

Classic Processes (CSE 422S)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

fork, exec and the shell (1)

- Example: Interactive Commands

```
> fib 10 > fib10.out
> fib 2000 > fib2000.out &
```
- Your interactive shell (tcsh, bash) interprets commands
- Shell processing of 'fib 10'
 - » Shell looks for the 'fib' executable in a directory listed in the PATH environment variable
 - e.g., " :/usr/home/kenw/bin:/usr/bin:/usr/local/bin"
 - » Forks a copy of the shell; i.e., creates a child process
 - » Child redirects stdout to the file 'fib10.out'
 - » Child execs 'fib' passing in command-line arguments
 - 'fib' process replaces the shell's child process
 - » Parent shell waits for 'fib' process to terminate
 - » Display prompt after 'fib' terminates

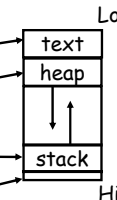
fork, exec and the shell (2)

- Shell processing of 'fib 2000'
 - » Processing is the same as for 'fib 10' except that parent does not wait for child to terminate
- Extended Example:
 - » Shell later waits for child

```
> fib 10 > fib10.out
> fib 2000 > fib2000.out &
...
> wait $!    # last bg proc
```
- Fundamental Abstraction: The Process
 - » *Traditional Process*: The process executes a single sequence of instructions in an address space
 - » The program counter (PC) tracks the sequence of instructions
 - » *Modern Process*: Multiple control paths

The UNIX Process Concept (1)

- A **process** (task) is a program in execution
 - » Needs system resources (e.g., CPU, memory) to execute
 - » Also, contains a thread of execution
 - Represented by the program counter (PC)
 - » Consists of a program area and data area
- Memory Layout
 - » Text (Instructions)
 - » Heap (Global Variables)
 - » Stack (Local Variables)
 - » Command-line args, environment variables
- Memory (address space) is virtual
 - » Part of content may be on a disk (*swap area*)
 - » The memory management subsystem shuffles parts to/from local (or remote) disk



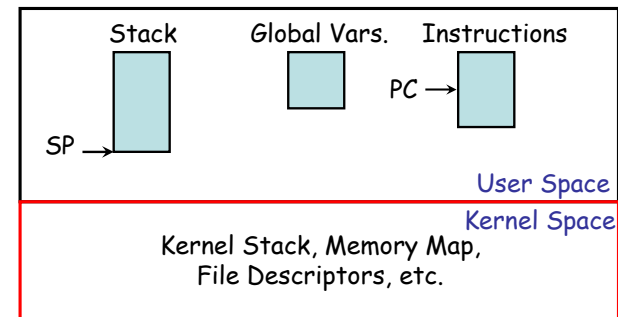
The UNIX Process Concept (2)

- Every process has a unique **process ID (PID)**
 - » PID 0: Scheduler `sched` or `swapper`, a kernel process
 - » PID 1: `/sbin/init` started at the end of the bootstrap process and never dies
 - All other processes are descendants of `init`
 - » PID 2: Page Daemon `pageout` or `pagedaemon`
- The execution environment is **multiprogrammed**
 - » Each process has a set of register values
 - » CPU only has one set of registers which is shared by all processes
 - OS kernel loads hardware registers with the register values of the currently running process
 - » Processes contend for system resources (CPU, memory, I/O devices)

5 - Ken Wong, 2/6/2006

Washington University in St. Louis

Traditional Process



- A unit of **resource ownership** (address space, Files)
- A unit of **dispatching**
 - » Has an execution state; scheduled by the OS

6 - Ken Wong, 2/6/2006

Washington University in St. Louis

OS Kernel

- The OS kernel is a special program that implements the process model and provides system services
 - » Resides on disk (e.g., `/vmunix`, `/unix`, `/boot/vmunix`)
 - » Loaded into memory from disk during startup
 - The startup procedure is called **bootstrapping**
- Services Provided
 - » Handles system call requests from user processes
 - » Handles user process generated exceptions
 - e.g., Hardware exceptions such as divide by zero, stack overflow
 - » Handles hardware interrupts
 - Devices use interrupts to notify kernel of I/O completion and status changes
 - » Performs some maintenance functions
 - e.g., Control number of active processes; allocating virtual memory

