Processes and Threads

What are processes?
How does the operating system manage them?

Nick Urbanik

nicku@nicku.org

A computing department

Introduction

Multitasking

Start of Process

Scheduler

Process States

top

Process Control Blocks

System Calls

A shell program

IPC

Threads

Race Condition

Synchronisation

Summary and References

What is a process?

- A *process* is a program in execution
- Each process has a *process ID*
- In Linux,

```
$ ps ax
```

- prints one line for each process.
- A program can be executed a number of times simultaneously.
  - Each is a separate process.

What is a process? — 2

- A process includes current values of:
  - Program counter
  - Registers
  - Variables
- A process also has:
  - The program code
  - It's own address space, independent of other processes
  - A user that owns it
  - A group owner
  - An *environment* and a *command line*
- This information is stored in a *process control block*, or *task descriptor* or *process descriptor*
  - a data structure in the OS, in the *process table*
  - See slides starting at §34.

What is a thread?

- A *thread* is a lightweight process
  - Takes less CPU power to start, stop
- Part of a single process
- Shares address space with other threads in the same process
- Threads can share data more easily than processes
- Sharing data requires *synchronisation*, i.e., locking — see slide 95.
- This shared memory space can lead to complications in programming:

> "Threads often prevent abstraction. In order to prevent deadlock. you often need to know how and if the library you are using uses threads in order to avoid deadlock problems. Similarly, the use of threads in a library could be affected by the use of threads at the application layer." – *David Korn*
>
> See page 180, ESR in references, §112.

Program counter

- The code of a process occupies memory
- The Program counter (PC) is a CPU register
- PC holds a memory address. . .
- . . . of the next instruction to be fetched and executed

Environment of a process

- The *environment* is a set of names and values
- Examples:

```
PATH=/usr/bin:/bin:/usr/X11R6/bin
HOME=/home/nicku
SHELL=/bin/bash
```

- In Linux shell, can see environment by typing:

```
$ set
```

Permissions of a Process

- A process executes with the permissions of its owner
  - The owner is the user that starts the process
- A Linux process can execute with permissions of another user or group
- If it executes as the owner of the program instead of the owner of the process, it is called *set user* ID
- Similarly for *set group* ID programs

Multitasking

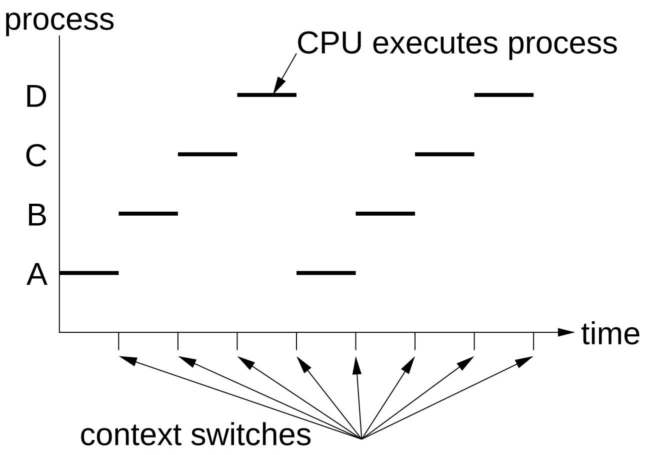
- Our lab PCs have one main CPU
  - But multiprocessor machines are becoming increasingly common
  - Linux 2.6.x kernel scales to 16 CPUs
- How execute many processes "at the same time"?

Multitasking — 2

- CPU rapidly switches between processes that are “ready to run”
- Really: only one process runs at a time
- Change of process called a *context switch*
  - See slide §36
- With Linux: see how many context switches/second using `vmstat` under “`system`” in column “`cs`”

Multitasking — 3

- This diagram shows how the scheduler gives a “turn” on the CPU to each of four processes that are ready to run



Birth of a Process

- In Linux, a process is born from a `fork()` system call
  - A *system call* is a function call to an operating system service provided by the kernel
- Each process has a *parent*
- The parent process calls `fork()`
- The child inherits (*but cannot change*) the parent environment, open files
- Child is *identical* to parent, except for return value of `fork()`.
  - Parent gets child’s process ID (PID)
  - Child gets 0

Process tree

- Processes may have parents and children
- Gives a family tree
- In Linux, see this with commands:

```
$ pstree
```

or

```
$ ps axf
```

Scheduler

- OS decides when to run each process that is ready to run ("runable")
- The part of OS that decides this is the *scheduler*
- Scheduler aims to:
  - Maximise CPU usage
  - Maximise process completion
  - Minimise process execution time
  - Minimise waiting time for ready processes
  - Minimise response time

When to Switch Processes?

- The scheduler may change a process between executing (or running) and ready to run when any of these events happen:
  - clock interrupt
  - I/O interrupt
  - Memory fault
  - trap caused by error or exception
  - system call
- See slide §17 showing the running and ready to run process states.

