

Lecture Thread: Unix process-oriented system calls

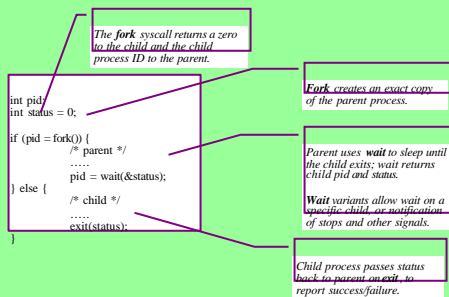
Lecture context switch =

Timer interrupt goes off + Save current lecture state (ESC out of current powerpoint slideshow) + Load state of other lecture (choose powerpoint window) + Reset timer + Run new lecture (Click on slideshow)

Unix Process Model

- Simple and powerful primitives for process creation and initialization.
 - *fork* syscall creates a *child* process as (initially) a clone of the parent
 - parent program runs in child process to set it up for *exec*
 - child can *exit*, parent can *wait* for child to do so.
- Rich facilities for controlling processes by asynchronous *signals*.
 - notification of internal and/or external events to processes or groups
 - the look, feel, and power of interrupts and exceptions
 - default actions: stop process, kill process, dump core, no effect
 - user-level handlers

Unix Process Control



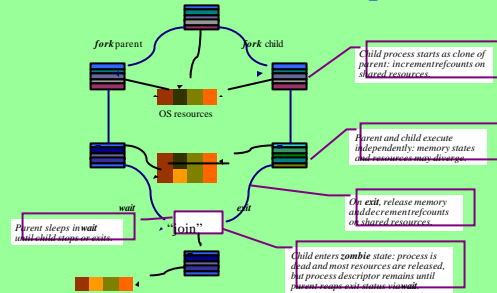
Child Discipline

- After a *fork*, the parent program (*not process*) has complete control over the behavior of its child process.
- The child inherits its execution environment from the parent...but the parent *program* can change it.
 - sets bindings of file descriptors with *open*, *close*, *dup*
 - *pipe* sets up data channels between processes
- Parent program may cause the child to execute a different program, by calling *exec** in the child context.

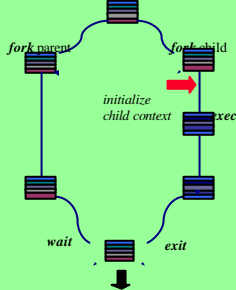
Exec, Execve, etc.

- Children should have lives of their own.
- Exec* “boots” the child with a different executable image.
 - parent program makes *exec** syscall (in forked child context) to run a program in a new child process
 - *exec** overlays child process with a new executable image
 - restarts in user mode at predetermined entry point (e.g., *cr0*)
 - no return to parent program (it’s gone)
 - arguments and environment variables passed in memory
 - file descriptors etc. are unchanged

Fork/Exit/Wait Example



Fork/Exec/Exit/Wait Example



```
int pid = fork();
    Create a new process that is a clone of its parent.

exec*("program" [, argv, envp]);
    Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);
    Exit with status, destroying the process.

int pid = wait*(&status);
    Wait for exit (or other status change) of a child.
```

Join Scenarios

- Several cases must be considered for join (e.g., *exit/wait*).
 - What if the child exits before the parent joins?
 - “Zombie” process object holds child status and stats.
 - What if the parent continues to run but never joins?
 - Danger of filling up memory with zombie processes?
 - Parent might have specified it was not going to wait or that it would ignore its child’s exit. Child status can be discarded.
 - What if the parent exits before the child?
 - Orphans become children of *init* (process 1).
 - What if the parent can’t afford to get “stuck” on a join?
 - Asynchronous notification (we’ll see an example later).

