CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

Unix Processes (cont.)

Today's Topics

- More about execve
- Debugging

fork()

- A forked child inherits open files of the parent
- The child process descriptor is a copy of the parent's process descriptor, except:
  - Return value from `fork()`
  - PID, PPID
  - Pending signals and alarms
  - File locks
  - Execution times

Executing a New Binary

- `execve() is used to execute a new program`
- Manual page!
- This function executes the program it is pointed to
- On success, `execve()` does not return: The process calling execve() is completely replaced by the newly executed process
- On error, -1 is returned
- File descriptors may be set to close on exec!

Creating a New Process

- Exec is most useful when used with fork
- In Unix, a new process is created by first forking an existing process, then calling a variant of exec from there
- Most process attributes are preserved, including the PID, PPID, file locks, pending signals, execution times and open files

execve() Example

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

main() {
  char *a[4], *e[3];

  a[0] = "child";
  a[1] = "argument1";
  a[2] = "argument2";
  a[3] = NULL;

  e[0] = "ENV0=val0";
  e[1] = "ENV1=val1";
  e[2] = NULL;

  execve("child1", a, e);
  /* If we get here, something went wrong */
  perror("parent1");
  exit(1);
}
```

execve() Example

```
#include <stdio.h>

main(argc, argv, envp)
int argc;
char *argv[], *envp[];
{
   int i;
   char **ep;
   printf("child is running\n");
   for (i = 0; i < argc; i++) {
      printf("argv[%d]=%s\n",i,argv[i]);
   }
   for (ep = envp; *ep; ep++) {
      printf("%s\n", *ep);
   }
}
```

More Examples

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main(argc, argv)
int argc; char *argv[];
{
  int forkid, charnum;
  char fdval[20];
  if (argc != 3) {
    fprintf(stderr,"Usage: pexec filename charnum\n");
    exit(1);
  }
  if ((forkid = open(argv[1], O_RDONLY)) < 0 ) {
    fprintf(stderr, "Cannot open %s\n", argv[1]);
    exit(2);
  }
  sprintf(fdval, "%d", forkid); /* sprintf! */
  if (fork() == 0) {
    execl("pchild", "pchild", fdval, argv[2], (char *)0);
    fprintf(stderr, "Unable to exec\n");
    exit(3);
  }
  printf("Parent is after fork/exec\n");
}
```

More Examples

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

main(argc, argv)
int argc;char *argv[];
{
   int myfd;
   char gotch, val;
   if (argc != 3) {
      fprintf(stderr,"Usage: pchild filename charnum\n");
      exit(1);
   }
   myfd = atoi(argv[1]);
   gotch = atoi(argv[2]);
   lseek(myfd, (off_t)gotch, SEEK_SET);
   read(myfd, &val, 1);
   printf("Child got char %d from fd %d: %c\n", gotch, myfd, val);
}
```

More Examples

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main(argc, argv)
int argc; char *argv[];
{
  int forkid, charnum;
  if (argc != 3) {
    fprintf(stderr,"Usage: pioexec filename charnum\n");
    exit(1);
  }
  if ((forkid = open(argv[1], O_RDONLY)) < 0 ) {
    fprintf(stderr, "Cannot open %s\n", argv[1]);
    exit(2);
  }
  if (fork() == 0) {
    close(0); dup(forkid); close(forkid);
    execl("piochild", "piochild", argv[2], (char *)0);
    fprintf(stderr, "Unable to exec\n");
    exit(3);
  }
  printf("Parent is after fork/exec\n");
}
```

More Examples

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

main(argc, argv)
int argc;char *argv[];
{
   int myfd;
   char gotch, val;
   if (argc != 2) {
      fprintf(stderr,"Usage: piokid charnum\n");
      exit(1);
   }
   gotch = atoi(argv[1]);
   lseek(0, (off_t)gotch, SEEK_SET);
   read(0, &val, 1);
   printf("Child got char %d from stdin: %c\n", gotch, val);
}
```

File Descriptor Example

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

main() {
   pid_t cpid;
   int x, fd;
   char ch;

   x = 5;
   cpid = fork();
   if(cpid == 0) {
      fd = open("aFile", O_RDWR|O_CREAT, 0644);
      x++; ch=x+48;
      write(fd,&ch,1);
   } else {
      fd = open("aFile", O_RDWR|O_CREAT, 0644);
      x++; ch=x+48;
      write(fd,&ch,1);
   }
}
```