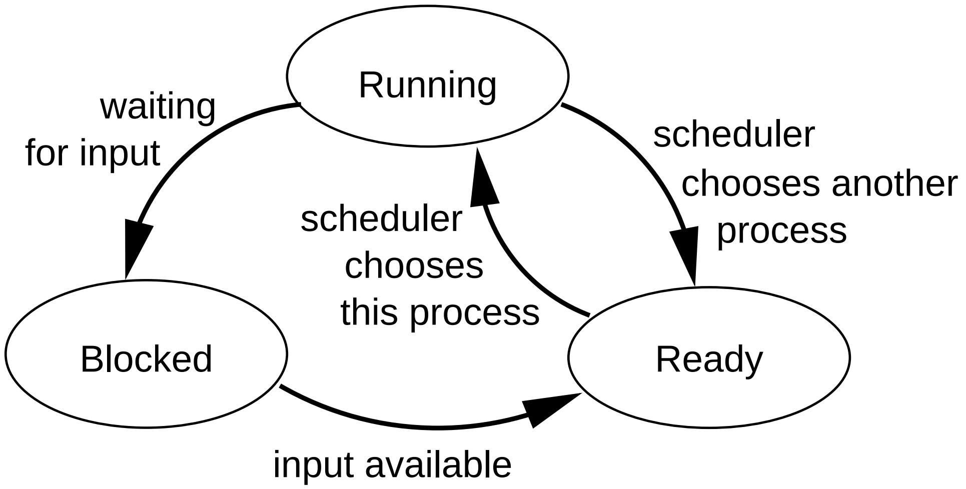
Scheduling statistics: `vmstat`

- The "`system`" columns give statistics about *scheduling*:
  - "`cs`" — number of context switches per second
  - "`in`" — number of interrupts per second
- See slide §36, `man vmstat`

Interrupts

- Will discuss interrupts in more detail when we cover I/O
- An *interrupt* is an event (usually) caused by hardware that causes:
  - Saving some CPU registers
  - Execution of *interrupt handler*
  - Restoration of CPU registers
- An opportunity for scheduling

Process States


Running
waiting for input
scheduler chooses another process
scheduler chooses this process
Blocked
Ready
input available

What is Most Common State?

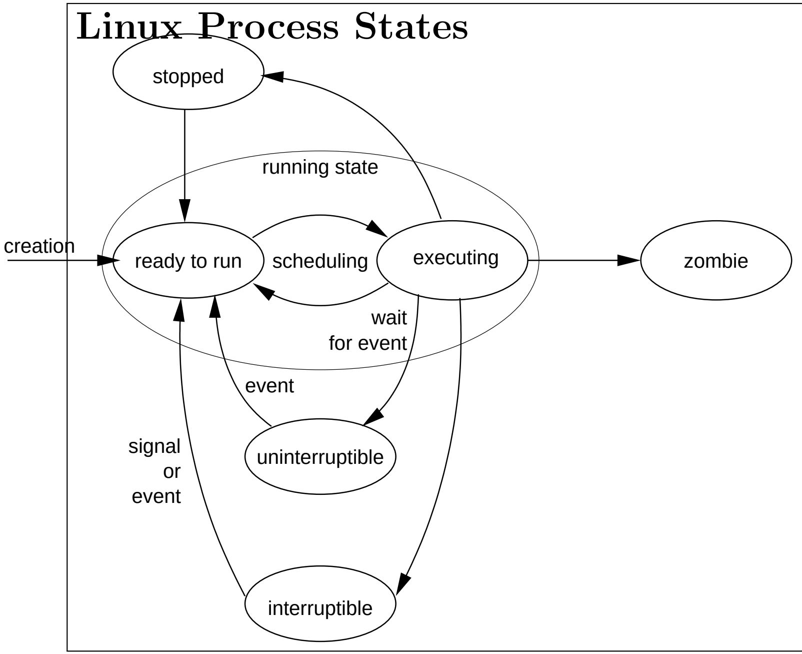
- Now, my computer has 160 processes.
- How many are running, how many are ready to run, how many are blocked?
- What do you expect is most common state?

Most Processes are Blocked

```
9:41am  up 44 days, 20:12,  1 user,  load average: 2.02, 2.06, 2.13
160 processes: 145 sleeping, 2 running, 13 zombie, 0 stopped
```

- Here you see that most are sleeping, waiting for input!
- Most processes are "*I/O bound*"; they spend most time waiting for input or waiting for output to complete
- With one CPU, only one process can actually be running at one time
- However, surprisingly few processes are ready to run
- The *load average* is the average number of processes that are in the ready to run state.
- In output from the top program above, see over last 60 seconds, there are 2.02 processes on average in RTR state

Linux Process States



Linux Process States — 2

- *Running* — actually contains two states:
  - *executing*, or
  - *ready to execute*
- *Interruptable* — a blocked state
  - waiting for event, such as:
    - end of an I/O operation,
    - availability of a resource, or
    - a signal from another process
- *Uninterruptable* — another blocked state
  - waiting directly on hardware conditions
  - will not accept any signals (even `SIGKILL`)

Linux Process States — 3

- *Stopped* — process is halted
  - can be restarted by another process
  - e.g., a debugger can put a process into stopped state
- *Zombie* — a process has terminated
  - but parent did not `wait()` for it (see slide 33)

Process States: `vmstat`

- The "`procs`" columns give info about process states:
- "`r`" — number of processes that are in the *ready to run* state
- "`b`" — number of processes that are in the *uninterruptable* blocked state

Tools for monitoring processes

- Linux provides:
- vmstat
  - Good to monitor over time:

    `$ vmstat 5`
- procinfo
  - Easier to understand than `vmstat`
  - Monitor over time with

    `$ procinfo -f`
- View processes with top — see slides 27 to §30
- The system monitor sar shows data collected over time: See `man sar`; investigate `sar -c` and `sar -q`
- See the utilities in the `procps` software package. You can list them with

```
$ rpm -ql procps
```

ps
free
pgrep
pkill
pmap
skill
slabtop
snice
tload
top
uptime
vmstat
w
watch

Monitoring processes in Win 2000

- Windows 2000 provides a tool:
- Start → Administrative Tools → Performance.
- Can use this to monitor various statistics

