Part I: Operating system overview:

Processes and threads
Overview

- Process concept
- Process Scheduling
- Thread concept
Process

These are all possible definitions:

- A program in execution, and process execution must progress in sequential fashion
- An instance of a program running on a computer
- Schedulable entity (*)
- Unit of resource ownership
- Unit of protection
- Execution sequence (*) + current state (*) + set of resources

(*) can be said of threads as well
A process includes
• text section
• program counter
• stack
• data section
• heap
### Data placement, seen from C/C++

<table>
<thead>
<tr>
<th>int a;</th>
<th>void func(int d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int b;</td>
<td>{</td>
</tr>
<tr>
<td>int c = 5;</td>
<td>static int e;</td>
</tr>
<tr>
<td>struct S {</td>
<td>int f;</td>
</tr>
<tr>
<td>int t;</td>
<td>struct S  w;</td>
</tr>
<tr>
<td>};</td>
<td>int *g = new int[10];</td>
</tr>
<tr>
<td>struct S s;</td>
<td>}</td>
</tr>
</tbody>
</table>

**Q.: where are these variables stored?**

**A.:**
- On stack: d, f, w (including w.t), g
- On (global) data section: a, b, c, s (including s.t), e
- On the heap: g[0]…g[9]
Process State

As a process executes, it changes state

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution
Process Control Block (PCB)

PCB records information associated with each process

• Process identifier (pid)

• Value of registers, including stack pointer
  ➢ Program counter
  ➢ CPU registers

• Information needed by scheduler
  ➢ Process state
  ➢ Other CPU scheduling information

• Resources held by process:
  ➢ Memory-management information
  ➢ I/O status information

• Accounting information
Context Switching

• Multiprogramming: switch to another process if current process is (momentarily) blocked

• Time-sharing: switch to another process periodically to make sure all processes make equal progress
  • this switch is called a context switch.

• Understand how it works
  ➢ how it interacts with user/kernel mode switching
  ➢ how it maintains the illusion of each process having the CPU to itself (process must not notice being switched in and out!)
CPU Switch From Process to Process

1) Save the current process’s execution state to its PCB
2) Update current’s PCB as needed
3) Choose next process N
4) Update N’s PCB as needed
5) Restore N’s PCB execution state
   May involve reprogramming MMU
Process Scheduling Queues

- Processes are linked in multiple 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** (wait queue) – set of processes waiting for an I/O device or other events
- Processes migrate among the various queues
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling
Schedulers

• **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - The long-term scheduler controls the *degree of multiprogramming*
  - A good mix of 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

• **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
CPU Scheduling

• Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

• CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running
Static vs Dynamic Scheduling

• Static scheduling
  ➢ The arrival and execution times of all jobs are known in advance. It creates a schedule, execute it
  – Used in statically configured systems, such as embedded real-time systems

• Dynamic/online scheduling
  • Jobs are not known in advance, scheduler must make online decision whenever jobs arrives or leaves
    – Execution time may or may not be known
    – Behavior can be modeled by making assumptions about nature of arrival process
Alternating Sequence of CPU And I/O Bursts
CPU Scheduling Model

Process alternates between CPU burst and I/O burst

I/O Bound Process

CPU Bound Process

Scheduling on the same CPU:

Waiting  CPU  I/O
CPU Scheduling Terminology

- A job (sometimes called a task, or a job instance)
  - Activity that’s schedulable: process/thread and a collection of processes that are scheduled together
- Arrival time: time when job arrives
- Start time: time when job actually starts
- Finish time: time when job is done
- Completion time (aka turn-around time)
  - Finish time – Arrival time
- Response time
  - Time when user sees response – Arrival time
- Execution time (aka cost): time a job need to execute
CPU Scheduling Terminology (cont’d)

• Waiting time = time when job was ready-to-run
  ➢ didn’t run because CPU scheduler picked another job
• Blocked time = time when job was blocked
  ➢ while I/O device is in use
• Completion time
  ➢ Execution time + Waiting time + Blocked time
CPU Scheduling Goals

• Minimize latency
  ➢ Can mean completion time
  ➢ Can mean response time

• Maximize throughput
  ➢ Throughput: number of finished jobs per time-unit
  ➢ Implies minimizing overhead (for context-switching, for scheduling algorithm itself)
  ➢ Requires efficient use of non-CPU resources

• Fairness
  ➢ Minimize variance in waiting time/completion time
Scheduling Constraints

Reaching those goals is difficult, because

- Goals are conflicting:
  - Latency vs. throughput
  - Fairness vs. low overhead

- Scheduler must operate with incomplete knowledge
  - Execution time may not be known
  - I/O device use may not be known

- Scheduler must make decision fast
  - Approximate best solution from huge solution space
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are \( n \) processes in the ready queue and the time quantum is \( q \), then each process gets \( 1/n \) of the CPU time in chunks of at most \( q \) time units at once. No process waits more than \( (n-1)q \) time units.

- No more unfairness to short jobs or starvation for long jobs!

- Performance
  - \( q \) large \( \Rightarrow \) FIFO
  - \( q \) small \( \Rightarrow \) \( q \) must be large with respect to context switch, otherwise overhead is too high
Round Robin – Cost of Time Slicing

• Context switching incurs a cost
  ➢ Direct cost (execute scheduler & context switch) + indirect cost (cache & TLB misses)
• Long time slices → lower overhead, but approaches FCFS if processes finish before timeslice expires
• Short time slices → lots of context switches, high overhead
• Typical cost: context switch < 10 us
• Time slice typical around 100 ms
  ➢ Linux: 100ms default, adjust to between 10ms & 300ms
  ➢ Note: time slice length != interval between timer interrupts
    ✓ Timer frequency usually 1000Hz
Multi-Level Feedback Queue Scheduling

- Objectives:
  - preference for I/O bound jobs (tends to lead to good I/O utilization)
  - longer timeslices for CPU bound jobs (reduces context-switching overhead)

- Challenge:
  - Don’t know type of each process – algorithm needs to figure out

- Solutions: use multiple queues
  - queue determines priority
  - usually combined with static priorities (nice values)
  - many variations of this
**MLFQS**

- Higher priority queues are served before lower-priority ones - within highest-priority queue, round-robin
- Only ready processes are in this queue - blocked processes leave queue and reenter same queue on unblock

Processes start in highest queue

Process that use up their time slice move down

Processes that starve move up
Scheduling Summary

- OS must schedule all resources in a system
  - CPU, Disk, Network, etc.
- CPU Scheduling affects indirectly scheduling of other devices
- Goals: (1) Minimize latency (2) Maximize throughput (3) Provide fairness
- In Practice: some theory, lots of tweaking
Single and Multithreaded Processes

- Single-threaded process
  - Code
  - Data
  - Files
  - Registers
  - Stack

- Multithreaded process
  - Code
  - Data
  - Files
  - Registers
  - Registers
  - Registers
  - Stack
  - Stack
  - Stack

thread

single-threaded process

multithreaded process
Benefits of Multithreading

• Responsiveness

• Resource Sharing

• Economy

• Utilization of MP Architectures
Thread Implementations

• User threads
  - Thread management done by user-level threads library
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

• Kernel threads:
  - Supported by the Kernel
  - Examples
    - Windows XP/2000
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X
An example: Pthread

```c
#include <stdio.h>
#include <stdlib.h> #include <pthread.h>

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we */
    /* wait we run the risk of executing an exit which will terminate */
    /* the process and all threads before the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

cc -lpthread pthread1.c
Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool