COMP 730 (242) Class Notes
Handout 2: Process Management

1 Scheduling and Context Switching

Reading chapters 3,4,5 Comer.

We saw earlier that an operating system gives the illusion of concurrency in a single processor system by switching the CPU among different processes, running one for a period of time before moving to another. We now study two components of this switching:

Context Switching, which consists of stopping one process and starting a new one.

Scheduling, which consists of choosing a new process among the processes that are eligible for execution.

1.1 Xinu Scheduling Policy

A **scheduling policy** determines how a new process is chosen for execution. The policy should be distinguished from the mechanism used to enforce it. We describe here the Xinu scheduling policy. The mechanisms are described later.

The Xinu scheduling policy has three components:

Each process is associated with a **priority**.

The highest priority ready process is always chosen for execution.

Among processes with equal priority scheduling is **round-robin**. By **round-robin** we mean that processes are selected one after another so that all members of a set have an opportunity to execute before any member has a second chance.

1.2 Process Table

The **process table** is a data structure maintained by the operating system to facilitate context switching and scheduling, and other activities discussed later. Each entry in the table, often called a **context block**, contains information about a process such as process name and state (discussed below), priority (discussed below), registers, and a semaphore it may be waiting on (discussed later). The exact contents of a context block depends on the operating system. For instance, if the OS supports paging, then the context block contains an entry to the page table.

In Xinu, the index of a process table entry associated with a process serves to identify the process, and is known as the **process id** of the process.

1.3 Process State

The system associates a process with a **state**, which helps it keep track of what the process is doing. Two of these states, used for context switching and scheduling, are **current** and **ready**. Other states will be discussed later.

The single process currently receiving CPU service is in the *current* state; other processes eligible for CPU service are in the *ready* state (Some processes are not eligible for CPU service, for instance a process waiting on a semaphore).

1.4 Xinu Mechanism for Context Switching and Scheduling

The Xinu mechanism for context switching and scheduling consists of three components, which are described below.

1.4.1 Ready List

The ready processes in the system are kept in a 'queue' called the **ready list**. This list is sorted by the priority of the processes; the lowest priority process appears at the head of the list and the highest priority process is at the tail. Processes of the same priority are sorted by the order in which they are to get service. Thus when a new process is inserted, it is inserted not at the end of the list, but at a point determined by its priority. The current process is not on this list, its id is stored in a global integer variable called **currpid**.

1.4.2 resched

resched is a routine that is called by the current process when rescheduling is to take place. It is called not only when the time quantum of the current process expires but also when a blocking call such as wait is invoked by the current process or when a new process (of potentially higher priority) becomes eligible for execution. (We shall discuss later the exact conditions that determine when a rescheduling takes place)

The routine does the following:

Choosing a New Process: It chooses the new process to execute based on the scheduling policy described above. Thus it looks at the process at the tail of the ready list. If the priority of this process is lower than the priority of the current process, then the current process, if it is executable, retains control of the CPU and the routine returns. (The scheduling policy dictates that a lower priority process in the ready list will be executed only if there are no higher priority processes in the list). Otherwise the process at the tail of the process is chosen as the new process and the following steps are taken.

Change of State of New Process: The new process is removed from the ready list, its state is changed from *ready* to *current*.

Allocation of Time: The new process is is allocated a time interval to execute (we shall study later how this is done). This time interval is the maximum time it will execute control before rescheduling takes place.

Change of State of Old Process: The new state of old process depends on what caused the reshedule. If a timer interrupt caused it, then the new state is *ready*. If on the other hand, a blocking call caused it, then the new state depends on the kind of blocking call. For example, in case of a semaphore wait call, the new state is *wait*. How does resched() set the right value of the state? More important, should it even be aware of state values such as *wait* assigned by higher layers?

The answer to the second question is No if we wish to strictly adhere to a layered architecture. Therefore, in Xinu, *resched()* assigns state values defined by it and leaves to higher level layers the job of assigning state values defined by them. The routine *resched* is never called directly by the user process. It is called indirectly by some other system provided routine such as *wait* on a semaphore. This routine may change the state of the process to a value defined by it. For instance *wait* changes the state of process to *wait* (as we will discuss later). At the point *resched* is called, the process is executable only if its state is still *current*. *resched* uses this information to determine if the old process needs to be put in the ready list. If the state is *current* it changes it to *ready* and inserts the process behind other processes with the same priority (so that it is picked last among processes with the same priority, thus ensuring round robin scheduling among processes with the same priority), and in front of processes with lower priority (so that a higher priority process is scheduled before a lower priority one). Otherwise it leaves the process untouched since some other routine took care of changing the state of the process and putting it in an appropriate list.

