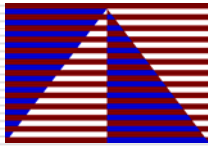


ECE3055 Computer Architecture and Operating Systems



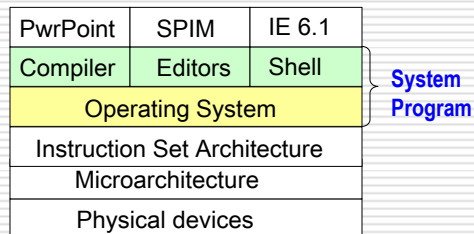
Lecture: Processes & Threads



Prof. Hsien-Hsin Sean Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

What is an Operating System?

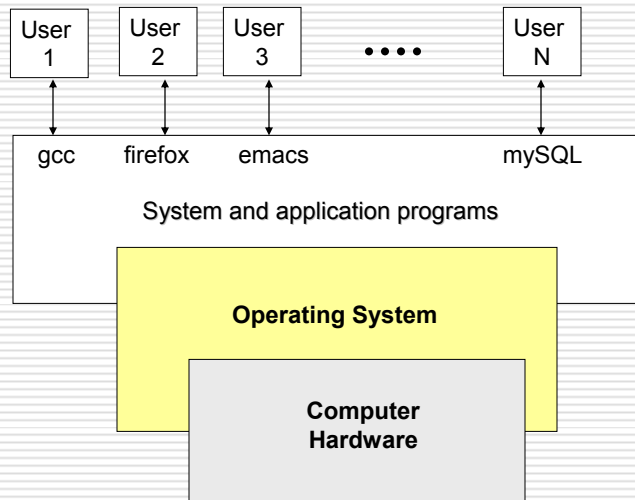
- An intermediate program between a user of a computer and the computer hardware (to hide messy details)
- Goals:
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient and efficient to use



Computer System Components

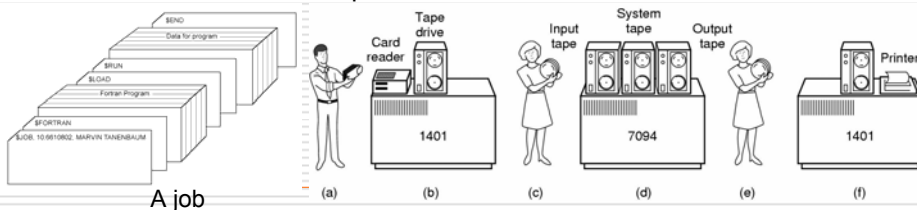
- Hardware
 - Provides basic computing resources (CPU, memory, I/O)
- Operating System
 - Controls and coordinates the use of the hardware among various application programs for various users
- Application Programs
 - Define the ways in which the system resources are used to solve the computing problems of users (e.g. database systems, 3D games, business applications)
- Users
 - People, machines, other computers

Abstract View of System Components



History of Operating Systems

- ❑ Vacuum Tubes and Plug boards (1945 – 55)
 - Programmers sign up a time on the signup sheet on the wall
- ❑ Transistors and batch system (1955 – 65)
 - Mainframes, operated by professional staff
 - Program with punch cards
 - Time wasted while operators walk around



Time-sharing Systems (Interactive Computing)

- ❑ The CPU is multiplexed among several jobs that are kept in memory and on disk (The CPU is allocated to a job only if the job is in memory)
- ❑ A job swapped in and out of memory to the disk
- ❑ On-line communication between the user and the system is provided
 - When the OS finishes the execution of one command, it seeks the next “control statement” from the user’s keyboard
- ❑ On-line system must be available for users to access data and code
- ❑ MIT MULTICS (MULTiplexed Information and Computing Services)
 - Ken Thompson went to Bell Labs and wrote one for a PDP-7
 - Brian Kernighan jokingly dubbed it UNICS (UNIplexed ..)
 - Later spelled to UNIX and moved to PDP-11/20
 - IEEE POSIX to standardize UNIX

Operating System Concepts

- Process Management
 - Main Memory Management
 - File Management
 - I/O System Management
 - Secondary Management
 - Networking
 - Protection System
 - Command-Interpreter System
-