7 - Ken Wong, 2/6/2006

Washington University in St. Louis

Process Creation Events

- System initialization (created during bootup)
 - » **Background Processes: Daemons** (email, Web, printer)
 - » **Foreground Processes:** Interact with user (login-shell)
- Process creation system call (e.g., `fork`)
 - » Ex. 1: Concurrent server application forks child to handle request
 - » Ex. 2: Run `make` utility in parallel mode (`make -j 4`)
- User request
 - » Each command to a shell
- Batch job initiation
 - » Ex: User schedules job startup for 2AM using `crontab` or `at` command

8 - Ken Wong, 2/6/2006

Washington University in St. Louis

Bootstrapping

- Simple loader loads level 1 bootstrap code into memory from a known place on a disk drive
 - » Level 1 bootstrap code doesn't understand virtual memory
 - » It does know about the file system structure
- Level 1 bootstrap code
 - » Zeros memory, moves itself (relocates) to high memory
 - » Reads level 2 bootstrap code into low memory
 - » Jumps to level 2 bootstrap code
- Level 2 bootstrap code
 - » Reads OS kernel from file system into memory
 - » Sets up some form of primitive virtual memory
 - » Jumps to OS kernel
- OS Kernel
 - » Initializes virtual memory and other data structures

9 - Ken Wong, 2/6/2006

Washington University in St. Louis

Unix Process Hierarchies

- Process Group
 - » A process and all of its descendants which have not changed the default process group ID (via `setpgid(2)`)
- User sends signal from keyboard
 - » Signal delivered to all members of process group associated with keyboard
 - » Each process can:
 - Catch signal
 - Ignore signal
 - Take default action (terminate)
- The init process: root parent of all user processes
 - » See later figure
- Windows doesn't enforce a hierarchy

10 - Ken Wong, 2/6/2006

Washington University in St. Louis

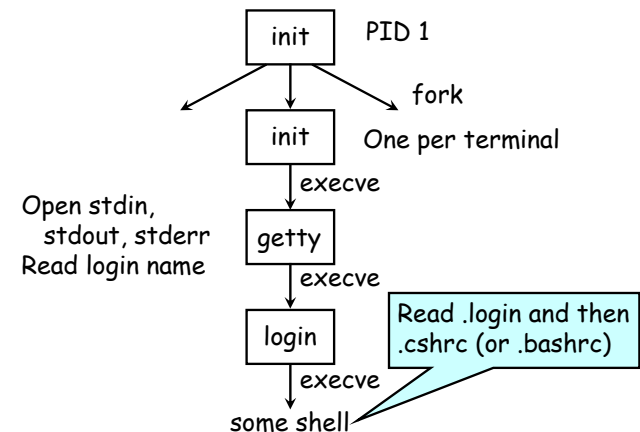
Signals From The Keyboard

- Signals here are generated by the terminal driver
- Interrupt key (ctrl-c)
 - » Generates SIGINT; terminate process
 - » Signal sent to all processes in foreground process group
- Quit key (ctrl-\)
 - » Generates SIGQUIT; terminate process with core dump
- Stop key (ctrl-z)
 - » Generates SIGSTP; stop process (can be resumed)
 - » SIGSTP (interactive stop) is different from SIGSTOP
 - » Signal sent to all processes in foreground process group

