CPS221 Lecture: Processes and Memory Management

last revised 9/5/12

Objectives

1. To introduce fundamental concepts concerning processes: state, pid, PCB
2. To introduce virtual memory
3. To introduce mechanisms for inter-process communication

Materials:

1. Ability to demonstrate Unix commands on Mac and Linux
2. pipedoc.txt and pipedict.txt examples for pipeline demo + file with command to cut and paste plus shell script version
3. Projectable version of Figure 6.4 from Hailperin

I. Introduction

A. Recall that earlier in the course we introduced the notion of a process as the fundamental abstraction in the realm of operating systems.

1. Recall that a process is a PROGRAM IN EXECUTION.

2. At any given time, the status of a process includes

a) Information about resources that may be allocated to the process (e.g. open files)

b) One or more threads of execution

c) A region of memory holding its code and data.

3. Recall also that, historically, processes had only one thread of execution, so that there was typically no distinction drawn between a process and its (unique) thread of execution. Earlier in the course, we focussed on the notion of threads of execution. Now we want to come back to the notion of a process, and in particular to the matter of managing a process's memory.

B. Though we will use the term “process”, other terms have been and are still used for the same concept, including “job” and “task” (though each term is also sometimes used to mean something different.)

C. PIDs and PCBs

1. To facilitate references to the process, operating systems typically assign each process a unique process ID (generally called a PID) at the time it is created. Typically, this is an integer, which starts at 1 when the system is first started and increases continually as it is running, so that no two processes ever have the same pid. (If a 64-bit integer is used for the pid, then if one process is created every microsecond the set of possible pids will run out in about a million years!)

2. The operating system uses a data structure called a process control block (PCB) to record vital information about a process.

   a) Its PID

   b) Resources such as memory and files that are assigned to it.

      Many systems also incorporate some provision for limiting the quantity of resources any once process can be given

      SHOW man page for getrlimit

   c) In systems where a process could only have a single thread of execution, information about the thread might also be stored in the PCB instead of in a separate PCB.

D. One of the most important sets of services an operating system provides is those related to the management of processes - including:

1. Process creation. Most systems allow a process that is running to create a new process.

a) The creator process is called the parent and the new process is called a child. (By repeated use of the create process service, a parent could create multiple children, of course.)

b) Various aspects of the relationship between a parent and child are handled differently by different systems:

   (1) Memory image:

      (a) Unix: When a child process is created by the fork() system call, it shares its parent's text and inherits a copy of all its parents data (data, heap, and stack). The child typically acquires its own, separate text by using the exec() system call.

         As a consequence, following a fork() both the parent and the child are executing the exact same code, beginning with the return from the call to fork(). The only difference is in the value returned from fork()

         i) In the parent, it is the PID of the newly-created child.

         ii) In the child, it is 0.

            Code like the following is typical in shells:

```
char * programToRun;
int childPID;
...
childPID = fork();
if (childPID == 0)
    // Begin running programToRun
else
    // wait for process whose PID is childPID to complete
```

      (b) Windows: The CreateProcess() system call specifies - among other things - the name of the application that is to be run by the child.

(2) Resources: As we have seen, most systems impose some upper limits on how large a quantity of various system resources a process can claim (e.g. memory, open files etc.) On some systems, the resources used by a child are counted against its parents quota; on others each child has its own quota.

(3) After new process creation, both the parent and child continue to execute in parallel. However, most systems provide a means by which the parent can wait until the child completes before proceeding further.

(4) Possibility of the child living on after the parent terminates.: On many systems termination of a parent process automatically causes its children to terminate as well.

DEMO: ssh to Linux, ps -xl. Discuss the following

COMMAND
UID:
PID: pid
PPID: (pid of parent)
STAT: S = waiting on an event (completion of child)
    R = ready or running
   (s = session leader, + = foreground process)

2. Process termination

a) A process may terminate itself by invoking a suitable system service.

b) On many systems, a parent process may also terminate one of its children through a system service call.

c) Many systems provide a mechanism whereby a process belonging to a suitably-privileged user can terminate someone else’s process as well.

3. Resource allocation. While CPU time is allocated automatically to a process (though there may be a provision for imposing a limit), other resources are typically allocated to a process upon request.

    a) When a process is first created, an initial allocation of memory is given to it, but operating systems typically provide a system service that allows more to be requested as the program runs.

    b) The act of opening a file is actually a form of resource allocation.

    etc

E. Multiprogrammed operating systems were first developed to allow multiple independent processes, but it soon became apparent that there were benefits to solving certain kinds of problems by the use of two or more cooperating processes working together on a common task. (An application of the principle of modularity that pervades computer science).

1. We have already seen how this works out in terms of separate threads in a single process, but the idea of cooperating processes actually precedes this concept, and may involve different programs.

2. One approach that arose in Unix is quite limited though often very useful - the use of something called a pipeline.

3. Consider the following example: We want to spell-check a document that uses a non-standard vocabulary (e.g. perhaps it's in a language that our regular spell checker does not recognize.) We have available a dictionary file for our vocabulary; what we want to do is to identify words in our document that do not appear in our dictionary.

