CS162, Spring 2004 **Discussion #2** **Amir Kamil**
UC Berkeley **1/29/04**

Topics: Multiprogramming, Processes and Threads

1 Announcements

- Office hours will be M 12-1 and Tu 1-2 in 551 Soda; in addition, I have HKN office hours Tu 10-11 and 2-3 in 345 Soda
- Section notes will be posted when they are done
- SSH
  - You need to generate a public key using `ssh-keygen`, if you haven't already
  - Pick a passphrase you can remember!
  - The key will be used for identification when using CVS
- Email `cs162@cory` if you are still looking for partners
- Use the newsgroup for MacOS help; none of the TAs use Macs
- Project info
  - We will be using Java (1.3); you mainly need to understand basics of language, classes, etc.
  - You will not be using Java threads (directly), so you don't really need to understand them either
  - First project design due on Feb 18th
    - Submit in text, pdf, or html format
    - An example design document will be on the webpage soon
    - Keep them short: 2000-4000 words
    - Content should be algorithmic pseudocode, not actual Java code
    - **GET STARTED NOW!** Don't wait until the last minute
  - Group management
    - Communication is very important! Lack of communication is the biggest reason groups break down
    - You should have regular meetings during project time, every other day or so. Schedule them in advance so no one has an excuse for missing them
    - We do not suggest you break the project into 5 pieces: the more pieces, the higher the communication costs. Break into no more than two or three pieces, assign multiple people to each section
    - Use CVS! This will make it much easier to manage your code, and will greatly reduce the chances of you clobbering each other's changes. A CVS walkthrough is on the class website

2 Multiprogramming

Recall that one of the main functions of an operating system is to coordinate resources and provide protection between programs. Modern operating systems support *mulitprogramming*, or allowing multiple programs to run simultaneously, providing each with the illusion that it is the only one running. In reality, there is only a limited amount of hardware resources: one CPU, a small amount of shared memory, one or two disks, and so on. So how does multiprogramming work?

2.1 Virtual Memory

In order to to provide protection between programs and abstraction, a multiproramming operating system usually gives each running program its own *virtual address space*. Each program has a range of *virtual addresses* that it can access, which the operating system and hardware translate to physical addresses. This protects programs from one another; a program can only access its own virtual address space, which is distinct from any other program's, so one cannot overwrite another's memory. Virtual memory also allows programs to be compiled using virtual addresses, and then be placed anywhere in memory when they are run. We will also see later that virtual memory can provide the illusion of having more memory then there actually is physically, through *paging*. The disadvantage to virtual memory is that translation takes time, slowing down every memory access.

The virtual address space abstraction requires both hardware and software support. The hardware must provide a *memory management unit (MMU)* and some extra CPU registers to perform address translation, and the operating system must manage the required translation tables (more detail on this later in the course). The hardware also usually provides a cache (TLB) to speedup translations. The operating system still requires direct access to physical memory, in order to allocate and deallocate resources or perform I/O, for example, so the hardware must support *dual-mode operation*, a *user mode* with translation and a *kernel mode* without it. The operating system also must provide various routines to allow user programs to perform protected operations such as I/O, in to form of *system calls*, which we discuss below.

2.2 Switching and Scheduling

Only one program can run on the CPU at a time, so how can multiple programs run simultaneously? The operating system and hardware provide an illusion of simultaneity by rapidly switching between running programs. When does this switch occur? In older operating systems such as Windows 3.1, programs were expected to play nice and *yield* to the OS frequently, allowing the operating system to switch to another program. Not unexpectedly, this is not the case in modern operating systems. Currently, the hardware provides a timer that *interrupts* the system at regular intervals, transferring control to the OS. This is called *preemption*, and the act of switching to another program is called a *context switch*.

What must the operating system do in a context switch? It first decides which program to run next, using some *scheduling* algorithm (more on this later in the course). It has to save the state of the old program (CPU registers, address translation tables, program counter, etc.) and load in the state of the new program. The address translation cache, if there is one, must also be flushed to prevent a program from accessing memory it doesn't own. Finally, the operating system returns the CPU to user mode and resumes execution of the new program.

2.3 Exceptions

We mentioned interrupts and system calls above, but what exactly are they? They are part of a larger class of events known as *exceptions*, which transfer control to the operating system. There are two types of exceptions: *traps* and *interrupts*[1].

Traps are synchronous events in the CPU that only happen following the execution of an instruction. These include division by zero, page faults, and system calls. For system calls, the CPU provides a special instruction to trap to the OS, and the user programs sets a register prior to executing the instruction in order to inform the OS of which routine it wants run. The operating system, upon receiving control, checks this register and performs the desired operation before transferring control back to the user or context switching.

Interrupts are asynchronous events generated outside the CPU and can happen at any time. These include I/O notifications and the timer interrupt. The operating system uses an interrupt vector to determine which interrupt handler to run, calls the appropriate handler, and resumes user operation.

Exceptions must be provided by the system in order to support multiprogramming with protection. User programs cannot do protected operations, so there must be a way to offload the work to the operating system.

[1]Hence the alternate term *EIT*, for *exceptions* = *interrupts* + *traps*.

3 Processes and Threads

We have been quite sloppy in our terminology so far; what we have referred to as a program above is more properly called a process. A *program* refers to a piece of source code, while a *process* refers to an executing program and its associated state: arguments, address space and contents, registers, program counter, etc.

An actual flow of execution within a process is called a *thread*. Each process must have at least one, but may contain many. Threads within a process share the same address space, code and data sections, heap, and OS resources (e.g files), but have their own stack and CPU registers. The operating system keeps track of a thread using a *thread control block (TCB)*. This contains the current overall thread state (new, running [on CPU], blocked [waiting for I/O], ready [waiting for CPU], terminated), its CPU state (registers), its program counter, some scheduling information, memory information (stack pointer), I/O information (file descriptors, sockets), and additional information.

Since threads within a process share the same address space, there is no protection between them. This should be fine, since all the threads within a process should be cooperating. However, we now have to worry about different threads accessing some shared state simultaneously. This is called a *race condition*, and we use *synchronization* to solve the problem (as we will see in the next couple of weeks).

What are the advantages and disadvantages of threads over processes? Threads share virtual memory, so there is a limit on how many there can be (since each has its own stack), and you need to worry about synchronization. Sharing also prevents threads from being used on a distributed memory machine. However, there is less overhead to a context switch between threads, since you don't have to change the page table and flush the TLB. In addition, a process that does both computation and I/O can use a thread for each in order to overlap the two.