Unix Signals

- Signals notify processes of internal or external events.
 - the Unix software equivalent of interrupts/exceptions
 - only way to do something to a process “from the outside”
 - Unix systems define a small set of signal types
- Examples of signal generation:
 - keyboard *ctrl-c* and *ctrl-z* signal the *foreground process*
 - synchronous fault notifications, syscall errors
 - asynchronous notifications from other processes via *kill*
 - IPC events (SIGPIPE, SIGCHLD) `signal == 'upcall'`
 - alarm notifications

Process Handling of Signals

1. Each signal type has a system-defined default action.
 - abort and dump core (SIGSEGV, SIGBUS, etc.)
 - ignore, stop, exit, continue
2. A process may choose to *block* (inhibit) or *ignore* some signal types.
3. The process may choose to *catch* some signal types by specifying a (user mode) *handler* procedure.
 - specify alternate signal stack for handler to run on
 - system passes interrupted context to handler
 - handler may munge and/or return to interrupted context

Predefined Signals (a Sampler)

Name	Default action	Description
SIGINT	Quit	Interrupt
SIGILL	Dump	Illegal instruction
SIGKILL	Quit	Kill (can not be caught, blocked, or ignored)
SIGSEGV	Dump	Out of range addr
SIGALRM	Quit	Alarm clock
SIGCHLD	Ignore	Child status change
SIGTERM	Quit	Sw. termination sent by kill

User's View of Signals

```

int alarmflag=0;
alarmHandler ()
{ printf("An alarm clock signal was received\n");
  alarmflag = 1;
}
main()
{
  signal(SIGALRM, alarmHandler);
  alarm(3); printf("Alarm has been set\n");
  while (!alarmflag) pause ();
  printf("Back from alarm signal handler\n");
}
    
```

Instructs kernel to send SIGALRM in 3 seconds
 Sets up signal handler
 Suspends caller until signal

Yet Another User's View

```

main(argc, argv)
int argc, char* argv[];
{
    int pid;
    signal (SIGCHLD, childhandler);
    pid = fork ();
    if (pid == 0) /*child*/
    { execvp (argv[2], &argv[2]); }
    else
    {sleep (5);
    printf("child too slow\n");
    kill (pid, SIGINT);
    }
}

childhandler()
{ int childPid, childStatus;
  childPid = wait (&childStatus);
  printf("child done in time\n");
  exit;
}
    
```

Collects status

SIGCHLD sent by child on termination; if SIG_IGN, dezombie

What does this do?

Files (& everything else)

- *Descriptors* are small unsigned integers used as handles to manipulate objects in the system, all of which resemble files.
- *open* with the name of a file returns a descriptor
- *read* and *write*, applied to a descriptor, operate at the current position of the file offset. *lseek* repositions it.
- Pipes are unnamed, unidirectional I/O stream created by *pipe*.
- Devices are special files, created by *mknod*, with *ioctl* used for parameters of specific device.
- Sockets introduce 3 forms of *sendmsg* and 3 forms of *recvmsg* syscalls.

File Descriptors

- Unix processes name I/O and IPC objects by integers known as *file descriptors*.
 - File descriptors 0, 1, and 2 are reserved by convention for *standard input*, *standard output*, and *standard error*.
 - “Conforming” Unix programs read input from *stdin*, write output to *stdout*, and errors to *stderr* by default.
 - Other descriptors are assigned by syscalls to open/create files, create pipes, or bind to devices or network sockets.
 - *pipe*, *socket*, *open*, *creat*
 - A common set of syscalls operate on open file descriptors independent of their underlying types.
 - *read*, *write*, *dup*, *close*

File System Calls

```

char buf[BUFSIZE];
int fd;

if ((fd = open("./zot", O_TRUNC | O_RDWR) == -1) {
    perror("open failed");
    exit(1);
}

while(read(0, buf, BUFSIZE) {
    if (write(fd, buf, BUFSIZE) != BUFSIZE) {
        perror("write failed");
        exit(1);
    }
}
    
```

Open files are named to by an integer file descriptor.

Pathnames may be relative to process current directory.

The perror C library function examines errno and prints type of error.

Process passes status back to parent on exit to report success/failure.

Process does not specify current file offset; the system remembers it.

Standard descriptors (0, 1, 2) for input/output/error messages (stdin, stdout, stderr).