11 - Ken Wong, 2/6/2006

Washington University in St. Louis

Unix init And getty Processes

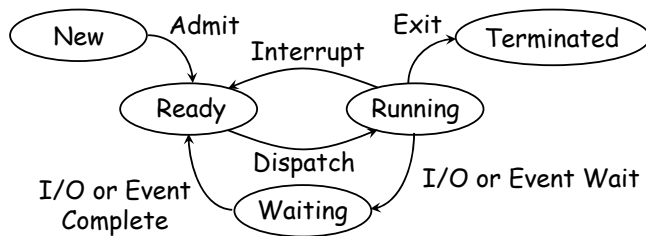


12 - Ken Wong, 2/6/2006

Washington University in St. Louis

Process Run States

Basic process run states



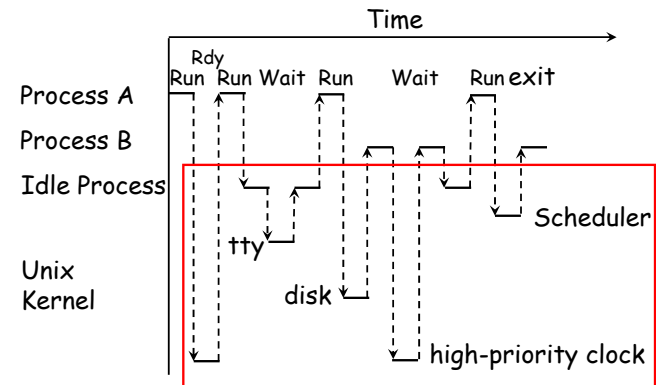
Other possible states

- » Stopped: Not terminated, but not to be scheduled
- » Running can be two states: In user space or kernel space
- » Zombie (to be described)

13 - Ken Wong, 2/6/2006

Washington University in St. Louis

Multiprogramming on a Uniprocessor



14 - Ken Wong, 2/6/2006

Washington University in St. Louis

Unix Process Context/State (1)

- All information describing a process; i.e., ...
- Process address space (text, heap, stack, etc.)
- Control information
 - » u area
 - Process Control Block (PCB) holds hardware context
 - » Contains hardware context when process no longer has CPU
 - Current system call args, return, and error status
 - Open file descriptor table
 - CPU usage and other accounting info (disk quota)
 - » proc structure (always memory resident)
 - PID and process group
 - Process state (e.g., RUNNING, SLEEPING)
 - Signal handling
 - Memory management information

15 - Ken Wong, 2/6/2006

Washington University in St. Louis

Unix Process Context/State (2)

- Identification (User ID, Group ID)
- Environment variables
- Hardware context
 - » Program Counter (PC): Address of next instruction
 - » Stack Pointer (SP): Address of top of stack
 - » Processor Status Word (PSW): Current/Previous mode, interrupt priority level, etc.
 - » Memory management registers
 - » Floating point unit (FPU) registers

16 - Ken Wong, 2/6/2006

Washington University in St. Louis

Unix UID And GID

- **UID and GID**
 - » Every user has a unique User ID (UID) in password file
 - » Every user belongs to at least one group, each with a unique Group ID (GID) in /etc/group file
 - » Affects file access permissions
- **The 'root' user or superuser**
 - » UID = 0
 - » Can access files regardless of file protection
 - » Can run restricted commands (e.g., dump, fsck)
- **Effective UID and GID**
 - » Normally same as UID and GID
 - » Change by su'ing to another user account; or Run a command that has setuid or setgid flag set
 - e.g., "-r-s-x-x root root /usr/bin/passwd"
 - `chmod 4611 myfile` or `chmod u=srwx,g=x,o=x myfile`

17 -Ken Wong, 2/6/2006

Washington University in St. Louis

Context Switching

- **Definition**
 - » The activity that occurs when the OS kernel switches between processes in an effort to share the CPU among competing, runnable processes
- **Actions**
 - » Save contents of hardware registers (PC, SP, ...)
 - » Load PC with location of kernel code
 - » Load hardware registers with new context if full context switch
- **Context-switch time is overhead**
 - » CPU utilization affected by quantum and context-switch time
 - Quantum: Time-slice (max CPU interval) given to a process
- **Types**
 - » *Voluntary*: Process blocks
 - » *Involuntary*: End of time quantum or higher-priority, runnable process gets control of CPU

18 -Ken Wong, 2/6/2006

Washington University in St. Louis

Process State (Alternative View)

