- `execlp("ls", "ls", "-l", NULL);`

The difference is similar for execv and execvp.

exec variations (cont'd)

By default, the new program inherits the old program's environment. Adding an "e" to the exec call allows the new program to run with a new environment **environ

execve and execle allow the user to specify the environment. The others inherit the old environment.

fork + exec

If exec succeeds, it does not return, as it overwrites the program.

No checking return values, executing multiple commands, monitoring results, etc.

Solution: fork a new process, and have the child run exec

Example 4: exec

```
int main() {
     char command[10];
     while(fscanf(stdin, “%s”, command)) {
         pid_t child_pid = fork();
         if (child_pid < 0) {
             perror("Fork failed");
             return -1;
         } else if (child_pid == 0) {
             // child code
             // execute command
             // check errors
         } else {
             // parent code
             // What belongs here?
         }
     }
     return 0;
}
```