For simplicity, assume that our document and dictionary are both ordinary text files. Also assume that writing a special-purpose program does not seem like a good option.

Here's an example (the language used is obbish! The dictionary has been made fairly minimal since this is only an illustration!)

PROJECT pipedoc.txt, pipedict.txt

We could proceed as follows.

a) Unix includes a transliterator utility called tr which performs transliteration. It is fairly easy to use this to take a text document and change the case of all letters to lowercase.

   DEMO: `tr A-Z a-z < pipedoc.txt > tmp1.txt`

b) This same program can be used a second time to replace a sequence of characters other than those that can appear inside words with a single newline. If we apply this to the output produced by the first use of tr, we ge a file containing each word, all lowercase, on a separate line.

   DEMO `tr -cs a-z '\012' < tmp1.txt > tmp2.txt`

c) Another utility called sort can sort a file into alphabetical order, with an option to squeeze out duplicates.

   DEMO: `sort -u < tmp2.txt > tmp3.txt`

d) Finally, the comm utility can be used to compare this file to the dictionary, reporting any lines that appear in the word list from the document but not in the dictionary (which are potentially misspellings).

   DEMO: `comm -23 tmp3.txt pipedict.txt`

e) Of course, a couple of problems with this approach is it involves four distinct steps and litters the directory with three temporary files that should be deleted (but may not be!)

4. An alternative approach is the use of a Unix pipeline. The Unix shells allow the user to specify several commands separated by '|'. Each command is run in a separate process, with the standard output of the first process connected to the standard input of the second, the standard output of the second connected to the third ...

   DEMO: `tr A-Z a-z < pipedoc.txt | tr -cs a-z '\012' | sort -u | comm -23 - pipedict.txt`

5. Actually, if this command were to be used frequently, it would be possible to turn it into an executable shell script by putting it in a file with file type .sh and making it executable.

   PROJECT, DEMO pipescript.sh

II. **Memory Management**

A. One of the most important resources an operating system manages on behalf of processes is memory.

B. With the advent of stacked job batch systems, it became necessary to partition memory into distinct monitor and a user program regions. With the advent of multiprogramming, the number of regions needed grew - one for the operating system, plus separate regions for each process.

   1. At this point, we need to digress a bit to talk about how memory addresses are handled. (We discuss this in much more detail in CPS311).

      a) Memory is composed of a series of distinct locations, each of which has its own numeric address. For example, a 1 GB memory would consist of 2^{30} bytes, each with an address lying in the range $0..2^{30} - 1$.

b) Each variable appearing in a program is stored at some location in memory, known as its address.

   If the variable is too big to fit in one location, it uses a series of consecutive locations, with the address of the first of these locations serving as the address of the variable.

   Example: An integer variable typically requires 4 bytes of space, So a global variable declaration like

   ```
   int x;
   ```

   would result in the variable x being assigned four consecutive bytes in memory. If the variable is assigned to locations 1000-1003, then it would be referred to by the address 1000.

c) When a program needs to reference a variable, it specifies the address of the memory location where that variable resides - e.g. if the variable x is stored at locations 1000-1003, then

   ```
   x++;
   ```

   might be compiled into a machine language instruction like

   ```
   Add 1 to the integer at memory address 1000
   ```

d) A similar situation holds when code needs to refer to other code - e.g. when code needs to call a procedure. To do so, it must specify the address in memory of the location where the procedure resides - e.g. if the procedure foo() begins at memory location 2000, then

   ```
   foo();
   ```

   might be compiled into a machine language instruction like

   ```
   Call the procedure beginning at memory address
   2000
   ```

2. Now we face an interesting problem: when we have several processes resident in memory at the same time time, how do we ensure that each uses a different set of memory addresses for its variables and code? (Consider the consequences that would follow if this were not the case!)

   a) One solution that was used in the early days of multiprogramming was to require that each program be written to use a different set of addresses.

      Of course, this is only possible if we knew ahead of time that a fixed set of programs would be running at the same time; it would be useless in the case where arbitrary programs can be run at any time - even two different processes running the same program.

   b) Another solution that was used was to modify the addresses appearing in a program when it was loaded so that it would use addresses not currently in use.

      Though workable, this is cumbersome.

C. The most commonly used approach is for the hardware to incorporate a component that performs address mapping (commonly called a memory management unit.)

   PROJECT Figure 6.4 from Hailperin

   1. Programs are compiled to use virtual addresses without having to be concerned about what other programs are running. (In fact, usually a standard set of conventions are used for assigning addresses when a program is compiled, so all programs end up being compiled to use the same addresses.)

   2. But when a program is running as a process, each virtual code or data address it emits is translated by the MMU into a distinct set of physical addresses that it alone is using.

3. For this to work, the MMU must use a distinct set of mappings for each process. Typically, this is managed by a table that the operating system sets up.

