Processes, Unix, and IPC

Dr. Daniel Andresen

CIS520 – Operating Systems

Today

Today we continue with process management topics.

- last time: address space creation/initialization
- today: process interactions and synchronization

    Exec/Exit/Join synchronization

    A peek at the Unix process model, its strengths and limitations

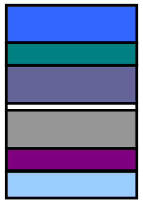
    Start on mechanisms for Interprocess Communication (IPC)

IPC mechanisms combine synchronization and data transfer

    pipes, streams, messages, Remote Procedure Call (RPC)

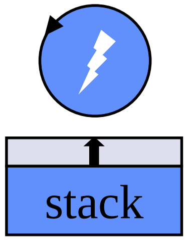
Process Internals

virtual address space



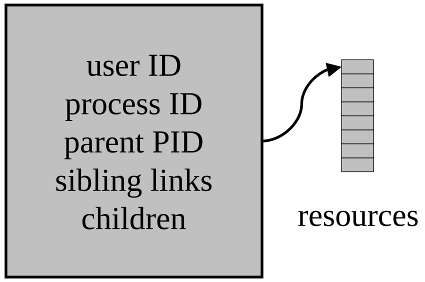
+

thread



+

process descriptor



The address space is represented by *page table*, a set of translations to physical memory allocated from a kernel *memory manager*.

The kernel must assign memory to back the VAS, and initialize it for the program to run.

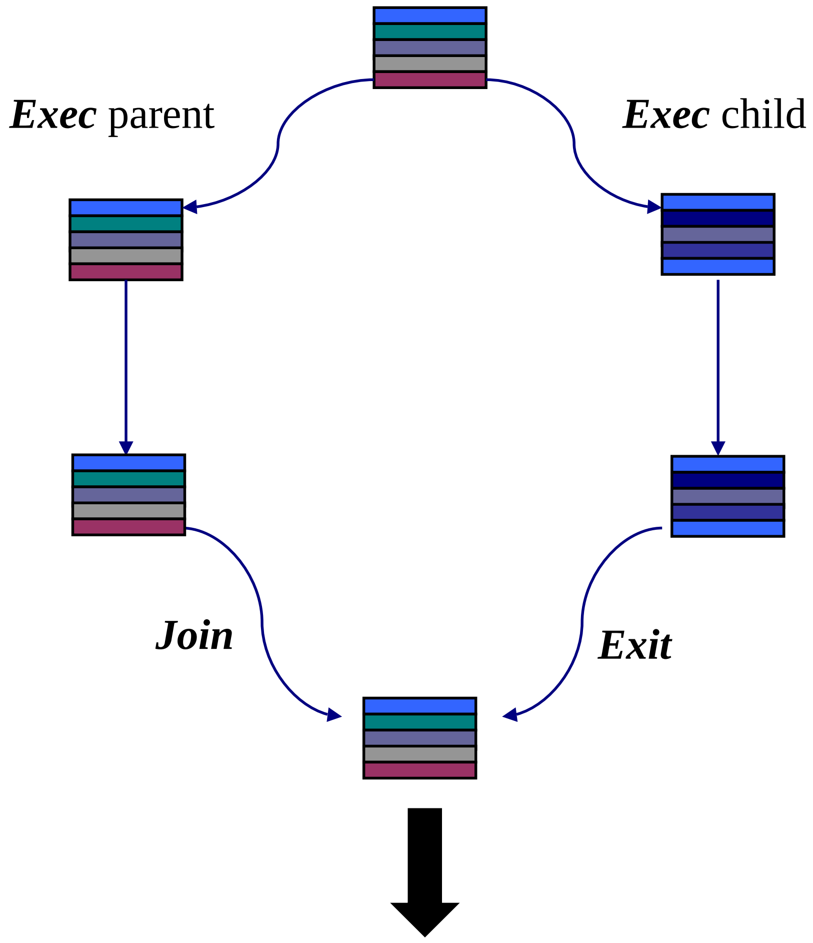
Each process has a thread bound to the VAS.

The thread has a saved user context as well as a system context.

The kernel can manipulate the user context to start the thread in user mode wherever it wants.

Process state includes a file descriptor table, links to maintain the *process tree*, and a place to store the exit status.

Nachos Exec/Exit/Join Example



SpaceID pid = Exec("myprogram", 0);

*Create a new process running the program "myprogram". Note: in Unix this is two separate system calls: **fork** to create the process and **exec** to execute the program.*

int status = Join(pid);

Called by the parent to wait for a child to exit, and "reap" its exit status. Note: child may have exited before parent calls Join!

Exit(status);

Exit with status, destroying process. Note: this is not the only way for a proess to exit!.