Process Monitoring with `top`

Process Monitoring — `top`

```
   08:12:13  up 1 day, 13:34,  8 users,  load average: 0.16, 0.24, 0.49
111 processes: 109 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total    0.0%    0.0%    3.8%   0.0%     0.0%    0.0%   96.1%
Mem:   255608k av,  245064k used,   10544k free,       0k shrd,   17044k buff
       152460k active,              63236k inactive
Swap: 1024120k av,  144800k used,  879320k free                  122560k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 1253 root      15   0 73996  13M 11108 S     2.9  5.5  19:09   0 X
 1769 nicku     16   0  2352 1588  1488 S     1.9  0.6   2:10   0 magicdev
23548 nicku     16   0  1256 1256   916 R     1.9  0.4   0:00   0 top
    1 root      16   0   496  468   440 S     0.0  0.1   0:05   0 init
    2 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kapmd
    4 root      34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd/0
    6 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 bdflush
    5 root      15   0     0    0     0 SW    0.0  0.0   0:11   0 kswapd
```

`top`: load average

```
08:12:13  up 1 day, 13:34,  8 users,  load average: 0.16, 0.24, 0.49
```

- *load average* is measured over the last minute, five minutes, fifteen minutes
- Over that time is the average number of processes that are *ready to run*, but which are *not executing*
- A measure of how "busy" a computer is.

top: process states

```
111 processes: 109 sleeping, 1 running, 1 zombie, 0 stopped
```

sleeping Most processes (109/111) are sleeping, waiting for I/O

running This is the number of processes that are both ready to run and are executing

zombie There is one process here that has terminated, but its parent did not `wait()` for it.

- The `wait()` system calls are made by a parent process, to get the `exit()` status of its child(ren).
- This call removes the *process control block* from the *process table*, and the child process does not exist any more. (§34)

stopped When you press Control-z in a shell, you will increase this number by 1

top: Processes and Memory

```
 PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
1253 root       15   0 73996  13M 11108 S     2.9  5.5  19:09   0 X
```

SIZE This column is the total size of the process, including the part which is swapped (paged out) out to the swap partition or swap file

Here we see that the process X uses a total of 73,996 Kb, i.e., 73,996 × 1024 bytes $\approx$ 72MB, where here 1MB = 2^{20} bytes.

RSS The *resident set size* is the total amount of RAM that a process uses, including memory shared with other processes. Here X uses a total of 13MB RAM, including RAM shared with other processes.

SHARE The amount of *shared* memory is the amount of RAM that this process shares with other processes. Here X shares 11,108 KB with other processes.

We can see that the total amount of RAM used exclusively by one process is $\text{rss} - \text{share}$. Here we see that X uses about $13 \times 2^{20} - 11{,}108 \times 2^{10} \approx$ 2 MB

Virtual Memory: suspended processes

- With memory fully occupied by processes, could have all in blocked state!
- CPU could be completely idle, but other processes waiting for RAM
- Solution: *virtual memory*
  - ○ will discuss details of VM in memory management lecture
- Part or all of process may be saved to swap partition or swap file

Suspended Processes

- Could add more states to process state table:
  - ready and suspended
  - blocked and suspended

Process Control Blocks

The Process Table

Data structure in OS to hold information about a process

OS Process Control Structures

- Every OS provides *process tables* to manage processes
- In this table, the entries are called *process control blocks* (PCBs), *process descriptor*s or *task descriptor*s. We will use the abbreviation PCB.
- There is one PCB for each process
- in Linux, PCB is called `task_struct`, defined in `include/linux/sched.h`
  - In a Fedora Core or Red Hat system, you will find it in the file `/usr/src/linux-2.*/include/linux/sched.h` if you have installed the `kernel-source` software package

What is in a PCB

- In slide §3, we saw that a PCB contains:
  - a process ID (PID)
  - *process state* (i.e., executing, ready to run, sleeping waiting for input, stopped, zombie)
  - *program counter*, the CPU register that holds the address of the next instruction to be fetched and executed
  - The value of other *CPU registers* the last time the program was switched out of executing by a *context switch* — see slide §36
  - scheduling priority
  - the *user* that owns the process
  - the *group* that owns the process
  - pointers to the *parent process*, and *child processes*
  - Location of *process's data* and *program code* in memory
  - List of *allocated resources* (including open *files*)
- PCB holds the values as they were when process was last switched out of executing by a *context switch* — see slide §36

Context Switch

- OS does a *context switch* when:
  - stop current process from executing, and
  - start the next ready to run process executing on CPU
- OS saves the *execution context* (see §37) to its PCB
- OS loads the ready process's execution context from its PCB
- *When* does a context switch occur?
  - When a process *blocks*, i.e., goes to sleep, waiting for input or output (I/O), or
  - When the scheduler decides the process has had its turn of the CPU, and it's time to schedule another ready-to-run process
- A context switch must be as *fast as possible*, or multitasking will be too slow
  - Very fast in Linux OS

Execution Context

- Also called *state of the process* (but since this term has two meanings, we avoid that term here), *process context* or just *context*
- The *execution context* is all the data that the OS must save to stop one process from executing on a CPU, and load to start the next process running on a CPU
- This includes the content of all the CPU registers, the location of the code, . . .
  - Includes most of the contents of the process's PCB.

