Outline for Today’s Lecture

• Administrative:
  – Demo plan for program 3
  – Collect Homework 1, to be returned Tuesday
  – Program 4 is hard! Don’t delay!
  – Midterm next Thursday – open book, open notes.
• Objective for today:
  – Finish advanced topics in scheduling
  – Preview of Program 4
  – Answer questions on process management (there will be additional opportunities next Tuesday in prep for exam)

Beyond “Ordinary” Processors

Multiprocessors
  – Co-scheduling and gang scheduling
  – Hungry puppy scheduling
  – Load balancing
  – Affinity scheduling

Networks of Workstations
  – Harvesting Idle Cycles - remote execution and process migration

Laptops and mobile computers
  – Power management to extend battery life, scaling processor speed/voltage to tasks at hand, sleep and idle modes

Multiprocessor Scheduling

What makes the problem different?

• Workload consists of parallel programs
  – Multiple processes or threads, synchronized and communicating
  – Latency defined as last piece to finish.
• Time-sharing and/or Space-sharing (partitioning up the Mp nodes)
  – Both when and where a process should run

Architectures

Symmetric mp

NUMA

Node 2

Node 3

cluster
Affinity Scheduling
- Where (on which node) to run a particular thread during the next time slice?
- Processor’s POV: favor processes which have some residual state locally (e.g. cache)
- What is a useful measure of affinity for deciding this?
  - Least intervening time or intervening activity (number of processes here since "my" last time) *
  - Same place as last time "I" ran.
  - Possible negative effect on load-balance.

Processor Partitioning
- Static or Dynamic
- Process Control (Gupta)
  - Vary number of processors available
  - Match number of processes to processors
  - Adjusts # at runtime.
  - Works with task-queue or threads programming model
  - Impact on "working set"

Process Control Claims
Typical speed-up profile

Co-Scheduling
John Ousterhout (Medusa OS)
- Time-sharing model
- Schedule related threads simultaneously
  Why?
  How?
  - Local scheduling decisions after some global initialization (Medusa)
  - Centralized (SGI IRIX)
Effect of Workload

Impact of communication and cooperation

- context switch
- common state
- lock contention
+ coordination

Issues:

CM*'s Version

- Matrix S (slices) x P (processors)
- Allocate a new set of processes (task force) to a row with enough empty slots
- Schedule: Round robin through rows of matrix
  - If during a time slice, this processor’s element is empty or not ready, run some other task force’s entry in this column - backward in time (for affinity reasons and purely local “fall-back” decision)

Networks of Workstations

What makes the problem different?
- Exploiting otherwise “idle” cycles.
- Notion of ownership associated with workstation.
- Global truth is harder to come by in wide area context

Harvesting Idle Cycles

- Remote execution on an idle processor in a NOW (network of workstations)
  - Finding the idle machine and starting execution there. Related to load-balancing work.
- Vacating the remote workstation when its user returns and it is no longer idle
  - Process migration
Issues

• Why?
• Which tasks are candidates for remote execution?
• Where to find processing cycles? What does “idle” mean?
• When should a task be moved?
• How?

Motivation for Cycle Sharing

• Load imbalances. Parallel program completion time determined by slowest thread. Speedup limited.
• Utilization. In trend from shared mainframe to networks of workstations → scheduled cycles to statically allocated cycles
  – “Ownership” model
  – Heterogeneity

Which Tasks?

• Explicit submission to a “batch” scheduler (e.g., Condor) or Transparent to user.
• Should be demanding enough to justify overhead of moving elsewhere. Properties?
• Proximity of resources.
  – Example: move query processing to site of database records.
  – Cache affinity

Finding Destination

• Defining “idle” workstations
  – Keyboard/mouse events? CPU load?
• How timely and complete is the load information (given message transit times)?
  – Global view maintained by some central manager with local daemons reporting status.
  – Limited negotiation with a few peers
  – How binding is any offer of free cycles?
• Task requirements must match machine capabilities
When to Move

- At task invocation. Process is created and run at chosen destination.
- Process migration, once task is already running at some node. State must move.
  - For adjusting load balance (generally not done)
  - On arrival of workstation’s owner (vacate, when no longer idle)

How - Negotiation Phase

- Condor example: Central manager with each machine reporting status, properties (e.g. architecture, OS). Regular match of submitted tasks against available resources.
- Decentralized example: select peer and ask if load is below threshold. If agreement to accept work, send task. Otherwise keep asking around (until probe limit reached).