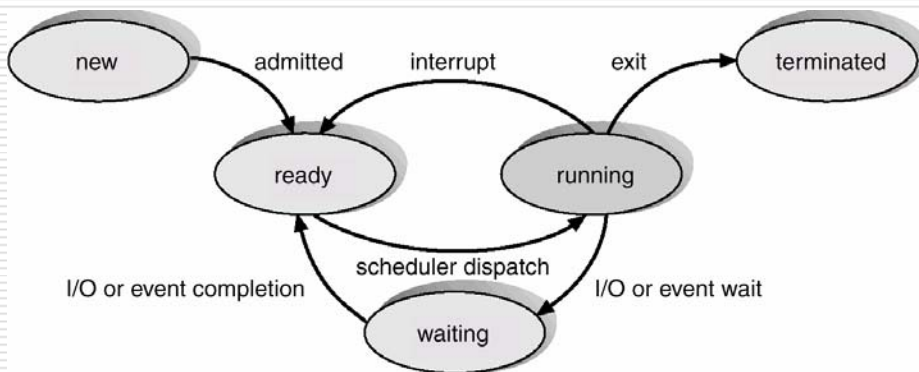
Process Management

- A *process* is a program in execution
 - A process contains
 - Address space (e.g. read-only code, global data, heap, stack, etc)
 - PC, \$sp
 - Opened file handles
 - A process needs certain resources, including CPU time, memory, files, and I/O devices
 - The OS is responsible for the following activities for process management
 - Process creation and deletion
 - Process suspension and resumption
 - Provision of mechanisms for:
 - process synchronization
 - process communication
-

Process State

- As a process executes, it changes *state*
 - new: The process is being created
 - ready: The process is waiting to be assigned to a processor
 - running: Instructions are being executed
 - waiting: The process is waiting for some event (e.g. I/O) to occur
 - terminated: The process has finished execution

Diagram of Process State



Process Control Block (PCB)

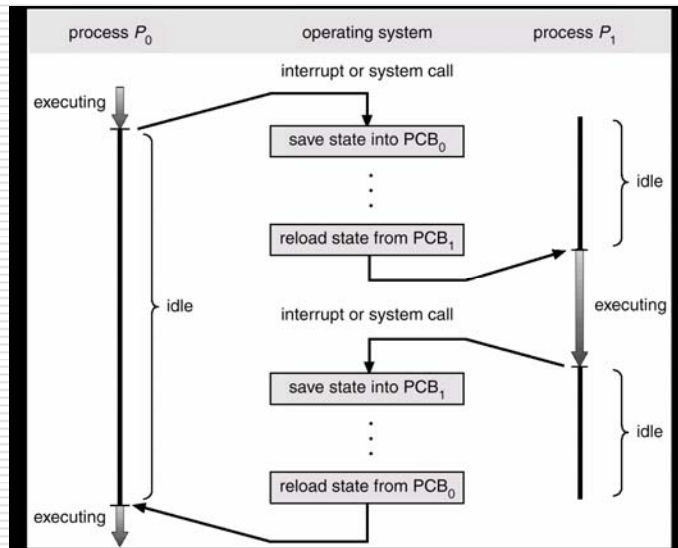
Information associated with each process

- ❑ Process state
- ❑ Program counter
- ❑ CPU registers (for context switch)
- ❑ CPU scheduling information (e.g. priority)
- ❑ Memory-management information (e.g. page table, segment table)
- ❑ Accounting information (PID, user time, constraint)
- ❑ I/O status information (list of I/O devices allocated, list of open files etc.)

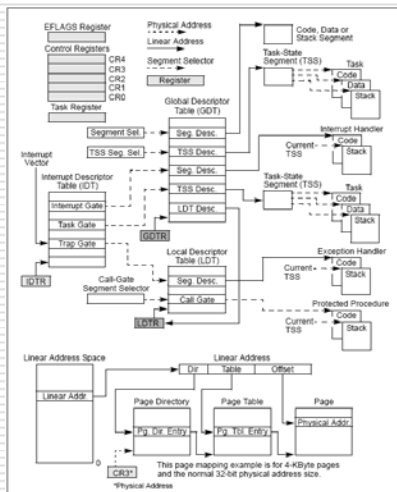
Process Control Block

| |
|--------------------|
| process state |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| ... |