Process Management Example

```
ExecHandler (char* executable)
        Process* p = new Process();
        Process* parent = CurrentProcess();
        create and initialize VAS for p;
        p->Birth(parent);
        start thread in child context;
        allocate and return SpaceID;

JoinHandler(int spaceid)
        Process* child = get from spaceID;
        int status = child->Join();
        return(status);

ExitHandler(int status)
        Process* p = CurrentProcess();
        tear down p's VAS, release memory;
        p->Death(status);
        delete p;
        destroy this thread;
```

1. Parent waits in *Join* for child to exit (by calling *Death*).

2. Exited child waits in *Death* for parent to join or exit.

3. Exited parent waits in *Death* for children to exit.

Needed:
- 3 **wait**
- 3 **signal/broadcast**

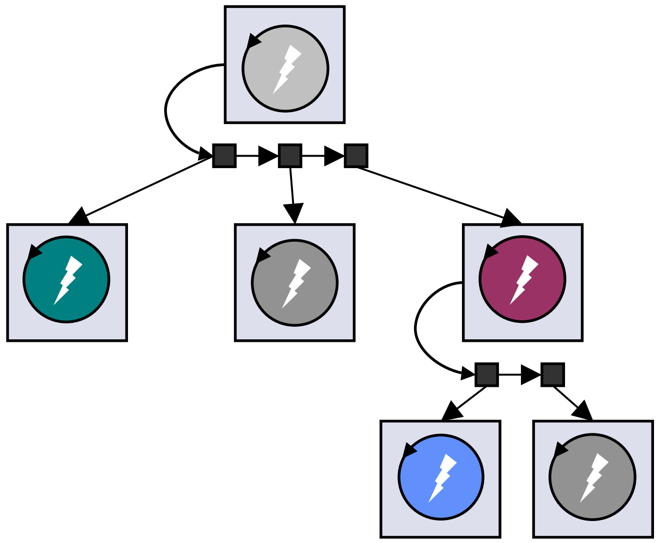
each in (*Join*, *Death*, *Death*)

Process Birth Example

pMx and *pCv* are global
(could use private CVs, but it's tricky)

```
Mutex* pMx;
Condition* pCv;

void
Process::Birth(Process *parent) {
        pMx->Acquire();
        this->parent = parent;
        parent->AddChild(this);
        status = 0;
        exited = 0;
        joinedOn = 0;
        pMx->Release();
}
```



Process Join Example

```
int
Process::Join() {
        int status;

        pMx->Acquire();
        while (!exited)
                pCv->Wait();        /* wait for child to exit */
        status = this->status;
        joinedOn = 1;
        pCv->Broadcast();           /* tell child that parent is done */
        pMx->Release();

        return(status);
}
```

1. Parent waits in *Join* for child to call *Death*.

2. Child waits in *Death* for parent to complete *Join*.

Process Death Example

```
void Process::Death(int status) {
        pMx->Acquire();

        exited = 1;                          /* process has exited */
        this->status = status;
        pCv->Broadcast();                    /* wake parent from join */

        while (!joinedOn && !parent->exited)
                pCv->Wait();
        parent->RemoveChild(this);           /* parent may delete now */
        pCv->Broadcast();

        while (!children->Empty())
                pCv->Wait();

        pMx->Release();
}
```

1. Parent waits in *Join* for child to call *Death*.

2. Child waits in *Death* for parent to call *Death*.

3. Parent waits in *Death* for children to call *Death*.

Elements of the Unix Process and I/O Model

1. rich model for IPC and I/O: “*everything is a file*”

    file descriptors: most/all interactions with the outside world are through system calls to read/write from *file descriptors*, with a unified set of syscalls for operating on open descriptors of different types.

2. simple and powerful primitives for creating and initializing child processes

    fork: easy to use, expensive to implement

3. general support for combining small simple programs to perform complex tasks

    standard I/O and pipelines: good programs don’t know/care where their input comes from or where their output goes

Unix 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.