   A consequence of this is that additional overhead is involved for a context switch between threads in distinct processes, as opposed to threads in the same process, since memory management information must also be modified. (This is typically minimized by having the mapping tables themselves reside in memory, with only the CPU register that contains the starting address of the tables to use needing to be changed during a context switch.)

D. A byproduct of using memory management is that it becomes possible to protect the operating system from user processes, and user processes from one another, by simply ensuring that different mappings are used for each.

   1. If no virtual address in a process maps to a given physical address, then there is no way that the process can access that location.)

   2. This can be made more fine-grained by extending the mapping capabilities to also specify whether a given region of physical memory can be read and written, or is read only (so the process can look at it but cannot change it.)

E. Though memory mapping was originally developed to solve the problem of ensuring that distinct processes reference distinct locations in physical memory, it can be used for other things as well.

   1. Rather than having an entire program in memory, it is possible to use disk as an extension of main memory by setting up the mapping tables for certain virtual addresses to indicate that they are referencing a location on disk, rather than main memory.

      a) Of course, accessing information on disk is <u>**much**</u> slower than accessing information in main memory - it takes on the order of

100,000 times longer. (So strategies are used to ensure that the code and data that is currently being used is all resident in memory.)

b) This does, however, allow a system to run programs whose sum total memory requirements exceed the available memory.

c) It was actually this usage that gave rise to the term *virtual* memory.

2. It is also possible to have library code (e.g. Windows DLLs) that is available to all processes on a system. If a process needs to use such a library, the operating system sets up its mapping tables to specify that certain virtual addresses in the process are mapped to the physical location where the library resides. (Typically, a read-only mapping is specified so that a process cannot modify a library other processes need to use!)

III. Inter-Process Communication

A. We talked earlier about the notion of having two or more processes cooperate to fulfill a common task, and we at one mechanism for doing this: pipelines. However, pipelines are primarily useful for the special case where we can string together existing programs to accomplish our task on a single system. Often, our task calls for special software and/or must be done using multiple systems. This, of course, raises the question of how two or more processes can share information in a general way.

B. One approach is shared memory.

1. In the standard multiprogramming model, each process has its own memory which is separate from that of all other processes.

2. Shared memory allows two processes to share some amount of memory in common (generally a subset of the memory allocated to each).

\

    a) One process (the “owner” of the memory) executes a system service that specifies that some region in its memory space is to be made available for sharing.

    b) The other process (or processes) then “attach” to this shared memory.

    c) The operating system manages this by having the mapping tables used for both processes map certain virtual addresses in the two processes to the same physical locations in memory.

3. Operating systems that support shared memory (and not all do) may allow the owner process to make shared memory available for other processes to read but not write, or may allow both reading and writing by other processes.

C. A second approach to inter-process communication is message passing.

1. The operating system supports two services - a send service that allows one process to send a message to another, and a receive service that allows a process to receive a message from another.

    a) The send service often completes immediately - the OS holds the message until it can be delivered.

    b) The receive service typically waits until a message arrives.

    c) From the operating system’s standpoint, the message is just a series of bytes - the sending and receiving processes must put their own interpretation on it.

2.A message passing facility in the operating system is often used as the foundation for a strategy known as "Remote Procedure Call".

a) We will briefly discuss an object-oriented version of this. In brief, a process is allowed to access methods of two different kinds of objects:

   (1) Local objects, residing in its own memory space.

   (2) Remote objects, residing in the memory space of another process, wh may the located on the same machine, or may be truly remote.

b) There is a certain amount of overhead involved in setting up an RPC connection. However, once this is done, the code needed to invoke a method of a remote object is essentially the same as the code needed to invoke a method of a local object.

   (1) When a method of a remote object is invoked, the RPC mechanism marshalls its parameters into a message, which is sent to the "remote" machine. Meanwhile, the process that invokes the message is placed into a waiting state until a reply message is received.

   (2) When the "remote" machine receives the message, it unmarshals the parameters, invokes the message, and then marshals the method's result into another message, which it sends back to the original machine. (If the method is void, a return message is still sent, but without any information.)

   (3) When the reply message arrives, the result is unmarshalled and returned to the calling process in just the way that it would have gotten a result back from a method on a local object.

c) We will discuss RPC more thoroughly in the networks section of the course, since it is most often used when the processes involved are on different machines.

3. Once again, not all operating systems provide a message-passing facility - though most do (even those supporting shared memory as well).

D. How do we compare these last two approaches to inter-process communication?

ASK

1. Shared memory is much faster than message passing, because shared memory only involves operating system overhead for initial setup, while message passing involves operating system overhead for every operation.

2. However, shared memory is restricted to processes running on the same machine. Message passing is easily extended to support communication betweeen cooperating processes located at different physical locations - the message is simply sent over a network.

3. Shared memory also requires some mechanism for synchronizing the two processes - while this is inherent in message passing.

4. Finally, shared memory adds significant complexity to the memory management portion of the operating system.

E. In the case of threads within the same process, shared memory can always be used, since they all share the same memory. However, message passing can also be used, as we shall explore in lab. (The motivation in this case has to do with avoiding the need for explicit synchronization of access to shared memory.)