Outline

- **Objective:**
  - Get specific with UNIX system calls
  - Continue with *message-passing* style interprocess communication (IPC) – including UNIX example

---

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 (Note: this is NOT the same as Nachos fork which is internal thread creation).
  - parent program runs in child process to set it up for *exec* (again, not a Nachos exec which is part of its API, but has different semantics)
  - 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

- *fork* syscall returns a zero to the child and the child process ID in the parent.
  - *fork* creates an exact copy of the parent process.
- *wait* waits for a child to exit and returns the child pid and status.
- *wait* variants allow wait on a specific child or notification of stops and other signals.
- Child process passes status back to parent on exit to report success/failure.
**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.

---

**Fork/Exit/Wait Example**

- Child process acts as clone of parent, incrementing refcounts of shared resources.
- Parent and child execute independently, memory states and resources may diverge.
- On exit, release memory and decrement refcounts on shared resources.
- Child enters zombie state: process is dead and most resources are released, but process descriptor remains until parent reaps exit status via wait.
- Parent sleeps in wait until child stops or exits.

---

**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., crt0)
  - no return to parent program (it’s gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged
Fork/Exec/Exit/Wait Example

- fork
  - Create a new process that is a clone of its parent.
- exec("program", [argvp, envp]);
  - Overwrite the calling process virtual memory with a new program, and transfer control to it.
- exit(status);
  - Exit with status, destroying the process.
- wait(&status);
  - Wait for exit (or other status change) of any child.

Join Scenarios

- Several cases must be considered for join (e.g., exit/wait).
  - What if the child exits before the parent does the wait?
  - "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).

Immediate Notification: Upcalls

- Problem: what if an event requires a more "immediate" notification?
  - What if a high-priority event occurs while we are executing the handler for a low-priority event?
- We need some way to preemptively "break in" to the execution of a thread and notify it of events.

upcalls
example: NT Asynchronous Procedure Calls (APC)
example: Linux signal
Prescriptive event handling raises synchronization issues similar to interrupt handling.
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
  - Inter-process Communication (IPC) events (SIGPIPE, SIGCHLD)
  - 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)

<table>
<thead>
<tr>
<th>Name</th>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Quit</td>
<td>Interrupt</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Dump</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Quit Kill (can not be caught, blocked, or ignored)</td>
<td></td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Dump</td>
<td>Out of range addr</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Quit</td>
<td>Alarm clock</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child status change</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Quit</td>
<td>Sw termination sent by kill</td>
</tr>
</tbody>
</table>
**User's View of Signals**

```c
int alarmflag = 0;
alarmHandler() {
    printf("An alarm clock signal was received in\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");
}
```

Sets up signal handler

Instructs kernel to send SIGALRM in 3 seconds

Suspends caller until signal

---

**User's View of Signals II**

```c
main() {
    int (*oldHandler)();
    printf("I can be control-c'd\n");
    sleep(3);
    oldHandler = signal(SIGINT, SIG_IGN);
    printf("I'm protected from control-c\n");
    sleep(3);
    signal(SIGINT, oldHandler);
    printf("Back to normal\n");
    sleep(3); printf("bye\n");
}
```

---

**Yet Another User's View**

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

```c
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);
}
```

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

Pathnames may be relative to process current directory.

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

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

Open files are named by an integer file descriptor.
File Sharing Between Parent/Child

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

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 mark`
**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.

```c
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]; /* create pipe */
/* 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 Pipelines**

```c
int setup_child(int rfd) {
    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);
        close(0); /* stdin */
        close(1); /* stdout */
        dup(rfd); /* takes fd 0 */
        dup(wfd); /* takes fd 1 */
        close(rfd); close(wfd);
    } else {
        /* child */
        close(pfd[0]); /* close far end of right pipe */
        close(0); /* stdin */
        dupfd(0); /* takes fd 0 */
        close(0); /* stdin */
        close(1); /* stdout */
        dupfd(1); /* takes fd 1 */
        close(1); /* stdout */
        … /* execs nth stage of pipeline */
    }
    return(pfd[0]);
}
```

**Setting Up a Child in a Pipeline**

```c
int setup_child (int rfd) {
    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);
        close(0); /* stdin */
        close(1); /* stdout */
        dup(rfd); /* takes fd 0 */
        dup(wfd); /* takes fd 1 */
        close(rfd); close(wfd);
    } else {
        /* child */
        close(pfd[0]); /* close far end of right pipe */
        close(0); /* stdin */
        dupfd(0); /* takes fd 0 */
        close(0); /* stdin */
        close(1); /* stdout */
        dupfd(1); /* takes fd 1 */
        close(1); /* stdout */
        … /* execs nth stage of pipeline */
    }
    return(pfd[0]);
```
Setting Up a Child in a Pipeline