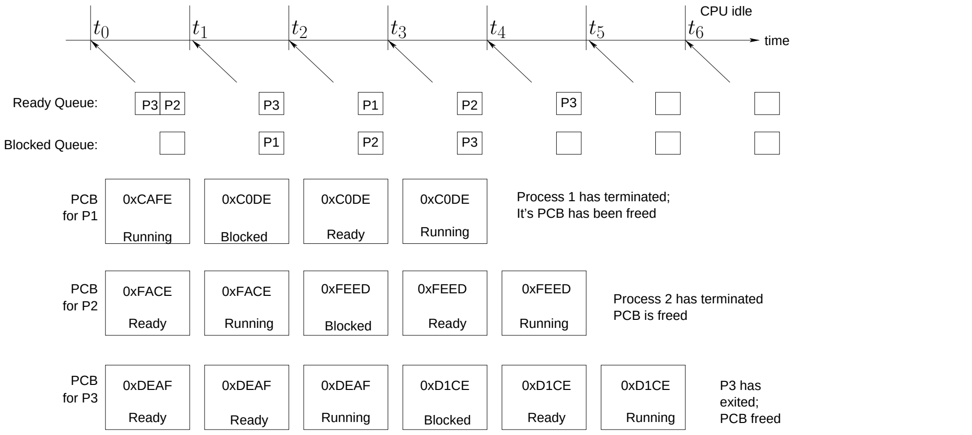
Program Counter in PCB

- What value is in the program counter in the PCB?
- If it is *not* executing on the CPU,
  - The address of the next CPU instruction that *will be* fetched and executed the next time the program starts executing
- If it *is* executing on the CPU,
  - The address of the first CPU instruction that *was* fetched and executed when the process began executing at the last context switch (§36)

Process Control Blocks—Example

- The diagram in slide §40 shows three processes and their process control blocks.
- There are seven snapshots t_0, t_1, t_2, t_3, t_4, t_5 and t_6 at which the scheduler has changed process (there has been a context switch—§36)
- On this particular example CPU, all I/O instructions are 2 bytes long
- The diagram also shows the queue of processes in the:
  - *Ready queue* (processes that are ready to run, but do not have a CPU to execute on yet)
  - *Blocked*, or *Wait queue*, where the processes have been blocked because they are waiting for I/O to finish.

PCB Example: Diagram



	t_0	t_1	t_2	t_3	t_4	t_5	t_6
Ready Queue:	P3 P2	P3	P1	P2	P3		
Blocked Queue:		P1	P2	P3			
PCB for P1	0xCAFE Running	0xC0DE Blocked	0xC0DE Ready	0xC0DE Running	Process 1 has terminated; It's PCB has been freed		
PCB for P2	0xFACE Ready	0xFACE Running	0xFEED Blocked	0xFEED Ready	0xFEED Running	Process 2 has terminated PCB is freed	
PCB for P3	0xDEAF Ready	0xDEAF Ready	0xDEAF Running	0xD1CE Blocked	0xD1CE Ready	0xD1CE Running	P3 has exited; PCB freed

PCB Example — Continued

- In slide §40,
  - The times t_0, t_1, t_2, t_3, t_4, t_5 and t_6 are when the scheduler has selected another process to run.
  - Note that these time intervals are *not equal*, they are just the points at which a scheduling change has occurred.
- Each process has stopped at one stage to perform I/O
  - That is why each one is put on the *wait queue* once during its execution.
- Each process has performed I/O once

What is the address of I/O instructions?

- We are given that all I/O instructions *in this particular example* are two bytes long (slide §39)
  - We can see that when the process is sleeping (i.e., blocked), then the program counter points to the instruction *after* the I/O instruction
  - So for process P1, which blocks with program counter $\text{PC} = \text{C0DE}_{16}$, the I/O instruction is at address $\text{C0DE}_{16} - 2 = \text{C0DC}_{16}$
  - for process P2, which blocks with program counter $\text{PC} = \text{FEED}_{16}$, the I/O instruction is at address $\text{FEED}_{16} - 2 = \text{FEEB}_{16}$
  - for process P3, which blocks with program counter $\text{PC} = \text{D1CE}_{16}$, the I/O instruction is at address $\text{D1CE}_{16} - 2 = \text{D1CC}_{16}$

Process System Calls

How the OS controls processes

How you use the OS to control processe

Major process Control System Calls

- `fork()` — start a new process
- `execve()` — replace calling process with machine code from another program file
- `wait()`, `waitpid()` — parent process gets status of its' child after the child has terminated, and cleans up the process table entry for the child (stops it being a *zombie*)
- `exit()` — terminate the current process

File I/O system calls: a sidetrack

```
#include <unistd.h>
ssize_t read( int filedes, void *buf,
              size_t nbytes );
```

- returns number of bytes read, 0 at end of file, −1 on error

```
ssize_t write( int filedes, void *buf,
               size_t nbytes );
```

- returns number of bytes written, else −1 on error
- Note: these are *unbuffered*, that is, they have effect "immediately".
- This is different from `stdio.h` functions, which are buffered for efficiency.

