Operating Systems

Week 1: Unix

Manpages

Command	Description
man man	describes how to use the man command
man [num] intro	displays information from the introduction to Chapter [num]
man [num] [topic]	displays information about [topic], which is found in Chapter [num]
man -k [keyword]	lists commands which have [keyword] in their synopsis

Man Pages Online
Man Pages Online 2

Convention	Description
bold text	type exactly as shown.
italic text	replace with appropriate argument.
[-abc]	any or all arguments within [ ] are optional.
-a|-b	options delimited by | cannot be used together.
argument ...	argument is repeatable.
[expression] ...	entire expression within [ ] is repeatable.

1. **User Commands**
   Executable programs or shell commands

Command	Description
cp	copy a file
mv	move or rename a file or directory
rm	remove (delete) a file
mkdir	make a new directory
ls	list the contents of a directory
cd	move to another directory
rmdir	remove (delete) a directory
pwd	print the name of the current working directory
chmod	change access permissions on files and directories
script	record a terminal session
telnet / ssh	Telephone Network protocol / login program
cat	print a file to the display
more / less	print a file to the display, a screenful at a time
lpr	send a file to the printer
grep	pattern match a string in files

2. **System Calls**
   Functions provided by the kernel
3. **Library Calls**
   Functions within program libraries
4. **Special Files**
   Usually found in /dev
5. **File Formats / Conventions**
6. **Games**
7. **Miscellaneous**
8. **System Admin Commands**
9. **Kernel Routine**

Program Development Process

Edit

Several built-in editors on the system (e.g. **vim**)

Compile

- GNU C Compiler (**gcc**)
- Compile with warnings (**-Wall**)
- Link math library (**-lm**)
- Standard I / O (**printf** / **scanf**)

Run

Successful compilation will produce an **a.out** file unless specified otherwise by **-o**

Debug

- GNU Debugger (**gdb**)
- Must use global command to enable debugging (**-g**)

Command	**Description**
b	set a breakpoint (your program stops at the specified program line or function name)
n	step to the next line in a program (steps over function calls)
s	step to the next line in a program (steps into function calls)
p	display (print) the value of a variable
watch	display (watch) the value of a variable whenever it changes
bt	display (backtrace) the current program execution stack

Week 2: What is an Operating System?

System and Application Programs

Operating system
Computer hardware

Mainframe: optimizes utilization of hardware
PC: support complex games, business applications and everything in between
Mobile: provide an environment for easy to use interface

Designed to be:
Easy
Convenient
Efficient

Computer System

1. Hardware
2. OS
3. Apps
4. Users

1. User's View

How is this device used?

- Single user / multiple users?
- Games /productivity
- User Interface, command line, touch, console, GUI

2. System View

- CPU time
- Memory space
- File storage space
- I/O devices

Manage execution of user programs to prevent
- Errors
- Improper use

Defining an OS
- Kernel program ✅
- System programs ✅
- Application Programs ❌

Computer Startup
Inital Program, bootstrap loads
- Systems processes / daemons
- Interupts

Storage Structure
CPU
RAM
Registers
Cache
Main memory
Solid state disk
^ cost ^ access speed

Multiprocessor systems

Asymmetric
One boss, assigns work

Symmetric
All peers no boss

Week 3: Processes

A program in execution
- Program itself is a **passive** entity
- Process is an **active** entity

Program becomes a process when it runs

Processes can be…
- **I / O bound:** spends more time doing I / O than doing computations
- **CPU bound:** spends more time doing computations than waiting for I /O

Statuses

1. **New**
   process is being created
2. **Running**
   instructions are executed
3. **Waiting**
   the process is waiting for some event to occur
4. **Ready**
   The process is waiting to be assigned to a processor
5. **Terminated**
   The process has finished execution

Process Control Block (PCB)

CPU Scheduling Info
Memory Management Info
Account Info
I/O status info

Schedulers

1. **Short-term Scheduler**
   Runs frequently, needs to make decisions faster
2. **Long-term Scheduler**
   Runs fairly infrequently, can take longer times to make decisions, not available on all computers

Context switch

What type of hardware might help speed up context switch

Creations

How many children can you fork from a process?

Only the parent can terminate a child process

Termination

Zombie Process
Still out there taking up resources until parent calls wait
Orphan
Parent terminates without wait call

Pipe Creation

Write 1
Read 0

System Calls

fork()

Description: Creates a new process by duplicating the calling process.
Arguments: None.
Return Values:

- 0 in the child process.
- Child process ID (PID) in the parent process.
- -1 on failure.

Example:

```
pid_t pid = fork();
if (pid == 0) {
    // Child process code
} else if (pid > 0) {
    // Parent process code
} else {
    // Error handling
}
```

exec()

Description: Replaces the current process image with a new process image, executing a specified program.
Options: Certainly! Here's the format you requested:

- **execv()**: Executes a program specified by a pathname with a vector of arguments.

- **execve()**: Executes a program with a vector of arguments and allows specifying environment variables.

- **execvp()**: Executes a program by searching the PATH for the executable, using a vector of arguments.

- **execvl()**: Executes a program with a list of arguments and a specified pathname.

- **execle()**: Executes a program with a list of arguments and allows specifying environment variables.

- **execlp()**: Executes a program by searching the PATH for the executable, using a list of arguments.

Arguments:

- Path to the executable file.
- Array of arguments (first element is the program name, last element must be `NULL`).

Return Values:

- `0` on success (does not return to the caller).
- `-1` on failure.

Example:

```
char *args[] = {"ls", "-l", NULL};
if (execvp("ls", args) == -1) {
    perror("execvp failed");
}
```

wait()

Description: Waits for a child process to terminate and retrieves its exit status.
Arguments: Pointer to an integer to store the child's exit status.
Return Values:

- Process ID of the terminated child on success.
- `-1` on failure.