- **Process Management**
 - » Registers (PC, SP, GRs)
 - » Program Status Word (PSW)
 - » Scheduling (priority)
 - » PID, PPID
 - » Usage accounting
 - » Signals, Timers
- **Memory Management**
 - » Pointers to text, data, and stack segments
 - » Page table
- **File Management**
 - » Directories (current, root)
 - » File descriptors
 - » UID, GID

19 -Ken Wong, 2/6/2006

Washington University in St. Louis

Process Termination

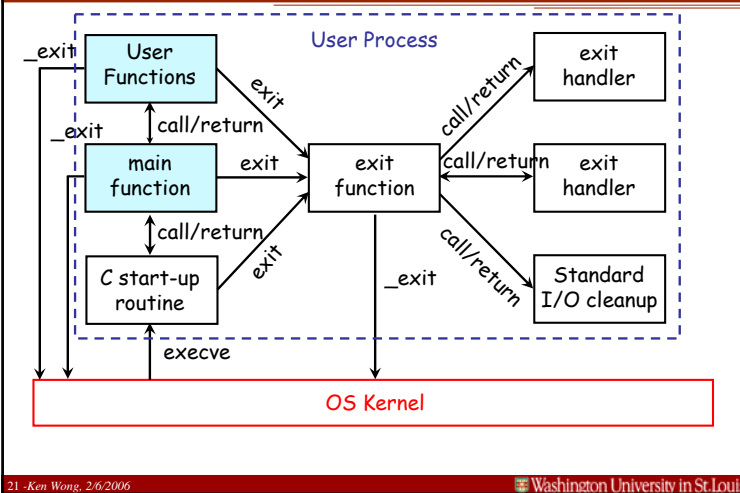
- **Termination Condition**
 - » Normal exit (see `exit(3)` and `_exit(2)`)
 - » Error exit
 - » Fatal error (e.g., illegal instr, bad addr, divide by zero)
 - » Killed by another process (*kill* system call)
- **Normal Unix Process Termination Activity**
 - » Close all files
 - » Save usage stats
 - » Make init process the parent of live children
 - » Change run state to ZOMBIE
 - » Release memory
 - » Send SIGCHLD to parent
 - » Wake up parent if asleep
 - » Call the scheduler

segmentation fault

20 -Ken Wong, 2/6/2006

Washington University in St. Louis

Unix C Program Startup/Termination



21 -Ken Wong, 2/6/2006

Washington University in St.Louis

Unix Zombie Processes

■ A Zombie Process

- » A process that has terminated but whose parent has not waited for it
- » When a process terminates, the OS kernel:
 - Discards all memory used by the process; closes all process' files
 - Keeps some info (PID, exit status, CPU time usage)
 - Provides info to parent when parent calls wait

■ ps output example

UID	PID	PPID	STAT	TT	TIME	COMMAND
7001	5608	5606	Ss	p3	0:03.88	tcsh
7001	7317	5608	S+	p3	0:00.02	testzombie
7001	7318	7317	Z+	p3	0:00.00	testzombie

```
if ((cpid = fork( )) == 0)    exit(0);
else {
    sleep(2);
    system("ps -l");
    // testzombie code
}
```

init process should harvest zombies

22 -Ken Wong, 2/6/2006

Washington University in St.Louis

waitpid

■ pid_t waitpid(pid_t pid, int *status, int options);

- » Unlike `wait(2)`, `waitpid` doesn't have to block
 - `wait(2)` blocks until one of its children terminates

■ pid values

- » `pid == -1`: Wait for any child to terminate
- » `pid > 0`: Wait for child whose PID equals `pid`
- » `pid == 0`: Wait for any child whose PGID equals the PGID of process `pid`
- » `pid < -1`: Wait for any child whose PGID = `abs(pid)`

■ *status

- » Contains exit status, signal number, and flags

■ options

- » Controls semantics of `waitpid`

23 -Ken Wong, 2/6/2006

Washington University in St.Louis

Examining waitpid 'status'

■ Contains exit status, signal number, and flags

■ WIFEXITED(status)

- » True if child terminated normally
- » Ex: `printf("exit = %d\n", WEXITSTATUS(status));`

