Nachos Threads and Concurrency

B.Ramamurthy

Thread

- Unit of work
- A process has address space, registers, PC and stack (See Section 3, page 9.. in A Roadmap through Nachos for the detailed list)
- A thread has registers, program counter and stack, but the address space is shared with process that started it.
  - This means that a user level thread could be invoked without assistance from the OS. This low overhead is one of the main advantages of threads.
  - If a thread of a process is blocked, the process could go on.
  - Concurrency: Many threads could be operating concurrently, on a multi threaded kernel.
  - User level scheduling is simplified and realistic (bound, unbound, set concurrency, priorities etc.)
  - Communication among the threads is easy and can be carried out without OS intervention.

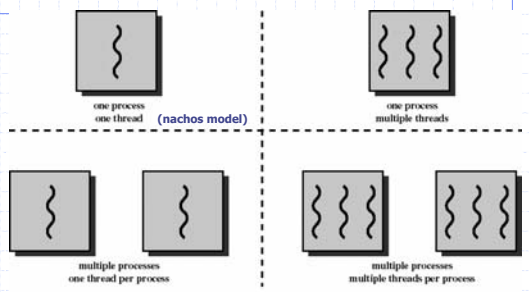
Thread requirements

- An execution state
- Independent PC working within the same process.
- An execution stack.
- Per-thread static storage for local variables.
- Access to memory and resources of the creator-process shared with all other threads in the task.
- Key benefits: less time to create than creating a new process, less time to switch, less time to terminate, more intuitive for implementing concurrency if the application is a collection of execution units.

Threads and Processes

- A thread is a unit of work to a CPU. It is strand of control flow.
- A traditional UNIX process has a single thread that has sole possession of the process's memory and resources.
- Threads within a process are scheduled and execute independently.
- Many threads may share the same address space.
- Each thread has its own private attributes: stack, program counter and register context.

Threads and Processes



Thread Operations

- Basic Operations associated with a thread are:

Spawn : newly created into the ready state

Block : waiting for an event

Unblock : moved into ready from blocked

Finish : exit after normal or abnormal termination.

Nachos Threads (Section 3, .. RoadMap)

- Nachos process has an address space, a single thread of control, and other objects such as open file descriptors.
- Global variables are shared among threads.
- Nachos "scheduler" maintains a data structure called ready list which keeps track of threads that are ready to execute.
- Each thread has an associated state: READY, RUNNING, BLOCKED, JUST_CREATED
- Global variable currentThread always points to the currently running thread.

Nachos Thread Description and Control

- Thread specific data: local data, stack and registers such as PC, SP.
- Control:
  - Thread creation (Ex: fork)
  - Thread schedule (Ex: yield)
  - Thread synchronization (Ex: using barrier?)
  - Code for execution (Ex: fork's function parameter)

Nachos Thread Class

Thread

//operations
Thread *Thread(char *name);
Thread *Thread(char *name, int priority);
Fork (VoidFunctioNPtr func, int arg);
void Yield(); // Scheduler::FindNextToRun()
void Sleep(); // change state to BLOCKED
void Finish(); // cleanup
 	
void CheckOverflow();// stack overflow
void setStatus(ThreadStatus st); //ready, running..
// blocked
char* getName();
void Print(); //print name

//data
int*stackTop;
int machineState[MachineStateSize];//registers
int* stack;
ThreadStatus status;
char* name;

Thread Control and Scheduling

- Switch (oldThread, newThread);
  // assembly language routine
- Threads that are ready kept in a ready list.
- The scheduler decides which thread to run next.
- Nachos Scheduling policy is: FIFO.

Nachos Scheduler Class

Scheduler

Scheduler();
~Scheduler();
 	
void ReadyTnRun(Thread* thread);
Thread* FindNextToRun();
 	
void Run(Thread* nextThread);
void Print();

List* readyList;

Barrier Synchronization

- Barrier is a synchronization primitive to synchronize among 2 or more threads. (n threads)
- Threads execution barrier synch wait until the expected number of threads execute the barrier synch primitive. Then they are all released.
- This is similar to "rendevouz" concept in Ada language.

Nachos Barrier.h

```
class Barrier {
  public:
    Barrier(char *debugName, int size);
    ~Barrier();
    char* getName() { return name; }    // debugging assist
    void barrierSynch();
    void print();

  private:
    char* name;                         // for debugging
    // plus some other stuff you'll need to define
};
```


Barrier Usage

```
//thread 1
Barrier bar = new Barrier ("sample", 3);
....
....
bar.barrierSynch(); // occurs a t(1)
//thread blocked
```

```
//Thread 2
bar.barrierSynch(); //occurs at t(2)
//thread blocked
```

```
//Thread 3
bar.barrierSynch();  //occurs at t(3)
//all threads are released
```

You are required to implement a kernel level barriers and corresponding system calls to barriers From user programs