```c
int setup_child(int rfd) {
int i, wfd;

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

Naming Destinations for Messages: Ports

Advantages of Ports

1. Ports decouple IPC endpoints from processes and threads.
   A thread may send to a port without knowing the identity of the
   process/thread that receives on that port.
   Different threads may listen to the same port, possibly at
different times.
2. A thread may listen to multiple ports, separating the
   message streams designated for different ports.
   E.g., audiophile ports to different devices or virtual services.
3. Ports are a convenient granularity to control message flow.
   E.g., selectively enable/disable ports independently, or assign
different priorities or access control to different ports.
Port Issues

1. Asymmetry and notification. How does a thread know when a message arrives on a port?
   How to receive from multiple ports, without blocking on one idle port while receiving messages on another?

2. Naming and binding. How do threads name the ports to send to or receive from (listen)?
   How do threads find the names, e.g., for services they want to use?

3. Protection and access control.
   How does the system know if a thread process has a "right" to send to or listen on a particular port? E.g., how can we prevent unauthorized programs from masquerading as a legitimate service?

Examples of Ports in Real Systems

1. Unix sockets and TCP/IP communication.
   - Common primitives/protocols for local messaging and network communication.
   - TCP/IP defines a fixed space of port numbers per node.
     System calls to bind to a particular port.
   - Some ports are reserved to processes running with superuser (root) privilege.
     Standard servers in /etc/services listen at well-known protected ports.

2. Mach supplies a rich set of port messaging primitives.
   - Open ports ("port rights") are kernel object handles.
   - Port rights may be passed as messages among processes.
     The only way to gain a communicative right is for some other process to pass it to you! This is a system-wide basis for protection.

Sockets for Client-Server Message Passing

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

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

Communication-oriented systems face an important problem:
How does a client or server know what to do next?

- Servers in networks or server-oriented systems must service
  many clients, possibly on different ports.
- The server must handle messages as they arrive, without blocking
  to move on to another part of the scheme before pending messages.

**Options:**

**Option 1:** Use blocking primitives with lots of threads.
Leave the scheduling to the thread scheduler.

**Option 2:** Introduce nonblocking primitives or provide
notifications or combined queuing of incoming messages.

A wide variety of mechanisms have been used to implement
polling: Unix select, lwait, port groups, event queues, etc.

Polling: Select

A thread/process with multiple network connections or open
files can initiate nonblocking I/O on all of them.
The Unix select system call supports such a polling model:
- pass a bitmask for which descriptors to query for readiness
- returns a bitmask of descriptors ready for reading/writing
- reads and/or writes on these descriptors will not block

Advantages of Server “Isolation” Afforded by Message Passing

Like the kernel, the server is isolated from its clients.

- Address space isolation is preserved, so the client cannot
  corrupt the server’s data.
- The only way a client can cause code to run in the server is
  to send a message.
- The server decides how to validate and interpret each message.
- The client is also protected from the server, although it must
  rely on its to correctly perform the service.

(Unlike the kernel, the server cannot access client memory.)
Reconsidering the Kernel Interface and OS Structure

The kernel can be thought of as nothing more than a server; it is special only in that it runs in a protected hardware mode.

- Many of the services traditionally offered by the kernel can be supported outside of the kernel, in servers or in libraries.
- What features must be implemented in the kernel? Could we implement (say) the estate Visa interface as an application?
- Why would we want to do such a thing? What are the advantages of supporting some OS features in a server rather than directly in the kernel? What are the costs?
- How would we design a kernel interface that is powerful enough to implement multiple OS “personalities“ as servers?

Servers and Microkernels

A number of systems have been structured as collections of servers running above a minimal kernel (“microkernel”).

- Microkernels provide, e.g., basic threading and scheduling, IPC, virtual address spaces, and device I/O primitives.
- Kernel is expected to be smaller, more reliable; and more secure.
- Policies (e.g., security) may be implemented outside of the kernel.
- Operating system “personalities“ (e.g., Unix or Windows) may be implemented as servers.
- OS may have multiple personalizations and policies, with new OS features and APIs added on the fly.
- The performance of server-structured systems is determined largely by the efficiency of the messaging primitives.

Microkernel with “User-Level” OS Server Processes

![Diagram of microkernel with user-level processes]
End-to-End Argument

• Application-level correctness requires checking at the endpoints to ensure that the message exchange accomplished its purpose
  – Application semantics involved
  – Notification of successful delivery (UPS tracking) is not as good as a direct response (thank you note) from the other end.

• Reliability guarantees in the message-passing subsystem provide performance benefits (short-circuiting corrective measures).
  – Re-transmitting packet may save re-transferring whole file.