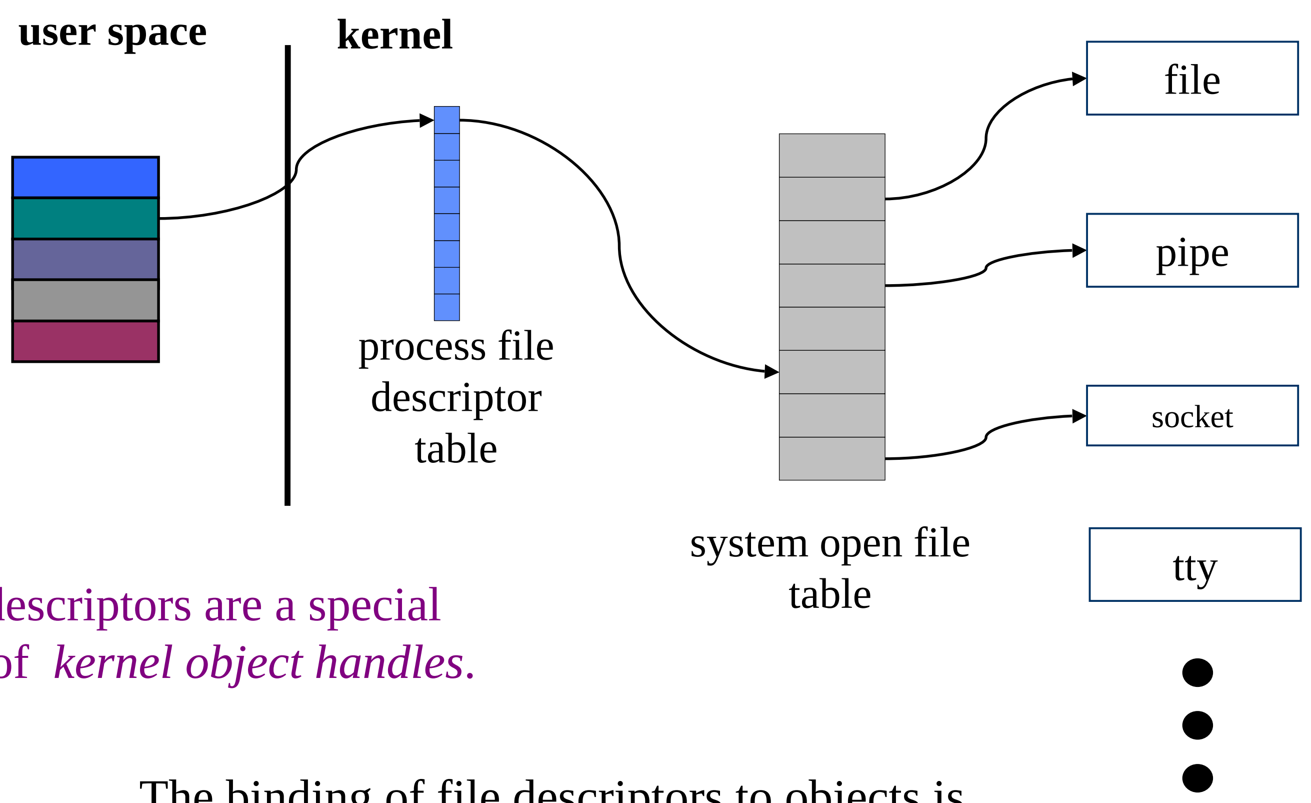
- Other descriptors are assigned by syscalls to open/create files, create pipes, or bind to devices or network sockets.

  pipe, socket, open, creat

- A common set of syscalls operate on open file descriptors independent of their underlying types.

  read, write, dup, close

Unix File Descriptors Illustrated



File descriptors are a special case of *kernel object handles*.

The binding of file descriptors to objects is specific to each process, like the virtual translations in the virtual address space.

Disclaimer: this drawing is oversimplified.

The Concept of Fork

The Unix system call for process creation is called *fork*().

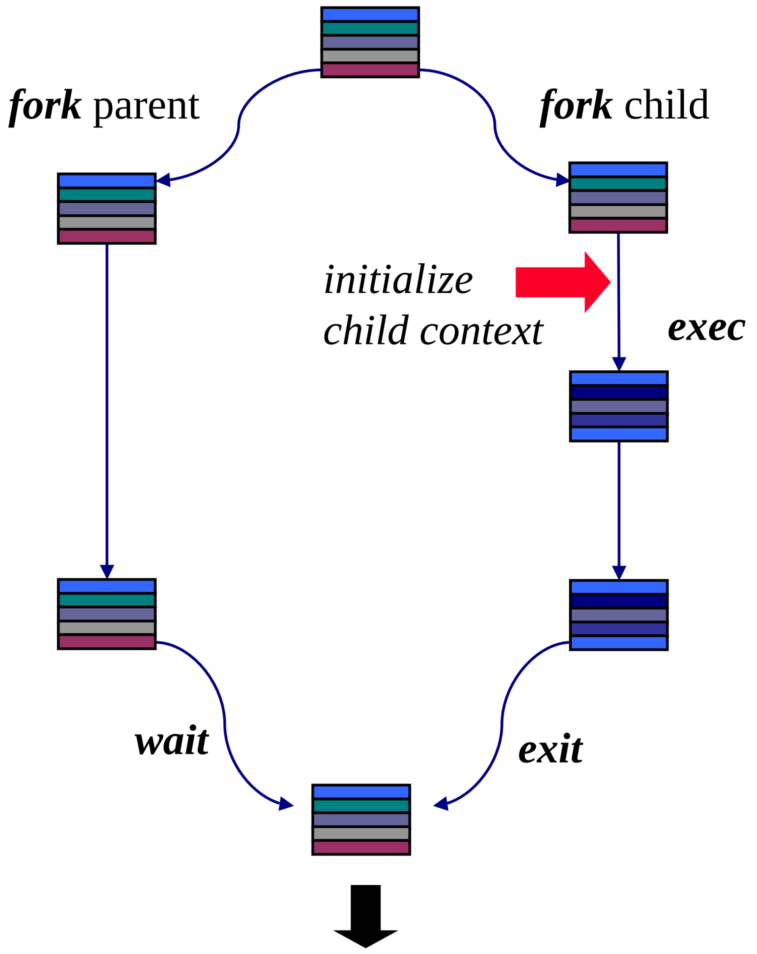
The *fork* system call creates a child process that is a clone of the parent.