■ WIFSIGNALED(status)

- » True if child terminated and didn't catch signal
- » Ex: `printf("signo = %d\n", WTERMSIG(s));`

■ WIFSTOPPED(status)

- » True if child is STOPPED
- » Ex: `printf("signo = %d\n", WSTOPSIG(s));`

24 -Ken Wong, 2/6/2006

Washington University in St.Louis

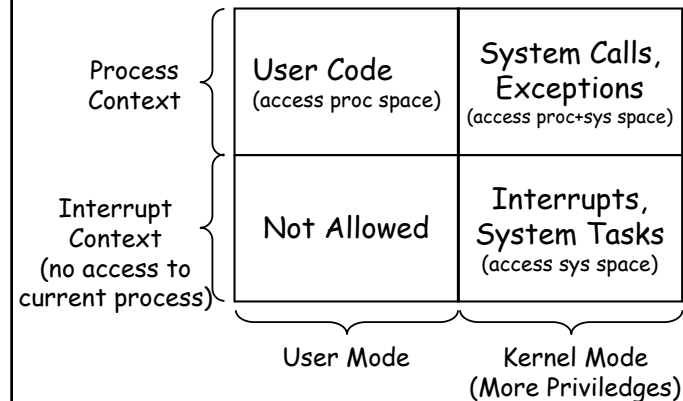
waitpid options

- options = 0
 - » No special handling
- options = WNOHANG
 - » Don't block if child is not available and return 0
 - » Ex: `rc = waitpid(pid, &status, WNOHANG);`
 if (rc == -1) { ... error ... }
 else if (rc == 0) { ... no child available ... }
 else { ... rc should equal pid ... }
- options = WUNTRACED
 - » Return status of child if stopped child and status has not already been returned (assumes job control support)

25 - Ken Wong, 2/6/2006

Washington University in St. Louis

Execution Context Versus Mode (1)



26 - Ken Wong, 2/6/2006

Washington University in St. Louis

Execution Context Versus Mode (2)

- Execution Mode
 - » User mode
 - Some parts of virtual address space can not be accessed
 - Some instructions (e.g., memory management) can not be executed
 - » Kernel mode
 - Can access kernel address space
 - Is a fixed part of the virtual address space of every process
 - System call puts user into kernel mode
- Unix kernel is reentrant
 - » Reentrant: Access to shared data must be coordinated
 - » Multiple processes can be "in the kernel" → Each process needs its own kernel stack
- Execution Context
 - » Process context: Kernel acts on behalf of user process
 - » Interrupt context: Kernel can't access context of current process

27 - Ken Wong, 2/6/2006

Washington University in St. Louis

Disk Interrupt and Multiprogramming

- Interrupt Vector
 - » One per I/O device class; usually stored in low memory
 - » Contains address of interrupt service routine for device
- Interrupt hardware pushes current PC, SP, PSW onto current kernel stack
- Hardware loads new PC from interrupt vector
- Software saves registers and sets up new stack
- Interrupt service routine updates process' I/O state
- Scheduler selects process to run next
- Software starts up selected process

28 - Ken Wong, 2/6/2006

Washington University in St. Louis

Interrupt Handling and Context

```
// Interrupt arrives to OS kernel
if (new IPL > current IPL) { // IPL = Interrupt Priority Level
    Create new context layer; // push old context onto kernel stack
    Push PC and PSW onto kernel stack (or interrupt stack);
    current IPL = new IPL;
    Call interrupt handler; // can even interrupt another handler
    IPL = IPL saved in PSW;
    if (no interrupts can be unblocked)
        Return to normal processing;
    else {
        Unblock highest blocked interrupt;
        // causes jump to interrupt handler
    }
} else {
    Save interrupt in special register;
    Block until IPL drops enough;
}
```

Setting IPL (4.3BSD):
 spl0 Enable all interrupts
 splbio Block disk interrupts
 splhigh Disable all interrupts
 splx Restore IPL to previous saved value

29 -Ken Wong, 2/6/2006

Washington University in St. Louis

Separate fork from exec

- Separating out fork from exec
 - » Greater flexibility
 - » Simplifies implementation
- Potential operations before 'exec'
 - » Redirect stdin, stdout, or stderr
 - » Close open files inherited from parent
 - » Change UID or process group
 - » Reset signal handlers
- Can get effect of copying parent without all of cost
 - » Copy-on-write:
 - Child shares virtual memory pages with parent but is read-only
 - A write to a shared page causes the actual page to be copied
 - » vfork:
 - Child temporarily uses parent's memory pages (dangerous!)

