Processes and Threads

1. Processes

1.1 The Process Model

All the runnable software on the computer, sometimes including the OS, is organized into a number of sequential processes, or just **processes** for short. **A process is just an instance of an executing program**, including the current values of the program counter, registers and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming**. Each process has its own program counter; however, only one real program counter exists. When each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory.

Note: for now it will be assumed that there is only one CPU. So, when we say that a CPU can really run only process at a time, if there are two cores (or CPUs), each of them can run only one process at a time.

When a process requires synchronization with another process, special measures must be taken in order to ensure that it happens.

The difference between a process and a program is subtle, but absolutely crucial. Let's take, as an example, a computer scientist (CS) baking a cake for his daughter's birthday. In this analogy, the recipe is the program, that is, an algorithm expressed in some suitable notation, the CS is the processor (CPU), and the ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients and baking the cake. Now imagine the CS' son comes running in, screaming, saying that he has been stung by a bee. The CS records where he was in the recipe (**the state of the current process is saved**), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking), to a **higher-priority process** (administering medical care), each having a different program (recipe vs first aid book). When the second process is done, the CS will go back to his cake, continuing at the point where he left off. The key idea here is that a process is an activity of some kind. It has program, input, output and a state. A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that may

be stored on disk, idle. If a program is running twice, it counts as two processes. The fact that they are running at the same time does not matter; they are different processes (the program may be shared so only one copy is in memory, but they still are !=).

1.2 Process Creation

Four principal events cause processes to be created:

1. System initialization
2. Execution of a process-creation system call by a running process
3. A user request to create a new process
4. Initiation of a batch job

Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons**. Large systems commonly have dozes of them. In UNIX, the *ps* command can be used to list the running processes. In Windows, the task manager can be used. In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue a system call to create one or more new processes to help it do its job. For example, if a large amount of data is being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removed the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster. In interactive systems, users can start a program by typing a command or clicking on an icon. Taking either of these actions start a new process and runs the selected program in it. In command-based UNIX systems running X, the new process takes over the window in which it was started. In Windows, when a process is started, it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows open at once, each running some process. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed. The last situation in which processes are created applies only to the batch systems found on large mainframes. When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it. In UNIX, there is only one system call to create a new process: *fork*. This call creates an exact clone of the calling process. After the fork, the two processes, **the parent and the child**, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes execve ora similar system call to change its memory image and run a new program. For example, when a user types a command, say, *sort*, to the shell, the shell forks off a child process and the child executes sort. The reason for this two-step process is to allow the child to manipulate its file descriptors after the fork but before the execve in order to

accomplish redirection of standard input, standard output, and standard error. In both UNIX and Windows systems, after a process is created, the parent and the child have their **own distinct address spaces**. If either process changes a word in its address space, the change is not visible to the other process.

1.3 Process Termination

Sooner or later, the new process will terminate, usually due to one of the following conditions:

```
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
```

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler then executes a system call to tell the operating system that it is finished. This call is *exit* in UNIX and ExitProcess in Windows. (The close button of a window tells the process to **remove any temporary files it has open** and then terminate) The second reason for termination is that the process discovers a fatal error. For example, when a compiler is told to compile an inexistent program, the compiler simply announces this fact and exits. The third reason is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In some systems (eg. UNIX), a process may be interrupted instead of terminated if it asks the OS to handle such errors by himself. The fourth and last reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In UNIX this call is *kill*. The killer must have the necessary authorization to do in the killee.

