Mon Sep 17, 2007
Lecture 3: Process Management

September 19, 2007

1 Review

"OS mediates between hardware and user software"

QUIZ: Q: Name three layers of a computer system where the OS is one of these layers.

A: Hardware, OS, Application Programs

Q: Which of the following concepts do not belong in the group?

Interrupt, Bootstrap, System Call, Trap, Exception

A: Bootstrap. The others are just forms of interrupts.

Interrupt is a generic name for any instruction that stops the CPU in its current execution. TRAP are self-generated software-generated interrupts System Calls. Exceptions are interrupts generated by some error condition or special status. System calls is an OS-specified function called via a TRAP. I/O Interrupt is generated by I/O hardware.

Q: An OS provide services. How does a user access these services?

A: Make system calls.

to READ from disk:

A: Steps of the READ system call:

(1) User Program loads the arguments and calls a system library READ routine

(2) This routine loads the code for reading and TRAPS

(3) CPU switch to kernel mode, and executes at fixed address in the kernel.

(4) CPU returns to the library READ routine

(5) Library READ returns to user program.

2 Basic Issues in Multiprogramming

We could begin with the simple view of an OS as *a software to manage a set of processes.*

A **process** (or **job** or **task**) is a program in execution. As such, it has source code, current state, memory and files, resources such as printers and I/O.

A closely related, more modern concept is **thread**. There are many variants of the thread concept, and its relation to processes. Here are some views:

(1) Threads as a executional unit of a process.

(2) Threads as a light-weight process.

(3) Threads as kernel objects, or as process objects.

In any case, an extended notion of OS is that "an OS manages a collection of processes and their threads".

The OS, as a manager of processes/threads, has to provide these functions:

- Process Management:

  (1) Create/delete, schedule, synchronize and provide communication among processes.

  (2) Process scheduling

- Memory management:

  (1) Main Memory Management

  (2) File management

  (3) Memory hierarchy issues

- Resources management:

  e.g., Deadlock management.

  e.g., Networking

  e.g., I/O management

- Security and Protection:

Just as there is some hierarchy involving threads and processes, we may also have hierarchy among the processes. E.g., in Unix, each process (except the root) has a parent process. The OS has to manage this hierarchy.

3 Memory Management for Multiprogramming

KEY FUNCTION: provides multiprogramming environment

Usefulness of multiprogramming: I/O bound vs CPU bound processes User processes vs Kernel processes When useful?

What is needed for multiprogramming? Process management (process table) Memory management Interrupts (and ability to turn on/off interrupts) ...

Requirements: to isolate processes from each other, to isolate processes from kernel, to relocate programs.

Relocation problem: when a program is compiled, it must be assumed that each instruction is placed in some location, and the locations conventionally start at 0. When this program is LOADED, it will begin at some address determined at runtime. Also, a certain amount of space is allocated for data.

THE CDC MODEL (CDC is an early computer, one of the first models was delivered to Courant Institute in NYU in the 70s):

- Each process occupies a contiguous chunk of memory.
- THIS CALLS for a BASE REGISTER and a LIMIT REGISTER: Limit register tells the size of the program+data. Chunk = [base, base + limit] These registers are protected from users
- How does this modify our execution cycle? Each instruction or data fetch is checked to be within the limit, and then "offset" by base. But actually, it is done by the MMU.
- THIS illustrates –virtual address vs physical address. –context switching: the values of the register pair is a context!
- Drawback: we need to predict how much data is needed. So everyone overestimates.
- MODIFIED CDC MODEL: Have two pairs of Base+Limit registers. Pair 1 for program, Pair 2 for data. ADVANTAGE: processes can share program!

4 Processes

Our first programming effort will involve multi-processes. So...

We will examine processes in Unix and Windows!

How are processes created?

- At system initialization (foreground process, e.g., login shells), (background, e.g., email, http daemons)
- Users can create them (e.g., multiple window processes)

  E.g., > cat chap1.txt chap2.txt — grep process

  Current process creates two new process, one to run "cat" (which contenates 2 files), the other to run "grep" (to look for word "process").
- UNIX: "fork" (creates a clone of caller process! but the the child can next do some manipulation and ultimately execute a "execve" to do its job)

  BUT both have different address spaces.