More Examples

How would we need to code to get the following process structure:

- Parent
  - Child 1
  - Child 2

More Examples

How would we need to code to get the following process structure:

- Parent
  - Child 1
  - Child 2

More Examples

How would we need to code to get the following process structure:

- Parent
  - Child 1
    - Child 2

Exec Example

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

main() {
   pid_t cpid;
   int i;

   for(i = 0; i < 2; i++) {
      cpid=fork();
      if (cpid==0) execl("bogus", "bogus", (char *)0);
   }
}
```

Exec Example

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

main() {
   pid_t cpid;
   int i;

   for(i = 0; i < 2; i++) {
      cpid=fork();
      if (cpid==0) {
         execl("bogus", "bogus", (char *)0);
         exit(1);
      }
   }
}
```

Parent/Child Synchronization (wait/exit)

- The exit() and _exit() calls
- As usual, take a look at the manual page.
- Terminates the calling process immediately
- As a convention: exit status of 0 is *normal* termination
- Any other status denotes an error or an exceptional condition on termination

wait() call

- The wait() call
- The manual!
- A parent process is obligated to wait for its children to exit
- Suspends execution of the current process until a child has exited

No Waiting

```c
#include <unistd.h>
#include <stdlib.h>
main() {
   if (fork() == 0) {
      exit(1);
   } else {
      sleep(20);
      exit(1);
   }
}
```

With Waiting

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main() {
   int status;
   if (fork() == 0) {
      sleep(20);
      exit(51);
   } else {
      printf("pid = %d\n", wait(&status));
      printf("status = %x\n", status);
      if (WIFEXITED(status))
         printf("Status (via macro): %d\n", WEXITSTATUS(status));
      exit(0);
   }
}
```

Debugging

- We'll focus on gcc and gdb
- We'll also take a look at ddd, which is a GUI for gdb and various other debuggers
- What is debugging?
  - Program *state* is a snapshot of all variables, PC, etc.
  - A statement in your program transforms one program state into another
  - You should be able (at some level) to express what you expect the state of your program to be after every statement
  - Often state predicates on program state; i.e., "if control is here, I expect the following to be true."
- Let's look at a toy example

Sample Program

```
#include <stdio.h>
int sum=0, val, num=0;
double ave;
/* sum and num should be 0 */
main()
{
  /* sum should be the total of the num
  input values processed */
  while (scanf("%d\n",&val) != EOF) {
    sum += val;
    num++;
  }
  /* sum should be the total of the num
  input values and there is no more input */
  if (num > 0) {
    ave= sum/num;
    /* ave should be the floating point mean of
    the num input data values */
    printf("Average is %f\n", ave);
  }
}
```

Using gdb

- Make sure to compile source with the -g switch asserted.
- In our case, `gcc -g ave.c`
- Breakpoint: line in source code at which debugger will pause execution.
- At breakpoint, can examine values of relevant components of program state. `break` command sets a breakpoint; `clear` removes the breakpoint.
- Diagnostic `printf()` crude way of getting a snapshot of program state at a given point.
- Once paused at a breakpoint, use gdb `print`, or `display` to show variable or expression values. `display` will automatically print values when execution halts at breakpoint.
- From a breakpoint, may `step` or `next` to single step the program. `step` stops after next source line is executed. `next` similar, but executes functions without stopping.

Using gdb

- Can find out where execution is, in terms of function calls, with the `where` command.
- Let's play with the toy program!
- We'll put the buggy set in a data file.
- We can also debug post mortem in crashing programs (bintree.c)!
- May need to enable `ulimit -c`

A GUI for gdb: ddd

- The ddd program is a front-end for gdb
- Can use the mouse to set breakpoints!
- Values are graphically displayed
- Can visualize complex structures

Debugging Tips

- Examine the most recent change
- Debug it now, not later
- Read before typing
- Make the bug reproducible
- Display output to localize your search
- Write a log file
- Use tools
- Keep records