How - Execution Phase

- Issue - Execution environment.
  - File access - possibly without user having account on destination machine or network file system to provide access to user’s files.
  - UIDs?
- Remote System Calls (Condor)
  - On original (submitting) machine, run a “shadow” process (runs as user)
  - All system calls done by task at remote site are “caught” and message sent to shadow.

Remote System Calls

```
<table>
<thead>
<tr>
<th>Submitting machine</th>
<th>Executing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow</td>
<td>Remote Job</td>
</tr>
<tr>
<td>OS Kernel</td>
<td>User code</td>
</tr>
<tr>
<td>Regular syscall</td>
<td>Remote syscall</td>
</tr>
<tr>
<td>stubs</td>
<td>stubs</td>
</tr>
</tbody>
</table>
```


How - Process Migration

Checkpointing current execution state (both for recovery and for migration)
- Generic representation for heterogeneity?
- Condor has a checkpoint file containing register state, memory image, open file descriptors, etc. Checkpoint can be returned to Condor job queue.
- Mach - package up processor state, let memory working set be demand paged into new site.
- Messages in-flight?

Idleness is Powerful

Scheduling for Voltage Scaling
(adjust clock based on past window, no process reordering involved)

Introducing User Programs into Nachos

Conceptually:
Nachos thread encapsulates user program, remains the schedulable entity
Nachos Systems Call (Process)

userprog/syscall.h

• Spaceid Exec (char *name, int argc, char ** argv, int pipectrl) - Creates a user process by creating a new address space, reading the executable file into it, and creating a new internal thread (via Thread::Fork) to run it. To start execution of the child process, the kernel sets up the CPU state for the new process and then calls Machine::Run to start the machine simulator executing the specified program's instructions in the context of the newly created child process.

• Exit (int status) - user process quits with status returned. The kernel handles an Exit system call by destroying the process data structures and thread(s), reclaiming any memory assigned to the process, and arranging to return the exit status value as the result of the Join on this process, if any.

• Join (Spaceid pid) - called by a process (the joiner) to wait for the termination of the process (the joinee) whose SpaceId is given by the pid argument. If the joinee is still active, then Join blocks until the joinee exits. When the joinee has exited, Join returns the joinee's exit status to the joiner.

StartProcess(char *filename)
{
  OpenFile *executable = fileSystem ->Open(filename);
  AddrSpace *space;
  if (executable == NULL) {
    printf("Unable to open file %s \n", filename);
    return;
  }
  space = new AddrSpace (executable);
  currentThread -> space = space;
  delete executable; // close file
  space -> InitRegisters(); // set the initial register values
  space -> RestoreState(); // load page table register
  machine -> Run(); // jump to the user program
  ASSERT(FALSE); // machine -> Run never returns;
  // the address space exits
  // by doing the syscall "exit"
}

ExceptionHandler(ExceptionType which)
{
  int type = machine -> ReadRegister (2);
  if ((which == SyscallException) && (type == SC_Halt)) {
    DEBUG('a', "Shutdown, initiated by user program. \n");
    interrupt -> Halt();
  } else {
    print("Unexpected user mode exception \"%d \"%d\", which, type);
    ASSERT(FALSE);
  }
}

Note: system call code must convert user-space addresses to Nachos machine addresses or kernel addresses before they can be dereferenced.
// first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;  // for now, virtual page # = phys page #
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;  // if the code segment was entirely
    // on a separate page, we could set its
    // pages to be read-only
}

// zero out the entire address space, to zero the uninitialized data segment
// and the stack segment
// bzero(machine->mainMemory, size); m for Solaris
memset(machine->mainMemory, 0, size);

// then, copy in the code and data segments into memory
if (noffH.code.size > 0) {
    DEBUG(a, "Initializing code segment, at 0x%x, size %d\n",
          noffH.code.virtualAddr, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
                      noffH.code.size, noffH.code.inFileAddr);
}

if (noffH.initData.size > 0) {
    DEBUG(a, "Initializing data segment, at 0x%x, size %d\n",
          noffH.initData.virtualAddr, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
                        noffH.initData.size, noffH.initData.inFileAddr);
}