1.4 Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a **process hierarchy**. In UNIX, a process and all of its children and further descendants together from a **process group**. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard. Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

1.5 Process States

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. Dependending on the relative speeds of the two processes (which depends on

both the relative complexity of the programs and how much CPU time each one has had), it may happen that the second one is ready, but the information it needs from the first may not be ready yet. It must then block until that information is available. When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible fora process that is conceptually ready and able to run to be stopped bcause the operating system has decided to allocate the CPU to another process for a while. These two cases are completely different. In the first case, the suspension in inherent in the problem (you cannot process the user's command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor). Therefore, we have three states a process may be in: - Running (actually using the CPU at the instant) - Ready (runnable; temporarily stopped to let another process run) - Blocked (unable to run until some external event happens) We therefore have 4 possible transitions possible:

```
1. Running -> Blocked; process blocks for input
2. Running -> Ready; scheduler picks another process
3. Ready -> Running; scheduler picks this process
4. Blocked -> Ready; input becomes available
```

In UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Using the process model, it becomes much easier to think about what is going on inside the system. Instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk as been read or the character typed, the process waiting for it is unblocked and is eligible to run again. This view gives rise to a model where the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in what is here called the scheduler, which is actually not much code. The rest of the operating system is nicely structured in process form. Few real systems are as nicely structured as this however.

1.6 Process States

To implement the process model, the operating systems maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready*or *blocked* state so that it can be restarted later as if it had never been stopped. Some of the fields of a typical

process-table entry:

```
|Process management        | Memory Management           | File
management  |
|:------------------------- |:---------------------------
|:---------------- |
| Registers                 |Pointer to text segment info |Root
directory    |
| Program counter           |Pointer to data segment info |
Working directory |
| Program Status Word       |Pointer to stack segment info|File
descriptors  |
| Stack pointer             |                             |User
ID           |
| Process state             |                             |Group
ID          |
| Priority                  |                             |
|
| Scheduling parameters     |                             |
|
| Process ID                |                             |
|
| Parent process            |                             |
|
| Process group             |                             |
|
| Signals                   |                             |
|
| Time when process started |                             |
|
| CPU time used             |                             |
|
| Children's CPU time       |                             |
|
| Time of next alarm        |                             |
|
```

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained in one (or each) CPU. Associated with each I/O class is a location (typically at a fixed location near the bottom of memory) called the **interrupt vector**. It contains the address of the interrupt service procedure. When an interrupt occurs, all the hardware does is it saves the current process' state and then jumps to the address specified in the interrupt vector. From here on, it is up to the software, in particular, the interrupt service procedure. In sum:

```
    1. Hardware stacks program counter, etc;
    2. Hardware loads new program counter from interrupt vector;
    3. Assembly-language procedure saves register;
    4. Assembly-language procedure sets up new stack;
```

```
    5. C interrupt service runs (typically reads and buffers
input);
    6. Scheduler decides which process is to run next;
    7. C procedures returns to the assembly code;
    8. Assembly-language procedure starts up new current
process.
```

1.7 Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. A good model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process model spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is given by the formula *CPU UTILIZATION = 1 - p^n*.

2. Threads

2.1 Thread Usage

Why would anyone want to have a kind of process within a process? It turns out there are several reasons for having these miniprocesses, called **threads**. The main reason is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler. This is the argument used to justify the usage of processes. Only now, with threads, we add a new element: the ability for parallel entities to share an address space and all of its data among themselves. This is ability is essential for certain apps, which is why having multiple processes with their separate addres spaces) will not work. A second argument would be that they are lighter than processes, easier to create and to destroy (in many systems, creating a thread may be 10 to 100 times faster than creating a process). A third reason for having threads is also a perfomance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application. Finally, threads are useful for multiple CPUs, where real parallelism is possible. An example to evidence process vs thread differences would be a word processor. Imagine a writer writing a book. The word processor could use 3 threads: one for the input, other for processing changes throughout the document and a last one for periodic backups. With this, the writer could make a change to page 1, that the second thread would process the changes throughout the whole book, allowing the writer to access page (e.g.) 600 faster, as he wouldn't have to wait for the changes to be processed after he is done making his changes. While all this is happening, the third thread would be making periodic saves from RAM to memory. All this is possible because the

threads share a same address space, thus all have access to the document being changed.