Process IDs and `init`

- Every process has a process ID (PID)
- process 0 is the scheduler, part of kernel
- process 1 is `init`, the parent of *all* other processes
  - a normal user process, not part of kernel
  - program file is `/sbin/init`
- All other processes result from `init` calling the `fork()` system call
- This is the *only* way a new process is created by the kernel

SUID, SGID and IDs

- Every process has *six* or more IDs associated with it
- UID and GID of person who executes program file:
  - real user ID, real group ID
- IDs used to calculate permissions:
  - Effective UID, Effective GID
- IDs saved when use `exec()` system call:
  - Saved set-user-ID, saved set-group-ID
  - idea is can drop special privileges and return to executing with real UID and real GID when privilege is no longer required

Other system calls: getting process info

```
#include <sys/types.h>

#include <unistd.h>
```

- `pid_t getpid(void);` returns PID of calling process
- `pid_t getppid(void);` returns PID of parent
- `uid_t getuid(void);` returns real user ID of process
- `uid_t geteuid(void);` returns effective user ID of process
- `gid_t getgid(void);` returns real group ID of process
- `gid_t getegid(void);` returns effective group ID of process

fork(): what it does

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- returns 0 in child
- returns PID of child in parent
- returns -1 if error

Using fork(): pseudocode

```
if ( ( pid = fork() ) < 0 )
        fork_error has happened
else if ( pid == 0 )                                  /* I am the child */
        do things the child process should do
else                                                  /* I am the parent */
        do things the parent should do
```

Simple fork() Example (no Checking)

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        int pid = fork();
        printf( "PID is %d\n", pid );
        if ( pid == 0 )
                printf( "I’m the child\n" );
        else
                printf( "I’m the parent\n" );
}
```

An example using fork()

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int glob = 6;
char buf[] = "a write to standard output\n";
int main( void )
{
        int var = 88;                               /* local variable on the stack */
        pid_t pid;
        if ( write( STDOUT_FILENO, buf, sizeof (buf) − 1 )
              ≠ sizeof( buf ) − 1 ) {
                fprintf( stderr, "write error" );
                exit( 1 );
        }
```

Example using fork()—(contd.)

```
        printf( "before fork\n" );
        if ( ( pid = fork() ) < 0 ) {
                fprintf( stderr, "fork error\n" );
                exit( 1 );
        } else if ( pid == 0 ) {                                    /* child */
                ++glob;
                ++var;
        } else
                sleep( 2 ); /* parent */
        printf( "pid = %d, glob = %d, var = %d\n",
                getpid(), glob, var );
        exit( 0 );
}
```

Output of fork-example.c:

```
$ gcc -o fork-example fork-example.c
$ ./fork-example
a write to standard output
before fork
pid = 7118, global = 7, var = 89 child's vars changed
pid = 7117, global = 6, var = 88 parent's copy not changed
```

Running fork-example again

```
$ ./fork-example > tmp.out
$ cat tmp.out
a write to standard output
before fork
pid = 7156, global = 7, var = 89
before fork
pid = 7155, global = 6, var = 88
```

Why two "before fork" messages?

- write() system call not buffered
- write() called before fork(), so one output
- printf() is buffered
  - line buffered if connected to terminal
  - fully buffered otherwise; parent and child both have a copy of the unwritten buffer when redirected
- exit() causes both parent and child buffers to flush

So what does this show?

- It shows that the child is an exact copy of the parent, with all
- variable values,
- buffers,
- open files,. . .
- All are inherited by the child

Running another program — `exec()`

- To run another program file
- first call `fork()` to create a child process
- child calls `exec()` to replace current copy of parent with a totally new program in execution

`execve()` system call

```
#include <unistd.h>
int execve( const char *filename,
            char *const argv[],
            char *const envp[] );
```

- executes the program filename, replaces current process
- Passes the command line in `argv[]`
- passes the environment variables in `envp[]`
- Does not return, unless error, when returns with −1
- Usually called through library `exec*()` calls — see `man 3 exec`

`fork()` — `exec()` Example

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        int pid = fork();
        printf( "PID is %d\n", pid );
        if ( pid == 0 )
                printf( "I'm the child\n" );
        else
                printf( "I'm the parent\n" );
}
```

Using `execl()`

```
int execl( const char *path,
           const char *arg, ... );
```

- Parameter number:
  1. gives full path of the program file you want to execute
  2. gives name of the new process
  3. specifies the command line arguments you pass to the program
  4. last is a `NULL` pointer to end the parameter list.
- We must always put a `NULL` pointer at the end of this list.

`print.c`: a program we call

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
        // argv[0] is the program name
        int num = atoi( argv[1] );
        int loops = atoi( argv[2] );
        int i;
        for ( i = 0; i < loops; ++i )
                printf( "%d ", num );

}
```

Calling `./print` using `execl()`

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        printf( "hello world\n" );
        int pid = fork();
        printf( "fork returned %d\n", pid );
        if ( pid == 0 )
                execl( "./print", "print", "1", "100", NULL );

        else
                execl( "./print", "print", "2", "100", NULL );
}
```

`vfork()` sytem call

- A lightweight `fork()`
- Designed for running `execvp()` straight after
  - modern Linux `fork()` is very efficient when call `exec*()`
- Child does not contain an exact copy of parent address space;
- child calls `exec()` or `exit()` after `fork()`
- parent is suspended till child calls `fork()` or `exit()`

`wait()`, `waitpid()` system calls

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait( int *status );

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

- return process ID if OK, 0, or −1 on error

`wait()`, `waitpid()` system calls

- `wait()` can block caller until child process terminates
- `waitpid()` has option to prevent blocking
- `waitpid()` can wait for a specific child instead of the first child
- if child has terminated already (it's a *zombie*), wait returns immediately, cleaning up the process table data structures for the child

Part of Simple Shell Program

```
int main( int argc, char **argv )
{
        char *prog_name = basename( *argv );
        print_prompt( prog_name );
        read_command();
        for ( ;; ) {
                int pid = fork();
                if ( pid == 0 ) {
                        execvp( args[ 0 ], args );
                }
                wait( NULL );
                print_prompt( prog_name );
                read_command();
        }
}
```

Windows and Processes

- Windows provides a Win32 API call to create a process: `CreateProcess()`
- Creates a new process, loads program into that process
- `CreateProcess()` takes *ten* parameters

Windows and Processes — 2

- Win32 uses *handles* for almost all objects such as files, pipes, sockets, processes and events
- handles can be inherited from parent
- No proper parent-child relationship
  - caller of `CreateProcess()` could be considered as parent
  - but child cannot determine it's parent

CreateProcess() prototype

- CreateProcess() is *much* more complicated than

  `pid_t fork( void );`

- Four of the parameters point to structs, e.g.,
  - LPSTARTUPINFO points to a struct with 4 members
  - LPPROCESS_INFORMATION points to a struct with 18 members!