30 -Ken Wong, 2/6/2006

Washington University in St. Louis

Unix Process Creation

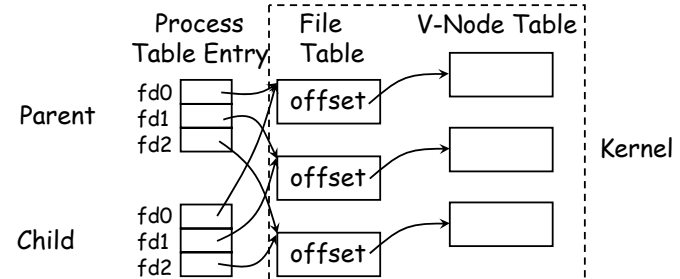
- Reserve swap space for child's data and stack
- Allocate new PID and proc structure
- Init child's proc structure and u area
 - » Copy UID, GID, and signal masks from parent
 - » Reset (zero) statistics
 - » Copy parent's registers to child's hardware context
- Setup virtual memory tables
 - » Child's data pages are the same as parent's except are read-only
- Mark child runnable and give to scheduler
- Return PID = 0 to child and child PID to parent

31 -Ken Wong, 2/6/2006

Washington University in St. Louis

Effect of fork

- Child gets a copy of its parent's memory
 - » BUT changes made by child are not reflected in parent ...
 - » EXCEPT a child can affect the I/O state of parent
 - File descriptors can point to same file table entry
 - » WARNING: Parent should "fflush(stdout)" before fork



32 -Ken Wong, 2/6/2006

Washington University in St. Louis

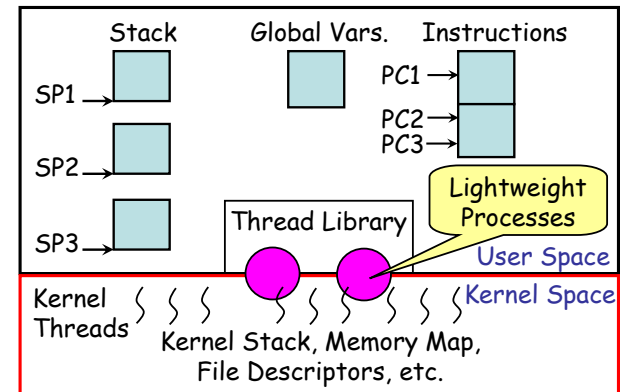
Exec Operations

- Find executable file (`argv[0]`)
- Verify that user may exec file
- Check that file header is a valid executable
- If file is *setuid* or *setgid*, change caller's effective UID and/or GID
- Copy exec args and env variables to kernel space
 - » Current user space will be destroyed
- Allocate heap and stack and free old process space
- Setup new address space
 - » Copy exec args and env variables into new space
- Reset all signal handlers
- Initialize hardware context

33 - Ken Wong, 2/6/2006

Washington University in St. Louis

A Modern Process (1)



34 - Ken Wong, 2/6/2006

Washington University in St. Louis

A Modern Process (2)

- Separate idea of execution from resource grouping
- Multithreading Support
 - » One or more threads of control (Program Counters)
 - » One stack for each thread of control
- Thread
 - » A unit of local dispatching (scheduling) and has priority
 - » Has a set of CPU registers
 - » Mapped to a lightweight process (LWP)
- LWPs are mapped to processors and globally scheduled
- Global variables are shared by all threads
- System state (file descriptors, working directory, etc) shared by threads

35 - Ken Wong, 2/6/2006

Washington University in St. Louis