Web server example options:

```
|Model |Characteristics |
|:---------------------
|:-------------------------------------------------------------
-- |
|Threads                 |Parallelism, blocking system calls
|
|Single-threaded process | No parallelism, blocking system calls
|
|Finite-state machine    | Parallelism, nonblocking system
calls, interrupts |
```

Threads offer a solution. The process could be structured with an input thread, a processing thread and an output thread. In this way, input, output and processing can all be going on at the same time. Of course, this models works only if a system call blocks only the calling thread, not the entire process.

2.2 The Classical Thread Model

The process model is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in. The **thread** has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU. What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called **lightweight processes**. The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process. There is no protection between threads because (1) it is impossible, (2) it is not necessary. Unlike different processes, which may be hostile to each other, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads can share the same set of open files, child processes, alarms, signals, and so on.

```
|Per-process items          |Per-thread items |
| ------------------------- | --------------- |
|Address space              |Program counter  |
|Global variables           |Registers        |
|Open files                 |Stack            |
|Child processes            |State            |
|Pending alarms             |                 |
|Signals and signal handlers|                 |
|Accounting information     |                 |
```

When multithreading is present, processes usually start with a single thread present. This thread has the ability to create new threads by calling a library procedure such as *thread_create*. When a thread has finished its work, it can exit by calling a library procedure, say, *thread_exit*. It then vanishes and is no longer schedulable. In some thread systems, one thread can wait for a specific thread to exit by calling a procedure, for example, *thread_join*. Another command thread call is *thread_yield*, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Thus it is important to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run. Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on.

2.3 POSIX Threads

To make it possible to write portable threaded programs, IEEE has defined a standard for threads. The thread packages is called **Pthreads**. Most UNIX systems support it. Some of the major function calls of this package:

```
|Thread call          |Description
|
| ------------------- |
------------------------------------------------- |
|Pthread_create       |Create a new thread
|
|Pthread_exit         |Terminate the calling thread
|
|Pthread_join         |Wait for a specific thread to exit
|
|Pthread_yield        |Release the CPU for another thread to run
|
|Phtread_attr_init    |Create and initialize a thread's
attribute structure |
|Pthread_attr_destroy |Remove a thread's attribute structure
|
```

2.4 Implementing Threads in User Space

There are two main places to implement threads: user space and the kernel. The first method is to put the threads package entirely in user space. The kernel knows nothing about them; as far as it's concerned, it is managing ordinary, single-threaded processes. The first and most obvious advantage is that a user-level threads package can be implemented on an operating system that does not support threads. With this approach, threads are implemented by a library. When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process. This table is analogous to the kernel's process table, only that it keeps track of per-thread properties, and it is managed by the run-time system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way the kernel stores information about processes in the process table. However, there is one key difference with processes. When a thread is finished running for the moment, for example, when it calls *thread_yield*, the code of *thread_yield* can save the thread's information in the thread table itself. Furthermore, it can tell the thread scheduler to pick another thread to run. These are local procedures; much faster than a kernel call. This makes thread scheduling very fast. It allows customized scheduling algorithms for each process.

Problems: - Blocking system call implementation: this will block all other threads; goes against the purpose of having threads; Alternative would be to change to nonblocking sys calls, but the initial purpose of user-level threads was to run with existing OS; Another alternative would be a system call to tell in advance if a sys call will block. The code placed around the system call to do the checking is called a **jacket** or **wrapper**. - Page fault: as the kernel is unaware of the threads, it will block all threads while the disk I/O is complete (to fetch the missing instruction) - If a thread a thread starts running, no other thread in the process will ever run unless the first thread voluntarily gives up the CPU. - Programmers usually want threads specifically for situations whre the threads block often. Having the kernel manage them eliminates the need for constantly check if a call will block or not. In the cases the threads won't block, there is no need to actually implement them

2.5 Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage threads. No run-time system is needed in each. Also, there is no need for a thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table. All call that might block a thread are implemented as sys calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready) or a thread from a different process. With user-level threads, the run-time system keeps running

threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run). Thread recycling is implemented in order to save some overhead of creating a new thread every time it is needed.