Call to ctxsw: Finally the routine *ctxsw* (discussed below) is called. This routine does work that cannot be done in a high level language (compared to assembly language) like C. In the LSI (and 8088) version, it passess as parameters, pointers to the old and new register area. When an assembly routine is called from a C routine, or vice versa, we must, in the assembly routine, do pre and post call work that is normally done by routines inserted by the compiler. This in turn, means, that we must understand the nature of this work. In addition, we must be aware of the stack layout at call time. Chapter 2 in the text book provides this information in detail for the C compiler, which is summarized below.

When a procedure is called, the stack consists of the return address, followed by the arguments, from first to last, followed by the registers. In type-safe languages the arguments are pushed in the sequence in which they are encountered while scanning the program, but in C they are pushed in the reverse order. This allows a variable number of arguments to be provided to the called routine - the first argument, which is at the well known location just below the return address, can indicate the number of arguments, which can be used by the called routine to access the remanining arguments. Both C's printf and Xinu's create use this feature.

It is the caller's responsibility to push the return address and arguments at start of call, and pop the arguments when the callee returns. It is the callee's responsibility to pop the return address before releasing control to the caller. It is also the callee's responsibility to save the register's it will use at the beginning of the call, and restore them at the end. In the LSI implementation, these two tasks are automated by two assembly routines, csv and cret, respectively, which save registers 2 to 5.

1.4.3 ctxsw

The routine **ctxsw (oldRegArea, newRegArea)**, described in the text (pg 60), does the following:

Saves the registers of the old process in the process table entry for it. Registers R1-R5, the stack pointer, the program counter, and the process status are all saved. (Compare this with register saves in a procedure call. Why are all registers being saved rather than only the ones ctxsw uses?) It has to be careful while saving a value for the PC. It does not save the address of any of the remaining instructions in *ctxsw* (why?). Instead, it saves the return address of ctxsw. Thus, when the old process is resumed, it starts executing after the statement in *Resched* that calls *ctxsw*. The stack pointer must be adjusted to make it look as if a return from ctxsw occurred. In the LSI version, this is done by popping the return address from the stack. The parameters will be popped by the calling procedure.

Loads the registers of the new process from the saved values in the process table entry for it. In particular, loads the saved stack pointer of the new process's context block, switching stacks. Again, care has to be taken with the program counter. This register can be loaded only after the rest of the state has been restored. The routine uses the *rtt* instruction to transfer return control to the new process. This instruction loads the PC and PS from values saved in the stack. Thus *ctxsw* makes sure that these values are pushed on the (new process's) stack before rtt is called. (What routine will the new process be executing when control is transferred to it?)

1.4.4 The Null Process

The code in *resched*, when deciding on a new process to execute, does not bother to verify if the ready list is empty. It assumes that one process is always available to catch interrupts. Xinu ensures that a ready process always exists, by creating an extra process, called the *null process*, when it initializes the system. This process has process id 0, its code consists of an infinite loop, and it has priority zero (why?). It is important to always have at least one ready process (whose stack can be used) to handle interrupts.

2 Process Resumption and Suspension

Process **suspension** and **resumption** are used to *temporarily* stop a process from executing and then restart it again.

2.1 Semantics

A process may suspend itself or another process may suspend it.

A process to be suspended should be in the *ready* or *current* state.

Suspended processes go into the *suspended* state.

A process can be resumed only if it is in the *suspended* state.

A resumed process goes to the *ready* state (why not to the *current* state?)

The resume call returns the priority of the suspended process at the time that resume was called.

The suspend call returns the priority of the suspended process just before suspend terminates.

The reasons for returning priorities are obscure, but the fact that the priority of a process can change during a system call is interesting.

Look at transition diagram in figure 5.1 in the text book.

2.2 Implementation

Process resumption and suspension are implemented by the procedures *resume* and *suspend* respectively.

2.2.1 Procedure resume

Takes as argument the process id of the process to be resumed.

Returns an error value if argument is not a valid process id or is not in the suspended state.

Puts the process in the *ready* list.

Asks for rescheduling. (why?)

Stores the value of the priority of the resumed process *before* the process is rescheduled (why does this value have to be stored?).

Disables interrupts while accessing the process table to read the priority of the resumed process. If interrupts are not disabled, a clock interrupt may ask for rescheduling. Thus some other process may change the priority of the resumed process before the resuming process has a chance to read the priority.

2.2.2 Procedure suspend

Takes as argument the process id of process to be suspended.

Returns an error value if process id is invalid or process is not in the ready or current state. Otherwise returns the priority of the process just before suspend terminates.

Changes state of process to *suspended*.

If process is in the *ready* state it is removed from the ready list. (Should it put the process in some other list?)

If process is in the *current* state, it calls *resched* (why?).

disables interrupts while accessing process table.

3 Process Termination

A process may terminate a) when it finishes execution of its procedure or b) as a result of a **kill** request made by the same or another process.

Process termination involves releasing resources held by the process and removing all traces of it.