Example:

```
int status;
pid_t pid = wait(&status);
if (pid > 0) {
    // Check status of child process
    if (WIFEXITED(status)) {
        printf("Child exited with status %d\n", WEXITSTATUS(status));
    }
} else {
    perror("wait failed");
}
```

exit()

Description: Terminates the calling process and returns a status code to the operating system.

Arguments: Integer status code (0 for success, non-zero for error).

Return Values: None (the process terminates).

Example:

```
if (error_occurred) {
    exit(1);  // Terminate with error status
}
exit(0);  // Normal termination
```

Functions

- **fgets()**: Reads a line of text from the standard input into a buffer, including newline characters.
- **strcspn()**: Finds the length of the initial segment of a string not containing any characters from another string (used to remove newline).
- **strcmp()**: Compares two strings and returns 0 if they are equal.
- **perror()**: Prints a description for the last error encountered.
- **wait4()**: Waits for a child process to terminate and retrieves resource usage statistics.
- **exit()**: Terminates the current process and returns a status code to the operating system.
- **wait()**: Waits for the termination of a child process and retrieves its exit status.
- **getrusage()**: Retrieves resource usage statistics for the calling process or its children.

Week 4: Threads

Benefits

- Responsiveness

- Resource Sharing
- Economy
  - Solaris makes threads 30x faster than
  - Solaris OS context switching is 5 times faster with threads
- Scalability
- Can run threads on multiple processors

Multicore Programming

Parallelism - can perform more than one task simultaneously
Concurrency - allowed all tasks to make progress by rapidly switching between processes on a single CPU
Amdahl's Law how much faster our speedup will be if we run things in parallel- multicore so

Programming Challenges

1. Identify tasks:
2. Balance
3. Data Splitting
4. Data Dependency
5. Testing and Debugging

Parallelism

Data Parallelism
Task Parallelism

Types

Models

Many-to-One

Every student is sharing the same bicicle

One-to-One

Everyone has a bike, even the people that don't need a bike
Every thread we fire off, we need a kernel thread to match it

Many-to-One

Pool of bikes, anyone who would like to use them can

Thread Libraries

- Pthreads
- Windows

Strategies

- Asynchronous
- Synchronous

Week 5: Critical Section

Critical Section

Mutual Exclusion
Progress - answered everyone elses questions before Don's
Bounded Waiting - bound must exist so other proce

Handling

- Preemptive
- Non-preemptive

Peterson's Solution

- Two Process solution
- Share too variables
    - Flag
    - Turn

Can't just use 1 variable because the registers might not be updated

Test and Set Lock

Test and set function(my lock) my lock can also be called target because it is the target we would like to aquire
returnValue = value stored at my lock
My lock = true
Return returnValue

Satisfies Mutual Exclusion and progress but not bounded waiting

Shared Memory

shmget()

- **Description**: Creates a shared memory segment and returns an ID (shmid) that processes can use to access the segment.
- **Example**:

```
int shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
```

shmat()

- **Description**: Attaches a shared memory segment to the address space of the calling process, allowing it to read/write to the shared memory.
- **Example**:

```
void *shared_memory = shmat(shmid, NULL, 0);
```

shmdt()

- **Description**: Detaches a shared memory segment from the address space of the calling process.
- **Example**:

```
shmdt(shared_memory);
```

shmctl()

- **Description**: Performs control operations on a shared memory segment, such as retrieving segment info or removing it.
- **Example**:

```
shmctl(shmid, IPC_RMID, NULL);
```

Week 6: Critical Section

Compare and Swap

Like test and set lock but lock is now an integer

Mutex Lock

Acquire and Release must be atomic

Semaphore

Avoid busy waiting (and wasting cpu cycles) - golf cart example
Sleep at the club table and gets woken up when golf cart is available
Wait = p
Signal = v

Counting semaphore: more than one process to access a lock
Binary semaphore: single process to access the lock

```
typedef struct
int value;
struct process *list;
} semaphore;
```

Week 7: Scheduling

Round Robin Scheduling

Next Process in queue gets [quantum] time on CPU
Round Robin scheduling gives each process an equal share of CPU time (known as the time quantum). Once a process's quantum expires, it is moved to the back of the queue, ensuring fairness and time-sharing.

Time Quantum 2

With a time quantum of 2, each process gets to run for 2 units of time before being swapped out. This ensures rapid switching between processes but may result in frequent context switching for longer processes.

Time Quantum 4

In this variant, the time quantum is 4, which means processes get to run longer before being preempted. This can reduce context switching but may delay shorter processes that are stuck behind longer ones in the queue.

Priority Scheduling

Highest Priority gets CPU
In Priority Scheduling, processes are assigned priorities, and the process with the highest priority gets the CPU. If two processes have the same priority, other factors such as arrival time may be used to break the tie. Lower-priority processes have to wait until higher-priority ones finish execution.

Shortest Job First (SJF) Scheduling

Shortest Job First selects the process with the shortest burst time for execution, either preemptively or non-preemptively. This approach minimizes waiting time for shorter tasks.

Preemptive (Shortest Remaining Time First)

With Interrupt
In the preemptive version of SJF, also known as Shortest Remaining Time First (SRTF), the system always selects the process with the shortest remaining burst time. If a new process arrives with a shorter remaining time than the currently running process, it preempts the running process.

Non-Preemptive

No Interrupts
In non-preemptive SJF, once a process starts execution, it runs to completion. The next shortest job is selected only after the current process finishes, meaning no process can interrupt another once it starts.

Week 8: Midterm (Review)