Thread and Synchronization

Task Model
(Module 18)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014

Why Talk About This Subject

- **A thread of program execution**
  - How a program start and end its execution
  - waiting for an event or a resource, delay a period, etc.
- **For concurrent operations → multiple threads of program execution**
- **How can we make this happen?**
  - support for program execution
  - sharing of resources
  - scheduling
  - communication between threads

Thread and Process

- **Process:**
  - an entity to which system resources (CPU time, memory, etc.) are allocated
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads
- **Thread:**
  - a sequence of control within a process and shares the resources in that process
- **Lightweight process (LWP):**
  - LWP may share resources: address space, open files, …
  - clone or fork – share or not share address space, file descriptor, etc.
  - In Linux kernel, threads are implemented as standard processes (LWP) that shares certain resources with other processes, and there is no special scheduling semantics or data structures to represent threads

Why Threads

- **Advantages:**
  - the overhead for creating a thread is significantly less than that for creating a process
  - multitasking, i.e., one process serves multiple clients
  - switching between threads requires the OS to do much less work than switching between processes
- **Drawbacks:**
  - not as widely available as the process features
  - writing multithreaded programs require more careful thought
  - more difficult to debug than single threaded programs
  - for single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many CPU cycles to be had)

POSIX Thread (pthread)

- **IEEE's POSIX Threads Model:**
  - programming models for threads in a UNIX platform
  - pthreads are included in the international standards
- **pthreads programming model:**
  - creation of threads
  - managing thread execution
  - managing the shared resources of the process
- **main thread:**
  - initial thread created when main() is invoked
  - has the ability to create daughter threads
  - if the main thread returns, the process terminates even if there are running threads in that process
  - to explicitly avoid terminating the entire process, use pthread_exit()

Linux task_struct

```
struct task_struct {                        /* Linux/include/linux/sched.h  */
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;    /* per process flags, defined below */
        unsigned int ptrace;

        int lock_depth;                     /* BKL (big kernel lock) lock depth */

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        ……..
        struct mm_struct *mm, *active_mm;
        struct thread_struct thread;      /* CPU-specific state of this task */
        struct fs_struct *fs;   /* filesystem information */
        struct files_struct *files;         /* open file information */
```

