Review

- Hardware Support for OS
- What happens on
  – an interrupt / an exception/ a system call
- Operating System Architectures
  – monolithic
  – microkernel
  – virtual machine

Outline

- An OS in action
- Processes and Programs
- A day in the life of a process
- CPU scheduling

An Operating System in Action

- CPU loads boot program from ROM (e.g. BIOS in PC's)
- Boot program:
  - Examines/checks machine configuration (number of CPU's, how much memory, number & type of hardware devices, etc.)
  - Builds a configuration structure describing the hardware
  - Loads the operating system, and gives it the configuration structure
- Operating system initialization:
  - Initialize kernel data structures
  - Initialize the state of all hardware devices
  - Creates a number of processes to start operation (e.g. getty in UNIX, the Windowing system in NT, e.g.)

O.S. in Action (cont'd)

- After basic processes have started, the OS runs user programs, if available, otherwise enters the *idle loop*
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (notebooks)
  - OS computes some function (DEC's VMS on VAX computed Pi)
- OS wakes up on:
  - interrupts from hardware devices
  - traps from user programs
  - exceptions from user programs

The Process Abstraction

- A process is an abstraction that supports running programs
- Different processes may run several instances of the same program
- In most systems, processes form a tree, with the root being the first process to be created
- At a minimum, the following resources are required:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

Process Management

So, What is a Program?

A program consists of:

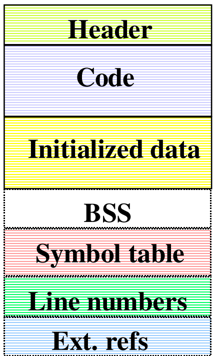
- code -- machine instructions
- data -- variables stored and manipulated in memory, classified into
  - initialized variables (globals)
  - dynamically allocated variables (malloc, new)
  - stack variables (C automatic variables, function arguments)
- DLL's -- libraries that were not compiled or linked with the program (containing code & data, possibly shared with other programs that use them)
- mapped files -- memory segments containing variables (mmap()), used frequently in database programs

Running a Program

- O.S. creates a "process" with some memory allocated for it
- The loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes "argc", "argv", "envp" on the stack
  - sets the CPU registers properly & calls "__start()" [Part of CRT0]
- Program start running at __start(), which calls main()

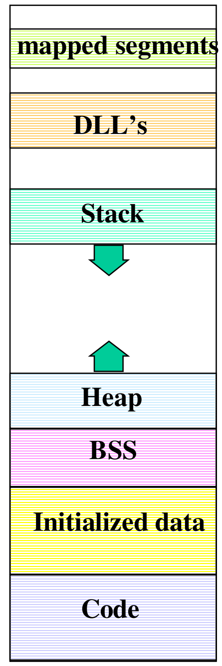
  we say "process" is now running, and no longer think of "program"
- When main() returns, CRT0 calls "exit()" which destroys the process and returns all resources

Anatomy of a Process



Executable File

Process's
address space



Example

```
int a;
int b = 9;
const char * s = "Hey dude";

main(int argc, char ** argv)
{
    char c;
    char * p = malloc(sizeof(char));
    static v;
    …
    return 1;
}
```

Identify the location of each variable both in the executable file and in the process space

Process Context

Definition:

The process context consists of its address space and the CPU registers

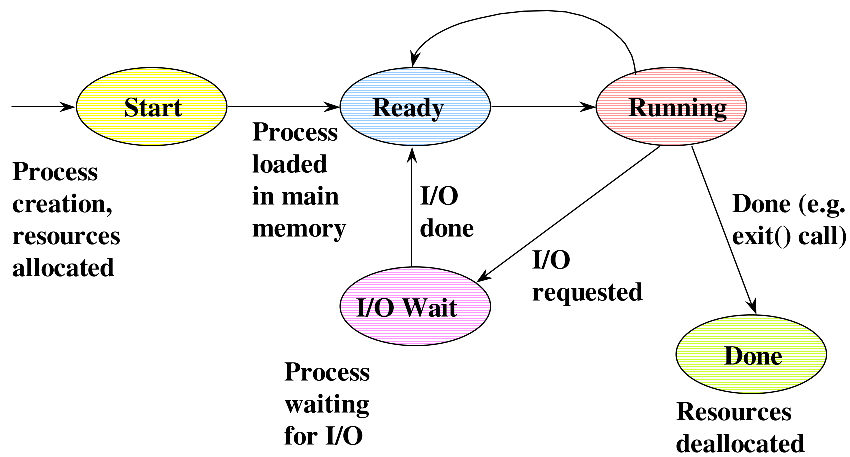
We say that we are running the context of process *p* if its address space is in memory and the CPU registers are used to run *p*

When a process *p* is stopped (e.g. waiting for I/O), then its context is not active

Process Control Block

process state
program counter
other CPU registers
memory limits
list of open files
⋮

The Process Life Cycle



Context Switching

- All programs use the CPU registers during execution
- The process implementing a program must have its own set of "private" CPU registers
  - stored in main memory
  - loaded into the CPU registers when process moves from "ready" to "running"
  - must be saved back to main memory when process moves from "running" to either "ready" or "I/O wait"
- A context switch occurs whenever an interrupt occurs, or when a process issues a system call

