Programs and Processes

- *Goals:*
  - * Understand the process model
  - * Learn to operate on processes
  - * Be Aware of the environment
- *Topics:*
  - * Motivation
  - * 2.0 What is a Process?
  - * 2.1,2.3-4, 2.9 Model: Layout, Attributes and States
  - * 2.5-8, 2.10 Operations: create, wait, kill, background...
  - * 2.11, 2.2 Critical Sections, Static variables in C
  - * 3 Views: System calls, Commands & Shell scripting
- *Readings: Chapter 2 (Robbins, pp. 29-76)*
- *Recommended Exercises: 2.1 - 8*

Why use processes?

- *Autonomous vehicle, e.g. Mars rover*
  * Take in surrounding terrain for path mapping
  * Read depth sensors to check distance from obstacles
  * Control power sent to different motors
  * Data gathering: air, temperature, light, soil
  * Listen to control tower on earth (e.g. unjam antenna)
- *Questions:*
  * How to structure your application?
  * multiple concurrent tasks!
  * give timely-enough response to many
- *Tools: Processes, Signal handlers, threads*
- *Ch. 2: How can processes help?*

Single Process Approach

- *Consider audio streaming (e.g. real audio)*

```
/* Cyclic Executive approach */
while (1) {
        /* 1. Synchronize to highest frequency */
        /* 2. Read Keyboard and mouse */
        /* 3. Recompute player position */
        /* 4. Uncompress audio */
        /* 5. Update display and emit sounds */
}
```

- *Pros: Okay for simple applications (few harmonic tasks)*

- *Cons: (i) Timing gets harder as number of tasks increase!*

  * (ii) Expensive steps make other steps wait too long
  - poll smaller tasks periodiclly from within longer ones?

  * (iii) May miss hard real-time constraints!

  * (iv) Performance tuning is tricky!
  - reorder steps, decompose long steps,

  * (v) Adding/removing a tasks => review entire loop

Multiprocess Approach

- *Strategy:*
  - * Each little while loop runs at its own pace
  - * Each process is scheduled by Operating System

    /* 1. Parent process creates N child processes */
    /* 2. child process(i) has a while loop on task(i) */
    /* 3. Parent runs GUI, passes user request to children */
    /* 4. Parent coordinates children, e.g. for exit */

- *Pros:*
  - * Simpler to design, code, debug, tune, port
  - * (Even better solution is threads)
  - * Scalability to multiprocessors or networks
  - * Modularity: adding/removing processes is simpler
  - * Protection: mission-critical application
  - \- Reduced impact of bugs in code for a task

- *Cons: costs more, are slower, needs scheduling*
  - * Coordination needs to be programmed explicitly.

2.0 What is a process?

- *Process*
  * "is instance of a ’program’ whose execution has started
  - but has not yet terminated" (pp. 29)
  * "has its own address space and execution state"
- *Recall what a Program is.*
  * C source program, e.g. Example 2.1 (pp. 40)
  * Executable program, e.g. a.out
- *Q? When does an executable program become a process?*
  * O.S. reads the program in memory
  * O.S. gives it a unique identifier, i.e. process ID
  * O.S. track its state, memory address / layout, ...
  * O.S. has allocates required resources
- *Q? How many processes can be created for a program?*
- *Q? How many executable program can a process run?*
  * At a given time
  * Over its lifetime

2.1 Process Model: Layout

- *A sample layout - Figure 2.1 (pp. 32)*
- *Sections*
  - * Program text - executable code
  - * Static data - e.g. global variables
  - * Dynamic Data
    - \- Heap - for malloc() on pointers
    - \- Stack - activation records during a function call
  - * Environment - e.g. command line arguments
- *Q? What is the life-time for a data-item in stack?*

\- in heap?

- *Q? Determine the size of initialized static data for*

\- the two C programs in Exercise 2.1 (p. 33)

- *Q? What is the layout of a.out file? (*)*

2.3-4, 2.9 Process Attributes

- *Process Attributes*
  * Ids: process ID, parent process ID, ... (Sec. 2.3)
  * User/System Environment (Sec. 2.9)
  * Context switch attributes for CPU scheduler (Sec. 2.4)
- *Process Id : a unique integer identifying processes*
  * Helps O.S. track process requests, state, etc.
- *Other attributes*
  * Parent process- requests creation of a process
  * Owner or "user" has special privilege over a process
  - ’effective user ID’ may vary over a process execution
  * User/System Environment (Section 2.9)
  - Current directory, terminal type, path, ...
- *Q? How to get process attributes?*
  * commands: ’ps’ (Ex. 2.2, pp.42), ’env’ (Ex. 2.17, pp. 64)
  * system calls: getpid(), getppid() in Ex. 2.1 (pp. 40),

2.3 Process Attributes

- *Context switch and Process context attributes*

  * Context Switch = transfer CPU between processes

  * Process context - information needed to restart a process
  - Process Id, User Id, previlege,
  - Layout: stack, heap, static data,
  - CPU registers (e.g. program counter)
  - handles for open files (e.g. STDIN), sockets, etc.
  - process state, status of I/O, scheduling/accounting info.