3.1 Implementation

The procedure **kill** handles both kinds of process termination. It may be explicitly called by a process that makes a *kill* request or it may be called implicitly when a process terminates normally.

3.1.1 Tasks Performed

(Error checking, ensuring mutual exclusion while accessing process table, and other functions common to *resume* and *suspend* will not be mentioned in the remaining implementation descriptions)

returns stack space.

frees process table entry, making it available for reuse.

remove process from any list it is on,

increments semaphore count if necessary,

reschedules if process killed itself. reschedule needs the "freed" process stack for passing arguments to ctxsw. Fortunately, this is not a problem, since interrupts are disabled between the time the stack is freed and ctxsw finishes processing the arguments. As a result, the freed stack will not be overwritten by another process while the arguments are accessed. However, it can cause problems when implementing Xinu threads inside a Unix process. In this case, you use C/Unix routines to allocate and deallocate memory. Depending on the compiler/OS, accessing unallocated memory can result in a memory access fault. This is the reason that in the first assignment, you are advised to not dellocate space.

Incrementing the semaphore count in terminate violates the layering principle as process management is now aware of process coordination. In general, layered systems often have to resort to *upaccesses*, which are accsses to higher level layers. These upaccesses should reveal as little details of the higher level layers as possible if the benefits of layering are to retained. It is possible to do better than the Xinu implementation by having each higher level layer register a cleanup routine that is called by process management rather than access the higher-level semaphore table directly. The cleanup routine can then deallocate resources it created. (Why is this approach better than the Xinu implementation?)

4 Process Creation

An existing process requests for a new process to be created.

The new process does not begin execution but is in the *suspended* state.

4.1 Implementation

Implemented by routine *create (procaddr, ssize, priority, name, nargs, args)*.

4.2 Arguments:

procedure to be executed

stack size

priority

name
number of args and args

4.3 Tasks Done

Create stack space

Create process table entry

Fill priority, name, registers and other fields of process table entry. (what value should the PC contain?)

Simulate a call to procedure to be executed. Copy arguments to the stack of the new process and fill return address of a routine (*userret*) that calls the *kill* procedure. Thus kill also handles normal termination.

Returns process id.

5 System Calls

These are procedure calls that a user program may make to access the services provided by the OS. So far we have seen four system calls: *create*, *kill*, *resume*, and *suspend*. It provides other calls that also have to do with process management,

5.0.1 Getpid()

This routine returns the process id of the process making the calls.

5.0.2 Getprio (pid)

Returns the priority of a process.

5.0.3 Chprio (pid, newprio)

Change the priority of a process and reschedule if necessary.

6 Need for Multiprocessing

Why do we need multiprocessing, when computation, instead of being speeded, may actually be slowed down in a single processor system (because of context switching overhead and overhead of making system calls such as *create* and *terminate*)? Why is sequential processing not sufficient?

Multiprocessing is useful for several reasons, which are discussed below.

6.1 Time Sharing

In the absence of multiprocessing, a single application is run at a time. Thus a single user is serviced at a time. Multiprocessing is necessary for time sharing systems, which allow several users to get service simultaneously.

6.2 Different Applications per User

It is also useful to have several applications running simultaneously on behalf of the same user. For instance an editor and a compiler could run together. The editor could be higher priority so that the compiler runs only when the editor is waiting for user input.

6.3 Several Processes per Application

A single application can be broken up into several parts, each one associated with a separate process. This division of labour often makes the program easier to write. As an example consider a screen manager that displays several windows simultaneously and accepts user input from each window. In a sequential processing system, exactly one process is assigned to this application. It would maintain a "context block" for each window and "switch" among these blocks based on user input. The program for this process would look as follows:

```
initialize all windows.
loop
    wait for user input
    determine the window to which it applies
    load state of window
    service input
    save state
end loop
```

In a multiprocess system, a process could be associated with each window. The state of a window is stored in the variables of its process, and the operating system is responsible for saving and restoring this state and switching among the processes. Moreover, it can support urgent windows ("abort missile launch") by giving higher priority to processes that manage them. Thus each process is concerned only with servicing input, and not with saving and restoring state.

Perhaps more compelling examples are found in the Web context. Multiprocessing allows a Web browser to execute different animations (shown on a Web page) as separate threads. In the absence of multithreading, the browser would be responsible for context switching between the animations. Even more compelling, multiprocessing allows an HTTP server to execute different pieces of user-supplied code as independent threads or "servlets" written potentially by different people. Thus, multiprocessing allows different people to independently write code executed in a single application (address space) such as a Java virtual machine in the example above.

6.4 Concurrent Computing

The different processes can execute on different computers in a multiprocessing system. Thus computation may be speeded up. As an example consider a checkers-playing application that evaluates different board positions. Each board position could be evaluated by a different process running on a different computer. Similarly, independent animations can be executed concurrently by different processors.