CPU Switch From Process to Process



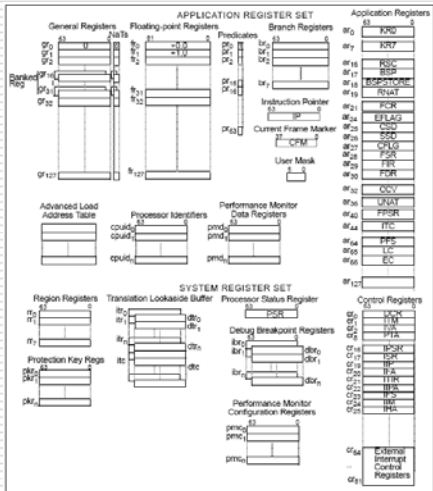
Example of System State: Intel Core Duo



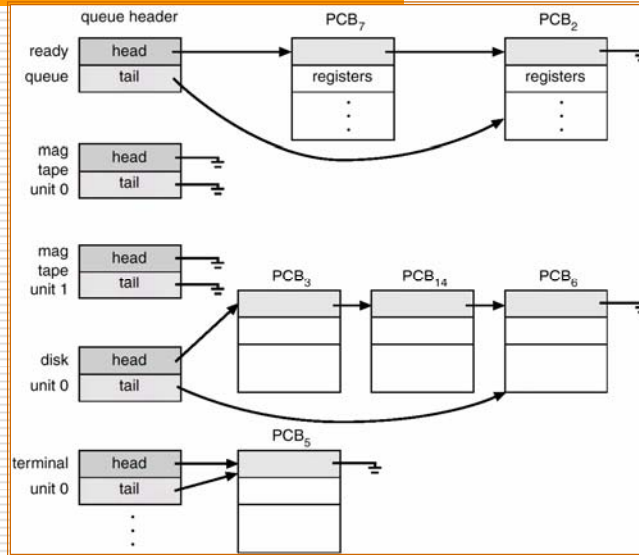
- Note the registers that are required to record the “state” of the executing process
- State is swapped on a context switch

Source: From Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1

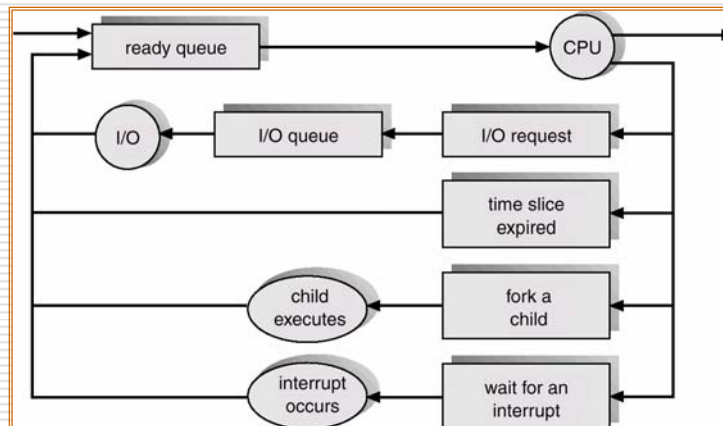
Example of System State: Intel Itanium



Ready Queue And Various I/O Device Queues



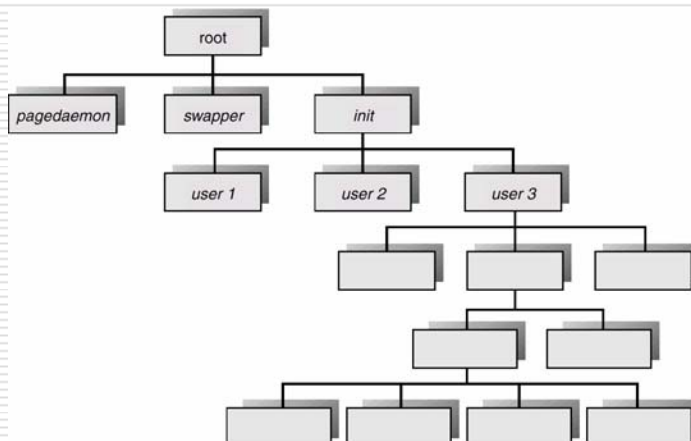
Representation of Process Scheduling



Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate

Processes Tree on a UNIX System



Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program

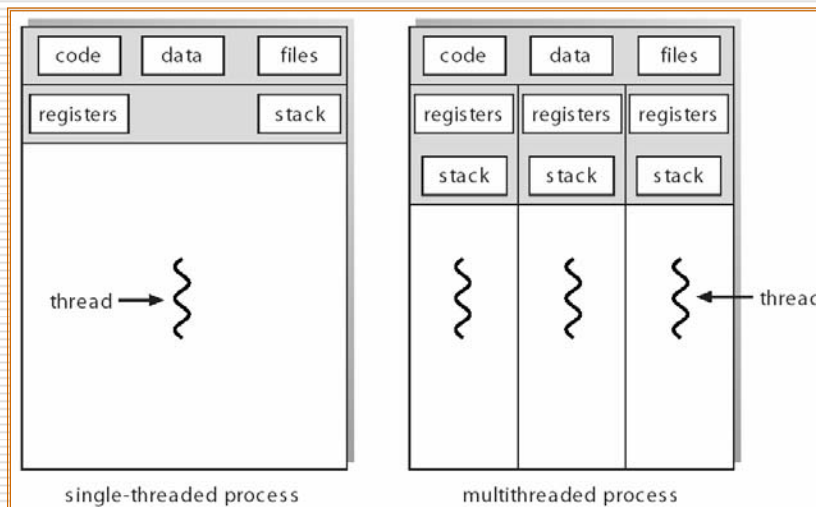
C Program Forking Separate Process

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child
process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the
child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - Some operating system do not allow child to continue if its parent terminates
 - All children terminated - *cascading termination*

