Operating Systems

Processes & Threads
Introduction to Processes

- **A process** is an abstraction of a running program

- **Why do we need processes?**
  - Modern computers can do several things at the same time
    - Concurrently reading and writing memory, reading and writing to a disk drive, while participating in an active network connection.
    - If operating systems only supported one running program then the CPU would be idle while the disk drive was reading a file.

- **Goal:** package running programs into processes and devise a schedule for running the processes in order to maximize computer utilization
Processes

• Processes enable psudoparallelism
  – CPU can only do one thing at a time
  – Utilize idle CPU time to perform other work
  – By rapidly switching between running processes, and by using idle time wisely, the system provides the perception of parallelism

• Anatomy of a process
  – From the processes perspective it has its own virtual CPU
  – Because each process is a running program it has its own:
    • Program counter
    • Registers
    • Memory space
    • Private resources (semaphores)
The Process Model

• All runnable software on the computer is organized into a number of sequential processes
  – Includes the operating system, which consists of a number of running processes

• The CPU determines a schedule to run each of the processes for a specific duration
  – Process runs for a maximum amount of time before being stopped
    • Timeslice - typically 10 to 100 msecs
  – Process runs until it needs to wait for a system resource such as a disk drive
    • I/O is the most common

• The rapid switching between running programs is called multiprogramming
The Process Model

All processes run for a fixed amount of time

Logical PseudoParallel Execution
The Process Model

Process A, B, C are 90% I/O bound, Process D is 100% CPU Bound
The Process Model

By performing a task switch when a process blocks we can greatly improve the processes execution time.
Process versus Program

• Important to distinguish between a process and a program

• A **Program** is a compiled binary file that is created using a programming language

• A **Process** is a running program and consists of a program image, input, output and a current state
Process Hierarchies

- Processes that are created by a process become **child processes** of the parent process.

- Processes can be organized into process hierarchies (trees).

- When a parent process terminates, most operating systems will terminate all of the child processes.

- Some operating systems allow a parent to create a **detached child process** where the child is disassociated from the parent.
  - Child continues to run after the parent terminates.
Creating A Process: Unix

int pid = fork();

if (pid < 0)
    /* fork() failed – handle error */
else if (pid > 0)
    {
        /* parent code goes here */
    }
else
    {
        /* child code goes here */
    }
Creating a Process: Win32

//-------------------------------------------------------------
//Local variables to support executing arnrun
//as a child process
//-------------------------------------------------------------
STARTUPINFO  siStartInfo;
PROCESS_INFORMATION  piProcInfo;
DWORD     dwExitCode;
char    abArgs[64];
//-------------------------------------------------------------
//Initialize the the STARTINFO structure
//-------------------------------------------------------------
siStartInfo.cb = sizeof(STARTUPINFO);
siStartInfo.lpTitle = NULL;
siStartInfo.lpReserved = NULL;
siStartInfo.lpReserved2 = NULL;
siStartInfo.cbReserved2 = 0;
siStartInfo.lpDesktop = NULL;
siStartInfo.dwFlags = 0;
Creating a Process: Win32

memset(abArgs, '\0', sizeof(abArgs));
sprintf(abArgs, "%s %s", "arnrun.exe", "rsp");

//--------------------------------------------------------
//Start the new arnrun process.
//-------------------------------------------------------

lrc = CreateProcess(
    NULL,       //application name
    abArgs,     //command line
    NULL,       //process security attributes
    NULL,       //primary thread security attributes
    TRUE,       //handles are inherited
    0,          //creation flags
    NULL,       //use parent's environment
    NULL,       //use parent's current directory
    &siStartInfo, //STARTUPINFO pointer
    &piProcInfo ); //receives
                //PROCESS_INFORMATION
Create a Process: Win32

```c
if( lrc == 0 )
{
    lrc = GetLastError();
    printf( "Unable to create process. "
          printf(" Error code = %ld., lrc ");
    return;
}
```

//------------------------------------------------------------------------
//Wait for the new process to finish and
//acquire it's exit code
//------------------------------------------------------------------------
WaitForSingleObject(piProcInfo.hProcess, INFINITE);
GetExitCodeProcess(piProcInfo.hProcess, &dwExitCode);
CloseHandle(piProcInfo.hProcess);
```
Process States

- Process states
  - Ready: Process is able to run at next allocated timeslice
  - Running: Process is currently running
  - Blocked: Process can not run until some external event happens
- Processes can be modeled by a state transition diagram
- The following diagram show legal state transitions
Scheduler

• The scheduler is a component of the operating system
• Scheduler uses one or more algorithms to manage running and ready processes
• Scheduler may use a priority scheme to give certain processes more CPU time
  – The active program
• Scheduler uses queues to manage processes

<table>
<thead>
<tr>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Scheduler
Process Scheduling Queues

- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.
Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.
Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) => (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) => (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Processes can be described as either:
  - I/O bound process: spends more time doing I/O than computations; many short CPU bursts.
  - CPU bound process: spends more time doing computations; few very long CPU bursts.
Context Switches

• When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
  – All registers, process table
  – Initiated and driven by interrupts

• Context-switch time is overhead; the system does no useful work while switching
  – Considered overhead from the perspective of the user
  – Must optimize frequency of context switching (how long is a timeslice?)

• Time dependent on hardware support
  – How much does the hardware directly support versus how much is implemented in software in the operating system
Interrupts

• When a process can not continue because it is waiting for an external resource it can do the following:
  – Use polling to consistently check if the resource is available
    • Wasteful of CPU resources
  – Register an interest in gaining access to the external resource and go into a blocked state
    • Efficient use of CPU
• The external resource maintains a list of interested processes that require servicing in a queue
• When the external resource is ready to service the processes request, it send an interrupt
• The interrupt causes the operating system to change the state of the waiting process from blocked to ready
Implementation of a Process

• Operating system maintains a **process table**
• The process table contains one entry per process
• Each process table entry contains the following attributes:
  – Process Identifier (PID)
  – Process state
  – Program counter
  – Stack pointer
  – Register and Flag values
  – Memory Allocation
  – Open file pointers
  – Accounting information
  – Scheduling information

• **Goal:** Save enough information so that a process can be restarted (ready to running) as if it had never been stopped.
Process Relationships

• Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

• Resource Sharing Options
  – Parent and children share all resources
  – Children share a subset of the parent processes resources
  – Parent and children share no resources

• Execution Model
  – Parent and child execute concurrently
  – Parent waits until child terminates
  – When parent terminates all children terminate

• Cooperating processes
  – Independent processes can not affect each other
  – Cooperating processes can affect each other
    • Information sharing
Threads

• We will study threads because they can simulate processes
  – Allows us to investigate OS algorithms
• A process has an address space and single thread of control
• It is possible to have a single process that has multiple threads of control
• A thread is very similar to a process
  – Each thread gets scheduled just like a processes
  – Each thread gets a private address space
  – Each thread has a state (running, ready, blocked)
  – Each thread gets a unique ID
• A thread also gets access to global process information
• A thread is sometimes referred to as a lightweight process
Threads

Three processes with one thread of control

One process with three threads of control
Threads

Each thread sees its own local thread data and the common global process data
Threads

• In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  – Cooperation of multiple threads in same job confers higher throughput and improved performance.
  – Applications that require sharing a common buffer (i.e., producer–consumer) benefit from thread utilization.

• Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.

• User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU)
  – Fast, OS does not manage

• Kernel-supported threads (Mach and OS/2)
  – Slower, but more functional. The OS kernel manages the threads

• Hybrid approach implements both user-level and kernel-supported threads (Solaris 2)