File Sharing Between Parent/Child

```
main(int argc, char *argv[]) {
    char c;
    int fdrd, fdwt;

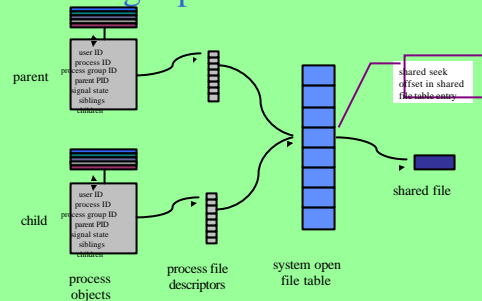
    if ((fdrd = open(argv[1], O_RDONLY)) == -1)
        exit(1);
    if ((fdwt = creat(argv[2], 0666)) == -1)
        exit(1);

    fork();

    for (;;) {
        if (read(fdrd, &c, 1) != 1)
            exit(0);
        write(fdwt, &c, 1);
    }
}
```

[Bach]

Sharing Open File Instances

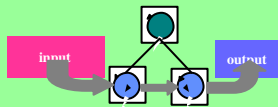


Producer/Consumer Pipes

```
char inbuffer[1024];
char outbuffer[1024];

while (inbytes != 0) {
    inbytes = read(stdin, inbuffer, 1024);
    outbytes = process data from inbuffer to outbuffer;
    write(stdout, outbuffer, outbytes);
}
```

Pipes support a simple form of parallelism with built-in flow control.

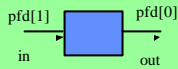


e.g.: `sort <grades | grep Dan | mail nadia`

Unnamed Pipes

- Buffers up to fixed size.
- Reading from a pipe:
 - If write end has been closed, returns end-of-input.
 - If pipe is empty on attempted read, sleep until input available.
 - Trying to read more bytes than are present, returns # bytes read
- Writing to a pipe:
 - Read end closed, writer is sent SIGPIPE signal (default is to terminate receiver)
 - Writing fewer bytes than capacity → write is atomic
 - Writing more bytes than capacity → no atomicity guarantee.

Setting Up Pipelines



```

int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
int in, out; /* pipeline entrance and exit */

pipe(pfd); /* create pipeline entrance */
out = pfd[0];
in = pfd[1];

/* loop to create a child and add it to the pipeline */
for (i = 1; i < procCount; i++) {
    out = setup_child(out);
}

/* pipeline is a producer/consumer bounded buffer */
write(in, ..., ...);
read(out, ..., ...);
    
```

Setting Up a Child in a Pipeline

```

int setup_child(int fd) {
    int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
    int i, wfd;

    pipe(pfd); /* create right-hand pipe */
    wfd = pfd[1]; /* this child's write side */
    if (fork()) { /* parent */
        close(wfd); close(rfd);
    } else { /* child */
        close(pfd[0]); /* close far end of right pipe */
        close(0); /* stdin */ close(1); /* stdout */
        dup(rfd); /* takes fd 0 */ dup(wfd); /* takes fd 1 */
        close(rfd); close(wfd);
    }
    return(pfd[0]);
}
    
```

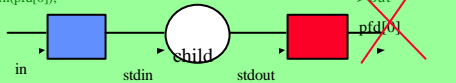


Setting Up a Child in a Pipeline

```

int setup_child(int fd) {
    int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
    int i, wfd;

    pipe(pfd); /* create right-hand pipe */
    wfd = pfd[1]; /* this child's write side */
    if (fork()) { /* parent */
        close(wfd); close(rfd);
    } else { /* child */
        close(pfd[0]); /* close far end of right pipe */
        close(0); /* stdin */ close(1); /* stdout */
        dup(rfd); /* takes fd 0 */ dup(wfd); /* takes fd 1 */
        close(rfd); close(wfd);
    }
    ...
}
return(pfd[0]);
}
    
```



Sockets for Client-Server Message Passing

Server

1. Create a named socket
syscalls:
sfd = (socket, ...)
bind(sfd, ptr, ...)
2. Listen for clients
listen(sfd, ...)
3. Connection made and continue listening
cfd = accept(sfd, ...)
5. Exchange data
write(cfd, ...)
6. Done: close(cfd);
close(sfd);

Client

3. Create unnamed socket & ask for connection
syscalls:
cfd = socket(...)
err = connect(cfd, ptr, ...)
5. Exchange data
read(cfd, ...)
6. Done: close(cfd);

