Agenda

- **Last time (Sept-14 7:30pm-8:45 / Sept-15 3:00-4:15pm)**
  - Processes/Threads (chpt 4 + 5)
- **This time (Tues Sept 19)**
  - Threads (chpt 5)
  - CPU scheduling (chpt 6)
- **Next time (Thurs Sept 21)**
  - CPU sched (chpt 6)
  - Intro Synchronization (chpt 7)

- **Extra class meeting this week (Thurs/Friday)!**

- **PA#2 due Thurs Sept 21 11am**

Before we start

- **Windows executive can sometimes service the request without going to the particular subsystem (e.g., Windows "ReadFile()")**
- **TA office hours (all in 002a):**
  - Wittawat: Wed 6-8pm
  - Dhruv: Thurs 7-9pm
  - Ray: Mon 5-7pm 002a

Before we start: PA#2

- **An critical skill: PATTERN MATCHING!**
- **What should happen when I execute "unhide" twice in a row?**
- ***NOTE: try "ebp-8" as necessary***
- **"GetProcAddress" of NtGetCurrentProcessorNumber in ntdll.dll ???? What's up with that????**

- **Part-2: Q2: step 2.2.6: the output could be "nothing".**

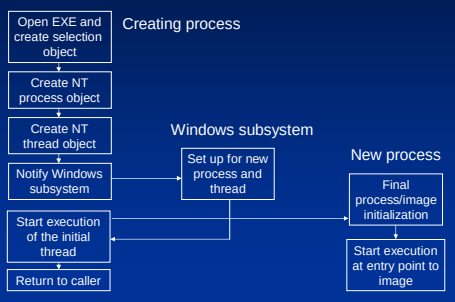
From last time: More Complicated

```
int main() {
  char prgname[1024];
  pid_t pid;
  int status;

  if (fgets(prgname, 1024, stdin) == NULL) {
    printf ("Did not read program name. Aborting!\n");
    exit(1);
  }
  prgname[strlen(prgname)-1]=0;

  if ((pid = fork ()) == -1) {
    perror("fork");
    exit(1);
  }
  if (pid == 0) {
    execlp(prgname,prgname,0);
    printf("%d I did not find program %s\n", getpid(), prgname);
    exit (1);
  } else {
    waitpid(pid, &status, 0);
    if (status == 0) printf ("Program %s finished!\n", prgname);
  }
  return 0;
}
```

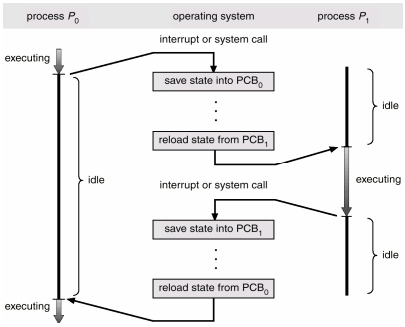

The main Stages Windows follows to create a process



Context Switch

- To run a process, the OS loads the values of the hardware registers (PC, SP, other registers) from the values stored in that process' PCB
- As the program executes, the CPU registers changes values (PC, SP)
- 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
- Context-switching is overhead; the system does no *useful work* while switching
- The duration of a context switch is dependent on hardware
- Typically, time sharing OS performs 100 to 1000 context switches per second
- Picture on next slide….

Context Switch

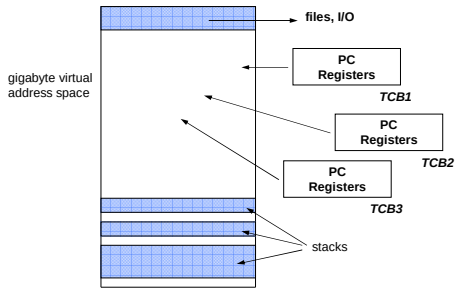


Threads

- **"Original" Process: address space and (single) flow of execution**
- **Thread**
  - we must separate address space (*process* or *task*) and flow of execution (*thread* or *lightweight process, LWP)*
- **Motivation:**
  - Context switch between cooperating processes is HUGE (reestablishing address space); context switch between cooperating threads is cheap
  - **fork(…)** of cooperating process is expensive; spawn of thread is cheap
  - programming is easier(?)

traditional UNIX | **embedded systems** | **Windows, Solaris (POSIX)**

Threads and Address Space



Kernel Threads

- **An improvement over only processes**
- **Creation/Switching still requires trapping to the kernel (system call)**
- **Thread data structure resides within the kernel:**
  - *Thread Control Block (TCB)* (execution state and scheduling info), so more complexity for the kernel
- **Only one scheduling policy per system**
- **OS (still) does not trust the user, so there must be a lot of checking on kernel calls**

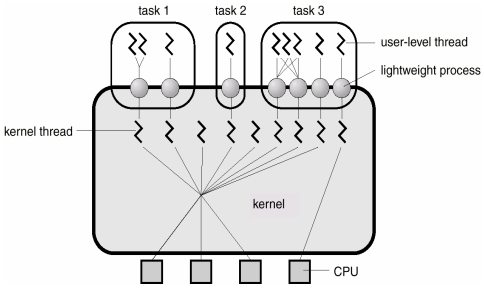
User-level Threads

- **Faster than kernel-level threads**
  - In what sense?
- **Managed by run-time system in user-space (no kernel calls)**
- **Creation, switching, and synchronizing between thread calls can be done without kernel involvement**
- **Process-specific scheduling policies are possible**
- **Problem: whole process blocks when one thread blocks (there are ways around this, but they're complex)**
- **OS can make poor scheduling choices, because OS has no notion of "amount of work" each process must do**

"Hybrid": Solaris 2

- **Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.**
- **LWP – intermediate level between user-level threads and kernel-level threads.**
- **Resource needs of thread types:**
  - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
  - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
  - User-level thread: only ned stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

Solaris 2 Threads



Creation of a Thread

1. The thread count in the process object is incremented.
2. An executive thread block (ETHREAD) is created and initialized.
3. A thread ID is generated for the new thread.
4. The TEB is set up in the user-mode address space of the process.
5. The user-mode thread start address is stored in the ETHREAD.