```
BOOL CreateProcess (
        LPCTSTR lpApplicationName, // pointer to executable
module
        LPTSTR lpCommandLine, // pointer to command line
string
        LPSECURITY_ATTRIBUTES lpProcessAttrib, // process
security
        LPSECURITY_ATTRIBUTES lpThreadAttrib, // thread
security
        BOOL bInheritHandles, // handle inheritance flag
        DWORD dwCreationFlags, // creation flags
        LPVOID lpEnvironment, // pointer to new environment
block
        LPCTSTR lpCurrentDirectory, // pointer to current dir
name
        LPSTARTUPINFO lpStartupInfo, // pointer to
STARTUPINFO
        LPPROCESS_INFORMATION lpProcessInformation //
pointer to
                                                       //
PROCESS_INFORMATION
);
```

CreateProcess()

- Can Specify Program in either 1st or 2nd parameter:
  - first: location of program to execute
  - second: command line to execute
- Creation flags:
  - if 0, runs in existing window

Example: CreateProcess()

```
#include <windows.h>
#include <stdio.h>

void main() {
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
                memset( &si, 0, sizeof( si ) );
        si.cb = sizeof(si);
        if ( ! CreateProcess( NULL,
                              "..\\..\\print\\Debug\\print.exe 5 100",
                              NULL, NULL, TRUE, 0, NULL, NULL, &si,
&pi) )
                fprintf( stderr, "CreateProcess failed with %d\n",
GetLastError() );
        WaitForSingleObject( pi.hProcess, INFINITE );
        CloseHandle( pi.hProcess );
        CloseHandle( pi.hThread );
}
```

Processes in Linux, Unix, Windows

- Linux often provides 2 or more processes per application
- Example: apache web server parent process watches for connections, one child process per client
- Linux processes have much less overhead than in Windows
- `fork()` — `exec()` very efficient
- POSIX threads are very efficient, and faster than `fork()` — `exec()`

- Windows have one process per application, but often 2 or more threads
- Windows `CreateProcess()` takes more time than `fork()` — `exec()`
- `CreateThread()` takes very much less time than `CreateProcess()`

IPC

Inter Process Communication

How Processes can Talk to Each Other

Problem with Processes

- Communication!
- Processes cannot see the same variables
- Must use ***Inter Process Communication*** (IPC)
- IPC Techniques include:
  - pipes, and named pipes (FIFOs)
  - sockets
  - messages and message queues
  - shared memory regions
- All have some overhead

Interprocess Communication (IPC)

- *Pipe* — circular buffer, can be written by one process, read by another
  - related processes can use unnamed pipes
    - used in shell programming, e.g., the vertical bar '|' in

      ```
      $ find /etc | xargs file
      ```
  - unrelated processes can use *named pipes* — sometimes called FIFOs
- *Messages* — POSIX provides system calls `msgsnd()` and `msgrcv()`
  - message is block of text with a type
  - each process has a message queue, like a mailbox
  - processes are suspended when attempt to read from empty queue, or write to full queue.

IPC — Shared Memory

- *Shared Memory* — a Common block of memory shared by many processes
- Fastest way of communicating
- Requires synchronisation (See slide 48)

IPC — Signals

- Some *signals* can be generated from the keyboard, i.e., Control-C — interrupt (`SIGINT`); Control-\ — quit (`SIGQUIT`), Control-Z — stop (`SIGSTOP`)
- A process sends a signal to another process using the `kill()` system call
- signals are implemented as single bits in a field in the PCB, so cannot be queued
- A process may respond to a signal with:
  - a *default action* (usually process terminates)
  - a *signal handler* function (see `trap` in shell programming notes), or
  - ignore the signal (unless it is `SIGKILL` or `SIGSTOP`)
- A process *cannot ignore*, or handle a `SIGSTOP` or a `SIGKILL` signal.
  - A `KILL` signal will *always terminate* a process (unless it is in interruptible sleep)
  - A `SIGSTOP` signal will *always* send a process into the stopped state.

Signals and the Shell

- We can use the `kill` built in command to make the `kill()` system call to *send* a signal
- A shell script uses the `trap` built in command to *handle* a signal
- *Ignoring* the signals `SIGINT`, `SIGQUIT` and `SIGTERM`:

```
trap "" INT QUIT TERM
```

- *Handling* the same signals by printing a message then exiting:

```
trap "echo 'Got a signal; exiting.';exit 1" INT QUIT TERM
```

- Handling the same signals with a function call:

```
signal_handler() {
    echo "Received a signal; terminating."
    rm -f $temp_file
    exit 1
}
trap signal_handler INT QUIT TERM
```

- *Sending* a `SIGKILL` signal to process with PID 3233:

```
$ kill -KILL 3233
```

Threads

Lightweight processes that can talk to each other easily

Threads and Processes

- Threads in a process all share the same address space
- Communication easier
- Overhead less
- Problems of *locking* and *deadlock* a major issue

- Processes have separate address spaces
- Communication more indirect: IPC (Inter Process Communication)
- Overhead higher
- Less problem with shared resources (since fewer resources to share!)

Threads have own. . .

- stack pointer
- register values
- scheduling properties, such as policy or priority
- set of signals they can each block or receive
- own stack data (local variables are local to thread)

Threads share a lot

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- A number of threads can read and write to the same memory locations, and so you need to explicitly *synchronise* access

Threads in Linux, Unix

- POSIX is a standard for Unix
- Linux implements POSIX threads
- On Red Hat 8.x, documentation is at

  ```
  $ info '(libc) POSIX Threads'
  ```

  - or in Emacs, `C-H m libc` then middle-click on `POSIX threads`
- Provides:
  - *semaphores*,
  - *mutex*es and
  - *condition variables*

  for locking (synchronisation)

hello.c: a simple threaded program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void * print_hello( void *threadid )
{
        printf( "\n%d:  Hello World!\n", threadid );
        pthread_exit( NULL );
}
int main()
{
        pthread_t threads[ NUM_THREADS ];
        int rc, t;
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Creating thread %d\n", t );
                rc = pthread_create( &threads[ t ], NULL, print_hello, ( void
* ) t );
                if ( rc ) { printf( "ERROR; pthread_create() returned
%d\n", rc );
                        exit( −1 );
                }
        }
        pthread_exit( NULL );
}
```

How to Compile a POSIX Threads Program

- Need to use the `libpthread` library
  - Specify this with the option `-lpthread`
- Need to tell the other libraries that they should be *reentrant* (or "*thread safe*")
  - This means that the library uses no static variables that may be overwritten by another thread
  - Specify this with the option `-D_REENTRANT`
- So, to compile the program ⟨*program*⟩.c, do:

```
$ gcc -D_REENTRANT -lpthread -o ⟨program⟩ ⟨program⟩.c
```

pthread_create()

```
#include <pthread.h>

void *
pthread_create( pthread_t  *thread,
                pthread_attr_t *attr,
                void *(*start_routine)(void *),
                void *arg );
```

- returns: 0 if successfully creates thread
- returns error code otherwise

pthread_create()

- Quite different from `fork()`
- Thread must always execute a *user-defined function*
- parameters:
  1. pointer to thread identifier
  2. attributes for thread, including stack size
  3. user function to execute
  4. parameter passed to the user function

Problem with threads:

- Avoid 2 or more threads writing or reading and writing same data at the same time
- Avoid *data corruption*
- Need to control access to data, devices, files
- Need *locking*
- Provide three methods of locking:
  - mutex (**mut**ual **ex**clusion)
  - semaphores
  - condition variables

Race Condition

Race Conditions

- *race condition* — where outcome of computation depends on sheduling
- an error in coding
- Example: two threads both access same list with code like this:

```
if ( list.numitems > 0 ) {
        // Oh, dear, better not change to
        // other thread here!
        remove_item( list ); // not here!
        // ...and not here either:
        --list.numitems;
}
```

Critical Sections

- *critical resource* — a device, file or piece of data that cannot be shared
- *critical section* — part of program only one thread or process should access contains a critical resource
  - i.e., you lock *data*, *not* code
- All the code in the previous slide is a critical section
- Consider the code:

```
very_important_count++;
```

- executed by two threads on a multiprocessor machine (SMP = **s**ymmetric **m**ulti**p**rocessor)

Race Condition — one possibility

thread 1	**thread 2**
read `very_important_count` (5)	
add 1 (6)	
write `very_important_count` (6)	
	read `very_important_count` (6)
	add 1 (7)
	write `very_important_count` (7)

Example — another possibility

thread 1	**thread 2**
read `very_important_count` (5)	
	read `very_important_count` (5)
add 1 (6)	
	add 1 (6)
write `very_important_count` (6)	
	write `very_important_count` (6)

Solution: Synchronisation

- Solution is to recognise *critical sections*
- use *synchronisation*, i.e., locking, to make sure only one thread or process can enter critical region at one time.
- Methods of synchronisation include:
  - file locking
  - semaphores
  - monitors
  - spinlocks
  - mutexes

File Locking

- For example, an `flock()` system call can be used to provide *exclusive access* to an open file
- The call is *atomic*
  - It either:
    - completely succeeds in locking access to the file, or
    - it fails to lock access to the file, because another thread or process holds the lock
    - No "half-locked" state
  - *No race condition*
- Alternatives can result in race conditions; for example:
  - thread/process 1 checks lockfile
  - thread/process 2 checks lockfile a very short time later
  - both processes think they have exclusive write access to the file
  - file is corrupted by two threads/processes writing to it at the same time

Methods of Synchronisation

What is it?

mutex, semaphore, condition variables, monitor, spinlock

Synchronisation

- *Synchronisation* is a facility that enforces
  - mutual exclusion and
  - event ordering
- Required when multiple active processes or threads can access shared address spaces or shared I/O resources
- even more critical for SMP (*S*ymmetric *M*ulti*p*rocessor) systems
  - kernel can run on any processor
  - all processors are of equal importance (there is no one CPU that is the “boss”)
  - SMP systems include PCs with more than one CPU, as you might find in the Golden Shopping Centre

Semaphores

- A variable with three opererations:
  - *initialise* to non-negative value
  - *down* (or *wait*) operation:
    - decrement variable
    - if variable becomes negative, then process or thread executing the *down* operation is blocked
    - has nothing to do with the `wait` system call for a parent process to get status of its child
  - *up* (or *signal*) operation:
    - increment the semaphore variable;
    - if value is not positive, then a process or thread blocked by a *down* operation is unblocked.
- A semaphore also has a *queue* to hold processes or threads waiting on the semaphore.