Penalties of Context Switching

- Explicit:
  - Cost of loading and storing registers from/into main memory
- Implicit:
  - In a pipelined CPU, must wait until pipeline is drained
  - If a CPU uses memory caches, the process that is switched in usually have a large number of cache misses when it runs until they are loaded from memory

Context switching overhead a factor in choosing scheduling policy

Context switching overhead is crucial for OS efficiency, and its cost continues to increase with faster CPU's

Scheduling

- Selecting which of the "ready" processes to run
- Must:
  - make users happy
  - use resources efficiently
  - be fair (what does it mean?)
  - …?

Metrics

- ⬆ CPU Utilization
  - time CPU is doing useful work/ total elapsed time
- ⬆ Throughput
  - number of completed processes/unit of time
- ⬇ Response Time
  - average(process completion time – process submission time)
- ⬇ Waiting time
  - time spent in ready queue

These goals are often contradictory

Given a set of processes, finding an optimal schedule is NP-complete

Scheduling Algorithms

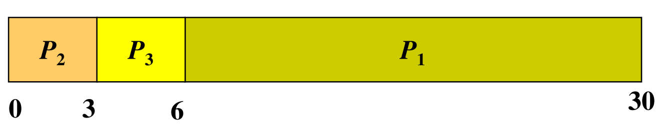
- Really cool thing to be into if you like Disco
- Traditional assumptions
  - one program/user
  - one thread/user
  - programs are independent
- What happens if you remove these assumptions?
- Can we schedule multiple resources together (*e.g.* CPU+ disk I/O)?

First Come First Served (FCFS)

- Scheduler implements a FIFO queue (ready queue or run queue)
- When a process becomes ready, it enters the queue
- Process at the head of queue is allowed to run *to completion* before bringing in the next process

Example

Response time: (24+27+30)/3 = 27



Response time: (3+6+30)/3 = 13

Pre-emptive vs. Non Pre-emptive

- In non pre-emptive scheduling, once a process starts it is allowed to run until it finishes
  - Simple and efficient to implement
  - Creates problems (what are they and how to solve them?)
- In pre-emptive scheduling, a process is switched back and forth between the “ready” and “running” states
  - More sophisticated with lots of capabilities
  - Less efficient (context switching)
  - Needs hardware support

Round Robin Scheduling (RR)

- Scheduler implements a FIFO queue
- Process at the head of the queue is allowed to run until:
  - a time quantum expires, in which case process re-enters queue
  - process enters the wait state, in which process is out of the queue and re-enters when the wait condition is no longer relevant

How do we choose the time quantum?

- What if too big?
  - bad response time (back to FCFS)
- What if too small?
  - poor throughput (spend all time context switching)
- In practice, between 10-100 ms
  - context switch time .1-1 ms (1% overhead)

FCFS vs RR

- Assume 0-cost context switch.
  Is RR always better that FCFS?

- 10 jobs
- each takes 100 seconds
- 1 second time quantum

	Job Completion Time	
Job #	FCFS	RR
1	100	991
2	200	992
3	300	993
…	…	…
9	900	999
10	1000	1000

STCF and SRTCF

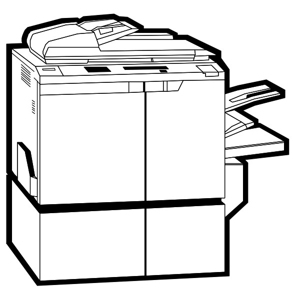
STCF: Shortest Time to Completion First

- Scheduler implements a queue sorted by amount of time to run the process
- Process at the head of queue is allowed to run *to completion* before bringing in the next process

SRTCF: Shortest Remaining Time to Completion First

- Same as STCF
- only, jobs can be preempted

The Idea



- Get short jobs out of the system quickly
- Big effect on short jobs, small effects on large jobs
- Get better average response time

In fact, these protocols are *optimal*!

The Catch

- Unfair
  - constant stream of short jobs can arbitrarily delay long jobs
- Clairvoyant
  - need to know the future!
  - easy: ask the user
  - yeah, right!

What if you don't subscribe to the Psychic Network?

Use past to predict future!

- e.g., if program was I/O bound in the past, likely to be I/O bound in the future
- used in virtual memory, file system, CPU scheduling…

t_n = length of last CPU burst

τ_{n+1} = predicted value of next CPU burst

$$\boxed{\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n}$$

Multi-Feedback Queues (UNIX)

- Scheduler implements several RR queues such that the processes in one queue all have the same priority
- Process at the head of the highest priority queue is allowed to run until:
  - a time quantum expires, in which case process re-enters queue
  - process enters the wait state, in which process is out of the queue and re-enters when the wait condition is no longer relevant
- After running for a while, a process is relegated to the next queue in priority order (its priority is decreased)
- After spending time in the I/O wait, a process is promoted to the highest priority queue (its priority is set to max)

Can we fool the protocol?