  THEY CAN share resources like open files.
- Windows: "CreateProcess" with 10 parameters! and lots of related functions.

Process Hierarchies

- Windows: no hierarchy! The parent process has only one special advantage: it has the "token" or "handle" to control the child, but can give the handle away!
- Unix: from fork, get parent-child relation. ALL PROCESSES are descendents of the "init" process!

  The hierarchy cannot change except through process creation or process termination: when a parent process exits, all its children will have the "init" process as parent!
- Unix: Process group – a process and all its descendents. If a signal is sent to root of group, it is delivered to all.

How are Processes terminated?

- Normal exit (voluntary),

  Error exit (voluntary),

  Fatal error exit (involuntary),

  Killed by another process exit (involuntary),
- Unix: normal termination is achieved by a process executing the `exit()` system call. The exiting process may return a status value to parent.
- Unix: a process can terminate another process by calling `abort()`. E.g., a parent process aborts a child process if it is taking too long, or its services is no longer needed.

EXAMPLE: Unix Fork

- The main interface to unix is the **shell** (there are many variants, **bash** being our default in cygwin).
- Each shell controls a window (assuming a multi-window GUI)
- The shell sits in a loop to process use commands.
- Suppose we type "ls" (list current directory).
- The shell would execute a "fork()" system call.
- This creates a new process, a CHILD of the current process.
- The PARENT process then executes a "wait()" for the CHILD to terminate.

- After the CHILD does its job (listing current directory), it terminates and sends the parent a signal.
- The parent receives the signal and continues.
- HERE IS THE CODE to SIMULATE the process:

```
#include < stdio.h >
#include < unistd.h >
int main (int argc, char *argv[]){
    int pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, ”fork failed”); exit(-1);}
    else if (pid == 0) {
        execlp(”/bin/ls”, ”ls”, NULL);}
    else {
        wait(NULL); printf(”child completed”); exit(0);}
}
```

5 Process States and Transition

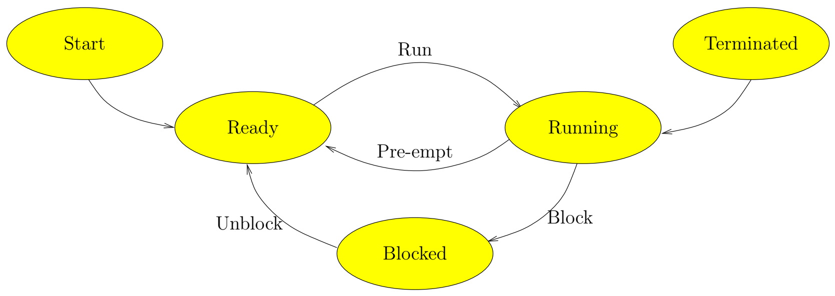

Figure 1: Process States and their transitions

- **Cyclic States:** Ready, Running, Blocked.

  Transitory States: New, Terminated.

  Refinements: Ready-suspended, blocked-suspended. (suspension is related to process swapping)

  NOTE: there is at most ONE process in the Running state (for each CPU).
- Possible state transitions (draw a transition diagram)

  Ready ↔ Running → Blocked → Ready.

  Also: New → Ready, Running → Terminated.
- Names of transitions:
  - (1) create (New → Ready)
  - (1) terminate (Running → Terminated)
  - (2) preempt (Running → Ready) caused by scheduler
  - (2) run (Ready → Running) caused by scheduler
  - (3) block (Running → Blocked)
  - (3) unblock (Blocked → Ready) caused by event

6 Process Scheduler

- OBJECTIVES OF SCHEDULING:
  - Efficient use of resources (e.g., CPU usage). That is the reason for ”blocking” a process.
  - Fairness among processes (time-to-completion should be proportional to job size). That is the reason for ”preempting” a process at the end of a quantum.

- (running → ready) and (ready → running)

  are done by process scheduler!

  These 2 transitions requires **context switching**, which is hardware supported.

- Scheduler [p.142-3]. Round Robin is simplest preemptive scheduler:

  (1) Ready processes are in FIFO queue (ready queue)

  (2) Each process runs in a fixed quantum (e.g. 50ms)

  (3) Current process P runs till it BLOCKS, in which case it is moved to ”blocked queue”, or it finished its quantum, and put in end of ”ready queue”.

  (4) Next process at top of FIFO queue is next running.

- Scheduler Queues:

  Job Queue: all current jobs

  Ready Queue: jobs in main memory in Ready state

  Device Queue: jobs waiting for a particular device

  See DIAGRAM of transitions between queues. [Silberschatz, Fig 4.5, p.109]

- REFINEMENTS of Schedulers (Silbershatz, p.109):

  CPU Scheduler (or short-term scheduler): manages the ready queue

  JOB Scheduler (or long-term scheduler): manages the Job Queue on disk (swaps processes in/out of main memory) It decides on how to admit New processes into the ready queue.

7 Process Management

We now look a bit into the nitty-gritty for process management.

- Datastructure called **Process Table** (one entry/process). Each entry is called a **Process Control Block** (PCB).

- Each entry or PCB:
  - Process ID
  - Process State
  - Registers: PC, SP, PSW, etc
  - memory allocation
  - files and their status
  - accounting (CPU usage, memory usage, etc)
  - scheduling info
  - I/O status
  - info for context switching

- OTHER INFORMATION: priority, parent process, process group, signals, time when process started, children's CPU time, time of next alarm; pointer to text segment, data segment, stack segment; root directory, working directory, file descriptors, user ID, group ID.
- Process Image: memory must be organized to store each current process (in any state).

  TYPICAL LAYOUT IN MEMORY:

  text section (code),

  data section (global variables),

  stack section (params of calls, return addr, local vars)
- Each I/O Class (floppy, hard disk, timers, terminals,...) has a location called an **interrupt vector**. It has the address of the interrupt routine.

  If process 3 is running when a disk interrupt occurs: then HARDWARE causes process 3's PC, registers, etc to be pushed onto the current stack, and make CPU jumps to the interrupt vector.

  That is all the hardware does. From then on, the interrupt service procedure takes over.

  When done, returns to the scheduler.

8 THREADS

What is a THREAD?

- View a traditional process as a single thread.
- A process can have SEVERAL threads of execution.

  The threads all SHARE the same address space.
- The reason for threads are similar to the ones for processes BUT:

  easy to create and destroy

  the need to share data

  useful when threads are I/O bound
- E.g. Process running an editor: the process can run several threads at the same time:

  (1) thread for backup of file in background

  (2) thread for reformatting of file while we continue to edit the file

  (3) thread for reading from keyboard

  (4) thread for writing to screen

  (5) thread to do spell checking in background

  WHAT IS THE POINT? WHY DON'T WE JUST HAVE JUST HAVE 5 PROCESSES?
- Processes share resources, but threads are executional units, that does not have independent resources. So sharing among threads within a process is very cheap.
- Thread can be implemented in kernel or user space

9 Buses

It is useful to know about an essential piece of hardware in all modern computers: the bus.

Various hardware devices talk to each other using buses. A **bus** is a set of wires with a well-defined protocol for sending and receiving messages along the wires. Various devices (monitor, procesor, keyboard, disk, etc) can be attached to the bus. Usually, devices have a dedicated unit called **controller** to interface with the bus.

PC's have 8 BUSES.

- original IBM:

  ISA (ISA bridge $\leftrightarrow$ various devices) Rate of 16 MB/sec.

- successor to ISA:

  PCI (PCI bridge $\leftrightarrow$ various slots, inc.ISA Bridge) Rate of 528 MB/sec.

- Newer buses:

  cache (CPU$\leftrightarrow$Level 2 cache)

  local (CPU$\leftrightarrow$PCI bridge)

  memory (PCI bridge$\leftrightarrow$ Main Memory)

  SCSI (high performance bus for fast disks, scanners, etc) Rate of 160 MB/sec.

  USB (for slow I/O devices like keyboard, mouse) Rate of 1.5 MB/sec. Convenient to add devices, without reboot.

  Since 2001, USB 2.0 (Hi-Speed USB) introduced with rate of 60 MB/sec.

  IDE (for peripherals such as CD ROMs, disks)

- New: FireWire (a.k.a. i.Link or IEEE 1394) (50 MB/sec) like USB but much faster (useful for audio/video devices like camcorders, etc) Partly viewed as replacement for USB.