Semaphores — 2

- The *up* and *down* semaphore operations are *atomic*
    - the *up* and *down* operations cannot be interrupted
    - each routine is a single, indivisible step
- Using semaphores—pseudocode

```
/* only one process can enter critical section at one time: */
semaphore s = 1;

down( s );
/* critical section */
up( s );
```

- *Initialise* semaphore to number of processes allowed into critical section at one time

Mutex—posix and Win32 Threads

- ***mut***ual ***ex***clusion
- Easier to use than semaphores (see slide 50)
- When only one thread or process needs to write to a resource
    - all other writers refused access
- A special form of the more general *semaphore*
    - Can have only two values;
    - sometimes called *binary semaphores*.

mutex — posix Threads Example (1)

- It is *good practice* to put the mutex together with the data it proects
- I have removed the error checking from this example to save space—in real code, always check library calls for error conditions

```
#include <pthread.h>
#include <stdio.h>

struct {
        pthread_mutex_t mutex; /* protects access to value */
        int value;                    /* Access protected by mutex */
} data = { PTHREAD_MUTEX_INITIALIZER, 0 };
```

mutex — posix Threads Example (2)

```
#define NUM_THREADS 5

void *thread( void *t_id ) {
        int i;
        for ( i = 0; i < 200; ++i ) {
                pthread_mutex_lock( &data.mutex );
                ++data.value;
                printf( "thread %d:  data value = %d\n",
                        t_id, data.value );
                pthread_mutex_unlock( &data.mutex );
        }
        pthread_exit( NULL );
}
```

mutex — posix Threads Example (3)

```
int main() {
        pthread_t threads[ NUM_THREADS ];
        int rc, t;
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Creating thread %d\n", t );
                pthread_create( &threads[ t ], NULL, thread,
                                ( void * ) t );
        }
        pthread_exit( NULL );
}
```

posix Condition Variables

- Lets threads sleep till a condition about shared data is true
- Basic operations:
  - signal the condition (when condition is true)
  - wait for the condition
    - suspend the thread till another thread signals the condition
- Always associated with a mutex
- Very useful
- Missing from Windows: See `http://www.cs.wustl.edu/~schmidt/win32-cv-1.html`

Monitors

- A *higher level structure for synchronisation*
- Implemented in Java, and some libraries
- main characteristics:
  - data in monitor is accessible only to procedures in monitor
  - a process or thread enters monitor by executing one of its procedures
  - Only one process or thread may be executing in the monitor at one time.
- Can implement with mutexes and condition variables.

Spinlocks

- Used in operating system kernels in SMP systems
- Linux uses kernel spinlocks only for SMP systems
- a very simple single-holder lock
- if can't get the spinlock, you keep trying (spinning) until you can.
- Spinlocks are:
  - very small and fast, and
  - can be used anywhere

Summary and References

Summary — Process States, Scheduling

- Scheduler changes processes between ready to run and running states
    - context switch: when scheduler changes process or thread
- Most processes are *blocked*, i.e., sleeping: waiting for I/O
    - understand the process states
    - why a process moves from one state to another
- Communication between processes is not trivial; IPC methods include
    - pipes
    - messages
    - shared memory
    - signals
    - semaphores

Summary — Processes and Threads

- With Linux and Unix, main process system calls are `fork()`, `exec()` and `wait()` — understand the function of each of these
- Windows provides `CreateProcess()` and various `WaitFor...()` Win32 API calls
    - The `WaitFor...()` calls have a purpose similar to that of the `wait()` system call in Linux and Unix
- *Threads* are lightweight processes
    - part of one process
    - share address space
    - can share data easily
    - sharing data requires synchronisation, i.e., locking

Summary — Synchronisation

- When two threads of execution can both write to same data or I/O,
    - Need enforce discipline
    - Use *synchronisation*
- We looked at the following methods of synchronisation:
    - semaphore
    - mutex
    - condition variable
    - monitor
    - spinlock
- There are other methods we have not examined here.

References

There are many good sources of information in the library and on the Web about processes and threads. Here are some I recommend:

- A good online tutorial about POSIX threads: `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`

- `http://www.humanfactor.com/pthreads/` provides links to a lot of information about POSIX threads

- The best book about POSIX threads is *Programming with POSIX Threads*, David Butenhof, Addison-Wesley, May 1997. Even though it was written so long ago, David wrote much of the POSIX threads standard, so it really is the definitive work. It made me laugh, too!

- *Operating Systems: A Modern Perspective: Lab Update*, 2nd Edition, Gary Nutt, Addison-Wesley, 2002. A nice text book that emphasises the practical (like I do!)

- Microsoft MSDN provides details of Win32 API calls and provides examples of code.

- William Stallings, *Operating Systems*, Fourth Edition, Prentice Hall, 2001, chapters 3, 4 and 5

- Deitel, Deitel and Choffnes, *Operating Systems*, Third Edition, Prentice Hall, 2004, ISBN 0-13-1182827-4, chapters 3, 4 and 5

- Paul Rusty Russell, *Unreliable Guide To Locking* `http://kernelnewbies.org/documents/kdoc/kernel-locking/lklockingguide.html`

- W. Richard Stevens, *Advanced Progamming in the UNIX Environment*, Addison-Wesley, 1992

- Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley, 2004, ISBN 0-13-142901-9.