Nachos Threads

Sivasankar Ponnambalam

Dr.Douglas Niehaus

Parts of a Thread

- **Thread Status** : Every thread in Nachos contains a variable status which defines the current status of the thread.
- Different values for status are JUST_CREATED, RUNNING, READY, BLOCKED, ZOMBIE.
- Status activity of the different threads.
- Thread status is accessed with the functions *getStatus* and *setStatus*.
  - enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED, ZOMBIE };
  - ThreadStatus status;

Thread Status

- A thread in Nachos (as in every thread package) requires some state storage so it can be interrupted and restarted.
- Status - set of status registers, general purpose registers and a PC.
- Storing state Nachos :
  - thread is running a user program on a MIPS simulator,
  - while the thread code itself runs on a host (i386) processor.
- Two sets of state registers:
  - unsigned int machineState[MachineStateSize];
  - int userRegisters[NumTotalRegs];

Thread State

- *MachineState* is where the host processor state is stored.
- A Nachos thread running on an i386 will have space for the EIP, EFLAGS, segment and general purpose registers in this array.
- The *userRegisters* are then the state of the thread on the simulated MIPS machine.
- The state of the MIPS processor is saved by *SaveUserSate* function.
- The state of the i386 processor is saved via assembly language contained in the file switch.s.

User Execution Environment

- Nachos thread represents a process.
  - it provides the execution environment for a user program.
  - it has to store the user program in a format that is accessible to the MIPS simulator on which the program runs.
  - Each thread contains a pointer to an address space object which provides a location for the actual user code.
  - Inside the MIPS machine, addresses are resolved into references into the address space object.
- AddrSpace *space;
- In addition to address space, thread structure also provides file descriptors similar to Unix.
  - Which allow user programs to open, read and write to files in the system.
- FDTEntry *FDTable [MAX_FD];

Family Relationships

- When a thread forks a new thread,
  - First thread = Parent
  - New thread = Child.
- A Family relationship is said to exist between these two processes.
- The operating system is responsible for keeping track of these relationships to ensure proper operation of various system calls
  - for example *Wait* will wait for a child to exit.
- In Nachos, the parent of a thread is recorded in the ParentPtr.
  - Thread* ParentPtr;
  - Thread* Get_Parent_Ptr();
  - int Set_Parent_Ptr(Thread * parent);

- When a child exits, it must “wait” for it's parent thread to wait on it.
- In Nachos each thread contains a list of children it needs to wait on.
- When a child exits, it uses the *ParentPtr* to add itself to that threads waiting list and place itself on the *ChildExited* Semaphore. When the parent thread then calls *Wait*, a child is removed from the *ChildList* and the exit value is recorded.
- The Semaphore is then used to signal the child that it can exit.
- These member functions are shown below.
  - void Queue_Child(Thread * child);
  - Thread* UnQueue_Child();
  - List ChildList;
  - Semaphore ChildExited

- For informational purpose, Nachos also stores the number of children in a variable Children.
- Useful when a parent thread is going to exit and in so doing will orphan children (i.e. at the time a thread calls exit, Children is greater than 0).
  - int Children;
  - void Add_Child();
  - void Remove_Child();
  - int Get_Num_Children();

Creation of a Thread

- A thread is created via the Fork system call.
- When a thread calls Fork, control is passed to the OS.
- Which then executes *Sys_Fork* function.
- This function is responsible for allocating the address space of the new thread, copying the contents of the parents memory into the child, and setting the parent pointer.
- When the thread-specific structures are allocated, the system then calls the thread Fork function
- pid = t->Fork((void(*)(size_t))Do_Fork, (size_t)(dummy));

Creation of a Thread

- The system passes the address of the function *Do_Fork* to the thread Fork function, which will place it on the new thread's stack.
- The function *Do_Fork* - first thing the new thread executes when it is switched in to run.
- *Do_Fork* then loads it's state onto the MIPS simulator and begins to run it's user program.

- It should be noted that until the function *Do_Fork* actually begins to run and calls
  - machine->Run(), the new thread is not running on the MIPS processor.

Context Switching

- State of a running thread Saved into its thread structure
- New thread's state is restored.
- New thread begins to run and the old thread is put back on the ready list.
- In Nachos, since the OS-side code is running on a host (i386) processor, while the user-program is running on a MIPS processor, there are two sets of state to save.
- The MIPS processor state is saved by the function SaveUserSate.
- User registers are accessible to the OS through the ReadRegister method.

```
Void Thread::SaveUserState()
{
     for (int i = 0; i < NumTotalRegs; i++)
        userRegisters[i] = machine->ReadRegister(i);
}
```

Context Switching

- Saving the state of the i386 processor is much more complicated.
- This requires assembly language which collects the values of the program counter, segment registers, stack, etc.
- These registers are saved one at a time in the *machineState* array.
- Since the program is executing code while it is saving it's program counter, it will save a location inside the context switch routine.
- This routine is where the thread will start executing again when it's context is restored.

Mechanics of Context Switch

```
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;

    if (currentThread->space != NULL) {
        currentThread->SaveUserState();
        currentThread->space->SaveState();
    }

    currentThread = nextThread;
    currentThread->setStatus(RUNNING);

   SWITCH(oldThread, nextThread);

    if (currentThread->space != NULL) {
        currentThread->RestoreUserState();
        currentThread->space->RestoreState();
}
```

- Inside the assembly code of the SWITCH call, the PC for the old thread (one we're switching out) is saved.
- This means that when the old thread is restored at some future point, it will be operating at the place it's PC was stored.
- It will appear to come out of the SWITCH call.
- Currently running thread saves it's MIPS processor state, then the nextThread is set to RUNNING.
- When the SWITCH statement returns, the currentThread (now nextThread) will be running on the i386 processor.
- We restore the state of the new thread on the MIPS processor, which continues to run its user program.
- A user level program will not notice that it has
- been swapped out in favor of something else.

Destroying a Thread

- Threads can destroy themselves by calling the Exit system call.
- Exit will delete the excess storage of the thread and place it on it's parent's *ChildExited* list.
- The child will be woken up when the parent Wait’s for it, at which point all remaining storage for the thread will be reclaimed.