- *Q? Compare and contrast process context and environment.*

- *2.9 Process Environment*

  * Unix command "env" (Example 2.17, pp. 64)

  * POSIX environment: Table 2.5 (pp. 62)
  extern char **environ - Example 2.15 (pp. 63)
  char *getenv(const char *name);

  * Example 2.16 (pp. 63-64)

2.4 Dynamic Model: State Transition Model

- *State of a process = status at a particular time*
  - * Common Process States (Table 2.2, pp. 41)
  - * new, running, blocked, ready, done
- *State transition diagram (Fig. 2.3, pp. 41) - Events*
  - * Create, Terminate
  - * CPU scheduler - selected to run and restarted,
  - \- quantum expired and
  - * O.S. service (e.g. I/O) request, service complete
- *Q? Trace life-cycle of Example 2.1 in the STD (Fig. 2.3).*
- *Q? Which of the above events lead to context switches?*

2.5-8, 2.10 Process Operations

- *Conceptual operations (User level)*
  - * Create (Sec. 2.5)
  - * O.S. service, e.g. wait for an event (Sec. 2.6)
  - * Change program code (Sec. 2.7)
  - * Run background (Sec. 2.8)
  - * Terminate (Sec. 2.10)
- *Operations can be performed via*
  - * Command line, Shell scripts
  - * System calls in C like language
- *Operation from Command line*
  - * Shell creates a child process to run each command
  - * Shell waits for the child to complete,
  - - unless child is to run in background
  - * ’kill’ or ’ctrl-c’ to terminate a process
  - * ’ctrl-z’ to stop a process
  - * Other: &, bg, fg, ps

Process Operations : System Calls

- *System calls*
    - * Create - fork(), Example 2.4 (pp 44)
    - - exec() to change code section- Program 2.6 (pp 53)
    - * Terminate - exit()
    - * Wait for a child - wait(), waitpid()
    - - Exercise 2.3 pp. 49
    - * Background - setsid(), Example 2.14 (pp 60-61)
- *Example 2.14: Simple Biff (pp 60-61)*
    - * 2.14 Exercise: Simple Biff (pp 72-73)

Process Creation - fork()

- *Syntax*

  pid_t fork(void) [See pp. 43]

  * Returns pid of child to parent process
  * Returns 0 to newly created child process

- *Coding style* - *See Example 2.4 (pp. 44)*

- *Semantics*

  * Layout of Child = layout of parent.
  * Different: value returned by fork(), pid, ppid,
    - CPU use meter, alarms, locks, pending signals
  * Identical but disjoint address space:
    - code section, program counter, data section,
    - environment, previledges, scheduling priority, ...
    - changes after fork() are local
  * Shared: open filepointers, system resources
    - changes after fork() are seen by other process!

Process Creation

- *Hierarchical Parent, child relationship*
  * Parent process creates children processes
  * which, in turn create other processes
  * forming a tree of processes
  * Example 2.7 (pp. 46-47)

- *Answer the following question on fork():*
  * Q? How many parent can a process have?
  * Q? How many children can a process have?
  * Q? Identify the default Concurrent Execution Option:

  - 1. Parent and children execute concurrently.
  - 2. Parent waits until children terminate.

  * Identify default resource sharing options:

  - 1. Parent and children share all resources
  - 2. Children share subset of parent’s resources
  - 3. Parent and child share no resources

2.10 Process Termination

- *Self determined termination*

void   _exit(int status)
void   exit(int status)

  * Ask OS for termination
  * Process' resources deallocated by OS
  * Send data to parent
    - parent does wait() to receive data
    - Example 2.8 (pp. 48)

- *Parent requested termination*

  * Parent executes kill(child_pid, signal_int)
  * Details in Chapter 5
  * Purpose: child exceeds allocated resources
    - task allocated to child not needed
    - parent is exiting

2.6 Coordinating Processes

- *Purpose: Data sharing, speed-up computation,*

- modularity, Convenience, etc.

- *Mechanisms:*

  * parent: wait(childpid) - child: exit(...)

  * Pipes (Ch. 3), Signals (Ch. 5), Message (Ch. 12)

  * Other - Critical section (8, 2.11), shared memory, ...

- *Syntax:*

  pid_t wait(int *stat_loc);

- *Semantics*

  * Return value = pid of terminated child
  - or -1 with error code in "errno"

  * Pause caller until a child terminates/stops
  - or caller receives a signal ("errno" = EINTR)

  * Return immediately if no children (errno = ECHILD)
  - if a un-waited-for child has already terminated

  * Program 2.5 (pp. 49)

- *Interesting Exercise 2.3 (pp. 49-50).*

2.6 Coordinating Processes (contd.)

- **stat_loc - to get return status of child process*

  * Recall child process : exit(status)

  * Macros to test status value
  - e.g. WIFEXITED, WIFSIGNALED, ...

  * Example 2.8 (pp. 48-49)

- *Waiting for a specific child*

  pid_t waitpid(pid_t pid, int *stat_loc, int options);

  * pid > 0 => wait for a specific child
  - pid = -1 => wait for any child

  * option = WNOHANG => non-blocking wait.

  * Example 2.11 (pp. 51)

  * See details in man page.

- *Interesting Exercises 2.4, 2.5 (pp. 52-53).*

Special Topics in Process Creation

- *2.7 The exec System call*

  * Used after fork() to change code section

  * Overwrites data (globals, stack, heap)

  * What is preserved after exec (Table 2.4 (pp. 58))
  - May preserve argv[] unless execle / execve
  - Preserves open files
  - Effect on signal, locks in Chapter 5.

- *2.8 Background Processes*, *e.g. ls -l &*

  * Parent does not wait for the process to finish

  * ctrl-C does not terminate it

  * Q? How to create a background process?
  - Program 2.9 (pp. 59)
  - setsid() - create session w/o controlling terminal

- *Daemon: background process run indefinitely*
- Q? Why?