Single and Multithreaded Processes



User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
 - POSIX Pthreads
 - Java threads
 - Win32 threads

Kernel Threads

- Supported by the Kernel

- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

Pthreads

- ❑ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ❑ API specifies behavior of the thread library, implementation is up to development of the library
- ❑ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Example

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* now wait for the thread to exit */
    pthread_join(tid,NULL);
    printf("sum = %d\n",sum);
}

void *runner(void *param) {
    int upper = atoi(param);
    int i;
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

Examples of Threads

- A web browser
 - One thread displays images
 - One thread retrieves data from network
- A word processor
 - One thread displays graphics
 - One thread reads keystrokes
 - One thread performs spell checking in the background
- A web server
 - One thread accepts requests
 - When a request comes in, separate thread is created to service
 - Many threads to support thousands of client requests
- RPC or RMI (Java)
 - One thread receives message
 - Message service uses another thread

29

Threads vs. Processes

Thread

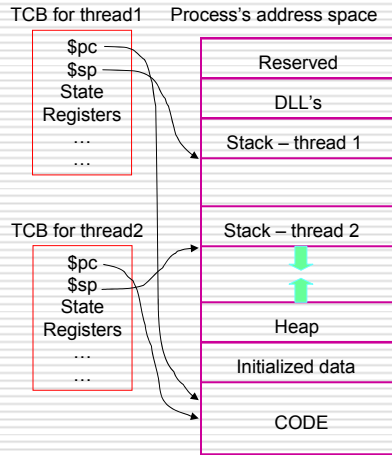
Processes

- | | |
|---|--|
| <ul style="list-style-type: none"> □ A thread has no data segment or heap □ A thread cannot live on its own, it must live within a process □ There can be more than one thread in a process, the first thread calls main and has the process's stack □ Inexpensive creation □ Inexpensive context switching □ If a thread dies, its stack is reclaimed by the process | <ul style="list-style-type: none"> □ A process has code/data/heap and other segments □ There must be at least one thread in a process □ Threads within a process share code/data/heap, share I/O, but each has its own stack and registers □ Expense creation □ Expensive context switching □ If a process dies, its resources are reclaimed and all threads die |
|---|--|

30

Thread Implementation

- Process defines address space
- Threads share address space
- Process Control Block (PCB) contains process-specific info
 - PID, owner, heap pointer, active threads and pointers to thread info
- Thread Control Block (TCB) contains thread-specific info
 - Stack pointer, PC, thread state, register ...



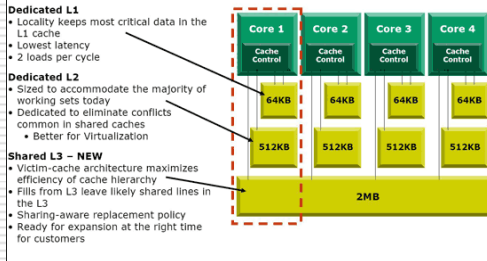
Benefits

- Responsiveness
 - When one thread is blocked, your browser still responds
 - E.g. download images while allowing your interaction
- Resource Sharing
 - Share the same address space
 - Reduce overhead (e.g. memory)
- Economy
 - Creating a new process costs memory and resources
 - E.g. in Solaris, 30 times slower in creating process than thread
- Utilization of MP Architectures
 - Threads can be executed in parallel on multiple processors
 - Increase concurrency and throughput

Why Threads?

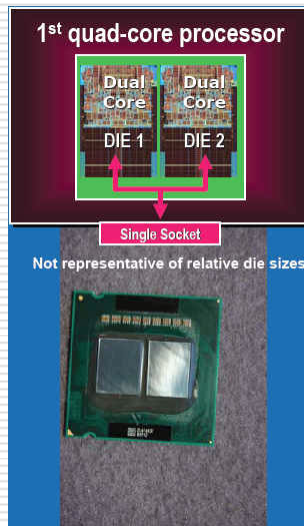
Balanced, Highly Efficient Cache Structure

AMD Barcelona



- ❑ Concurrency!
- ❑ Advantageous in a single core environment
 - Overlap I/O and computation → keep the core busy
- ❑ Natural in a multicore environment
 - Shared memory parallel programming
 - Parallel execution
 - ❑ Synchronization to enforce correctness

From Intel



Multicore is Here to Stay