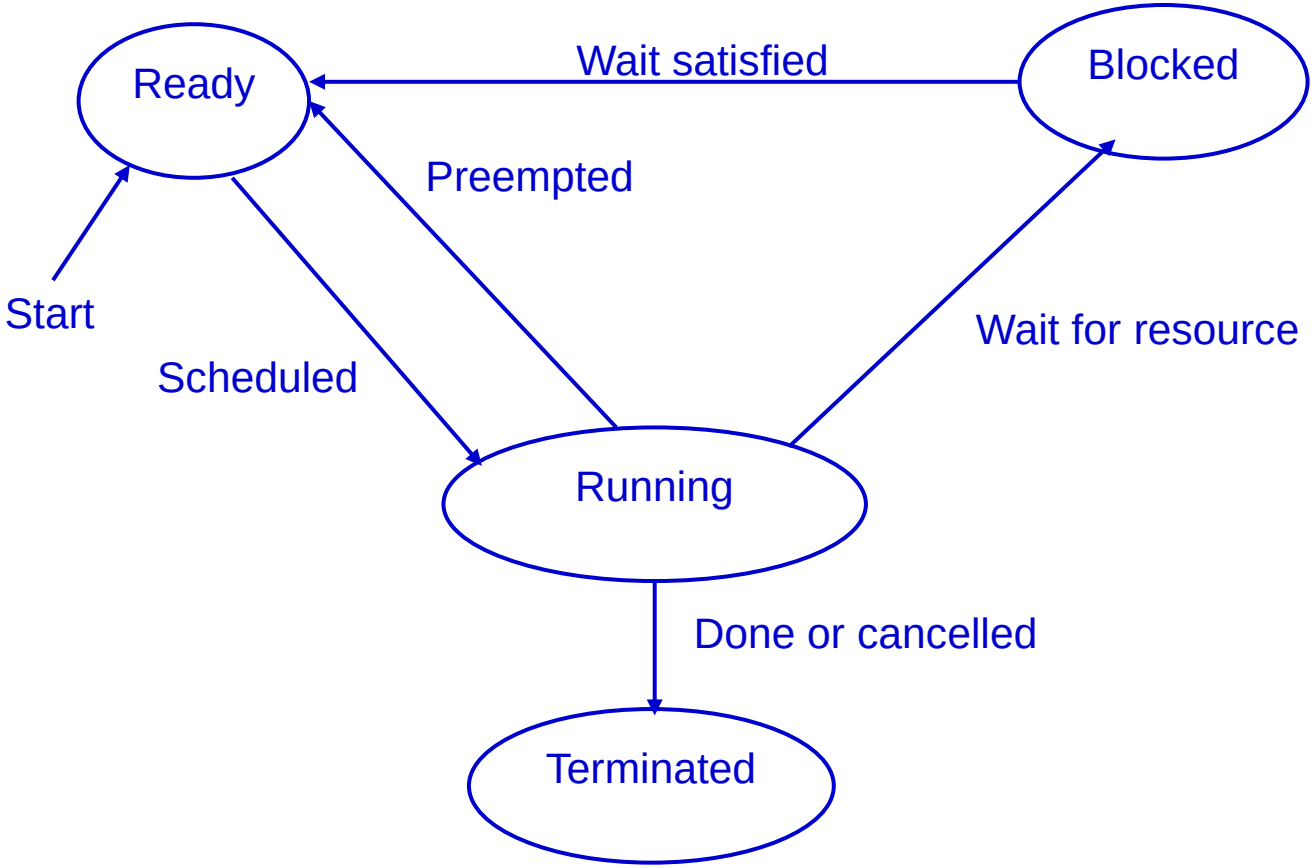
Process -- *task_struct* data structure

- **state: process state**
  - TASK_RUNNING: executing
  - TASK_INTERRUPTABLE: suspended (sleeping)
  - TASK_UNINTERRUPTABLE: (no process of signals)
  - TASK_STOPPED (stopped by SIGSTOP)
  - TASK_TRACED (being monitored by other processes such as debuggers)
  - EXIT_ZOMBIE (terminated before waiting for parent)
  - EXIT_DEAD
- **thread_info: low-level information for the process**
- **mm: pointers to memory area descriptors**
- **tty: tty associated with the process**
- **fs: current directory**
- **files: pointers to file descriptors**
- **signal: signals received ……**

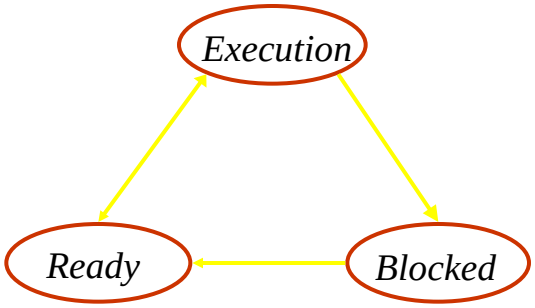
Linux Processor State

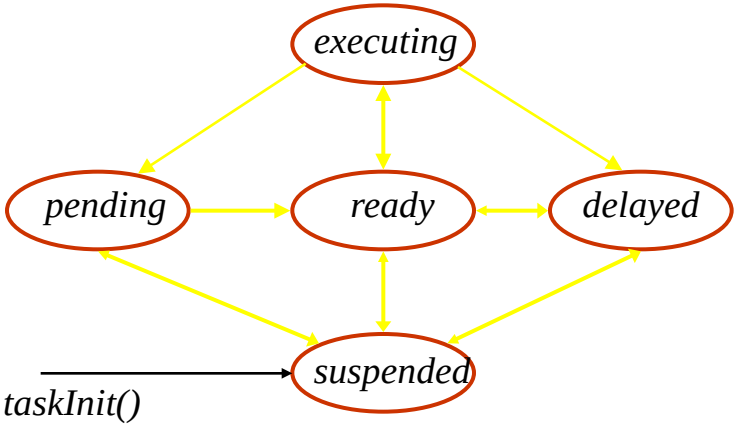
```
/* This is the TSS  (task State Segment) defined by the hardware and saved in stack. */
struct x86_hw_tss {
        unsigned short          back_link, __blh;
        unsigned long           sp0;
        unsigned short          ss0, __ss0h;
        unsigned long           sp1;
        unsigned short          ss1, __ss1h; /* ss1 caches MSR_IA32_SYSENTER_CS: */
        unsigned long           sp2;
        unsigned short          ss2, __ss2h;
        unsigned long           __cr3;
        unsigned long           ip;
        unsigned long           flags;
        unsigned long           ax;
        unsigned long           cx;
        unsigned long           dx;
        unsigned long           bx;
/*  For ARM, Linux/arch/arm/include/asm/thread_info.h.,
```

Linux Thread State Transition



Task Management in vxWorks





- **Task structure in task control block –**
  - priority(initial and inherited), stack frame, task current state,
  - entry point, processor states (program counter, registers)
  - callback function (hook) pointers for OS events
  - spare variables

VxWorks Task States

```
typedef struct windTcb                  /* WIND_TCB - task control block */
    {
        char *          name;       /* 0x34: pointer to task name */
        UINT            status;             /* 0x3c: status of task */
        UINT            priority;           /* 0x40: task's current priority */
        UINT            priNormal;          /* 0x44: task's normal priority */
        UINT            priMutexCnt;        /* 0x48: nested priority mutex owned */
        UINT            lockCnt;            /* 0x50: preemption lock count */
        FUNCPTR         entry;              /* 0x74: entry point of task */
        char *          pStackBase;/* 0x78: points to bottom of stack */
        char *          pStackLimit;/* 0x7c: points to stack limit */
        char *          pStackEnd;  /* 0x80: points to init stack limit */
#if (CPU_FAMILY==I80X86) /* function declarations */
        EXC_INFO        excInfo;            /* 0x118: exception info */
        REG_SET         regs;               /* 0x12c: register set */
        DBG_INFO_NEW    dbgInfo0;   /* 0x154: debug info */
#endif /* CPU_FAMILY==I80X86 */
```