- Child has a (virtual) copy of the parent's virtual memory.
- Child is running the same program as the parent.
- Child *inherits* open file descriptors from the parent.

  (Parent and child file descriptors point to a common entry in the system open file table.)
- Child begins life with the same register values as parent.

The child process may execute a different program in its context with a separate *exec()* system call.

Unix *Fork/Exec/Exit/Wait* Example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, 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.

Example: Process Creation in Unix

```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        …..
        pid = wait(&status);
} else {
        /* child */
        …..
        exit(status);
}
```

*The **fork** syscall returns twice: it returns a zero to the child and the child process ID (pid) to the parent.*

*Parent uses **wait** to sleep until the child exits; **wait** returns child pid and status.*

***Wait** variants allow wait on a specific child, or notification of stops and other signals.*

What's So Cool About *Fork*

1. *fork* is a simple primitive that allows process creation without troubling with what program to run, args, etc.

   Serves some of the same purposes as threads.

2. *fork* gives the parent program an opportunity to initialize the child process…*especially* the open file descriptors.

   Unix syscalls for file descriptors operate on the current process.

   Parent program running in child process context may open/close I/O and IPC objects, and bind them to *stdin*, *stdout*, and *stderr*.

   Also may modify environment variables, arguments, etc.

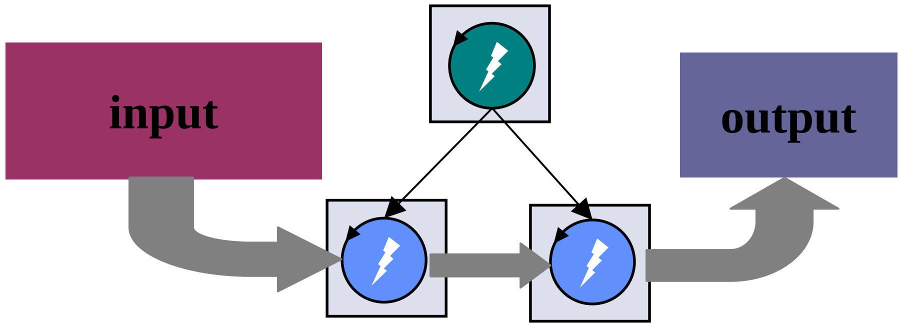
3. Using the common *fork/exec* sequence, the parent (e.g., a command interpreter or shell) can transparently cause children to read/write from files, terminal windows, network connections, pipes, etc.

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*.



This example illustrates one important use of the producer/consumer bounded buffer in Lab #3.

e.g.: *sort <grades* | *grep Dan* | *mail sprenkle*

Unix as an Extensible System

"Complex software systems should be built incrementally from components."

- independently developed
- replaceable, interchangeable, adaptable

The power of *fork/exec/exit/wait* makes Unix highly flexible/extensible...at the application level.

- write small, general programs and string them together

  general stream model of communication

- this is one reason Unix has survived

These system calls are also powerful enough to implement powerful command interpreters (*shell*).

The Shell

The Unix command interpreters run as ordinary user processes with no special privilege.

> This was novel at the time Unix was created: other systems viewed the command interpreter as a trusted part of the OS.

Users may select from a range of interpreter programs available, or even write their own (to add to the confusion).

> *csh, sh, ksh, tcsh, bash*: choose your flavor...or use *perl*.

Shells use *fork/exec/exit/wait* to execute commands composed of program filenames, args, and I/O redirection symbols.

> Shells are general enough to run files of commands (*scripts*) for more complex tasks, e.g., by redirecting shell's *stdin*.
>
> Shell's behavior is guided by *environment variables*.

Limitations of the Unix Process Model

The pure Unix model has several shortcomings/limitations:

- Any setup for a new process must be done in its context.
- Separated *Fork/Exec* is slow and/or complex to implement.

A more flexible process abstraction would expand the ability of a process to manage another externally.

This is a hallmark of systems that support multiple operating system “personalities” (e.g., NT) and “microkernel” systems (e.g., Mach).

Pipes are limited to transferring linear byte streams between a pair of processes with a common ancestor.

Richer IPC models are needed for complex software systems built as collections of separate programs.