Creation of a Thread

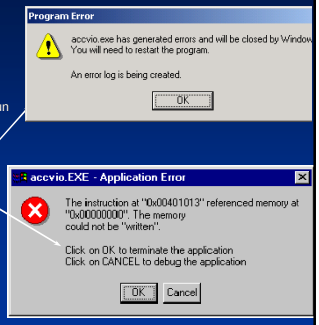
6. KeInitThread is called to set up the KTHREAD block.
   - The thread's initial and current base priorities are set to the process's base priority, and its affinity and quantum are set to that of the process.
   - KeInitThread allocates a kernel stack for the thread and initializes the machine-dependent hardware context for the thread, including the context, trap, and exception frames.
   - The thread's context is set up so that the thread will start in kernel mode in KiThreadStartup.
   - Finally, KeInitThread sets the thread's state to Initialized and returns to PspCreateThread.
7. Any registered systemwide thread creation notification routines are called.
8. The thread's access token is set to point to the process access token,
   - an access check is made to determine whether the caller has the right to create the thread.
9. Finally, the thread is readied for execution.

Linux Pthreads: Kernel or User-Level – How to tell?

- **Run a pthread program, try to infer behavior**
  - PRO: No need to inspect kernel code
  - CON: What behavior are we looking for? (what's the hypothesis?) – what's the test case?
- **Google for the answer**
  - PRO: Where exactly to find the answer?
    - http://www.tldp.org/FAQ/Threads-FAQ/OSsCompared.html
  - CON: Could be wrong; might learn more by "doing it yourself"
- **Linker line has "-lpthread", so it must be user-level**
  - Not necessarily – this could be just for the pthreads_api, not the "entire pthreads implementation"
- **Evidence: hmmm….**
  - "grep -i thread uml/linux-2.4.26/kernel/* | wc –l" returns 98
  - "grep -i thread uml/linux-2.4.26/arch/i386/kernel/entry.S" returns essentially nothing…

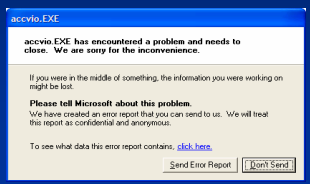
Process Crashes (Windows 2000)

- Registry defines behavior for unhandled exceptions
  - HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
  - Debugger=filespec of debugger to run on app crash
  - Auto 1=run debugger immediately 0=ask user first
- Default on retail system is
  Auto=1; Debugger=DRWTSN32.EXE
- Default with VC++ is
  Auto=0, Debugger=MSDEV.EXE



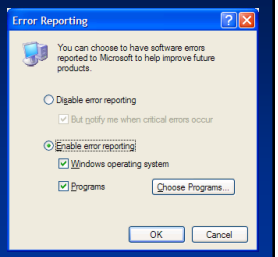
Process Crashes (Windows XP & Windows Server 2003)

- On XP & Server 2003, when an unhandled exception occurs:
  - System first runs DWWIN.EXE
    - DWWIN creates a process microdump and XML file and offers the option to send the error report
  - Then runs debugger (default is Drwtsn32.exe)

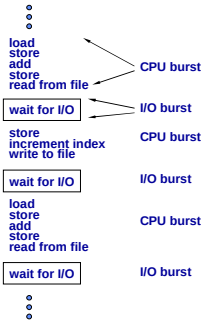


Windows Error Reporting

- Configurable with System Properties->Advanced->Error Reporting
  - HKLM\SOFTWARE\Microsoft\PCHealth\ErrorReporting
- Configurable with group policies
  - HKLM\SOFTWARE\Policies\Microsoft\PCHealth



Motivation for Scheduling: CPU and I/O Bursts



- **I/O-bound process: many short CPU bursts**
- **CPU-bound process: very long CPU bursts**
- ***What should happen to the CPU when a process performs I/O?***
- ***What should happen if a process never/rarely performs I/O?***

Scheduling

- **Multiprogramming:**
  - running more than one process at a time enables the OS to increase system utilization and throughout by overlapping I/O and CPU activities
- **[Policy vs. Mechanism]**
  - what to do vs. how to do it
  - separation is crucial! WHY?
- **[Process Execution State] remember from a few classes ago**
  - new, ready, running, waiting, terminated
- **[Long-Term Scheduling]**
  - How does the OS determine the degree of multiprogramming (the number of jobs executing at once in primary memory)?
  - Invoked very infrequently (seconds, minutes → may be slow)
- **[Short-Term Scheduling]**
  - How does the OS select a process from the ready Q to execute?
  - Invoked very frequently (milliseconds →must be fast)

Context Switches: Voluntary vs. Involuntary

- **Voluntary**
  - context switcher is invoked by the process that intends to relinquish the CPU **(nonpreemptive** scheduling); "yield" is for cooperative model
  - I/O requests and termination can also invoke context switcher
- **Involuntary**
  - Interrupt handler executes context switcher (preemptive scheduling)
  - Interval timer is used

Scheduling Criteria

- **user-oriented, performance-related**
  - [Response Time] (can be referred to as "wait time") time from submission of job to start of execution of job
  - [Turnaround Time] time from submission of job to completion of job
  - [Deadlines] time at which computation must complete
- **user-oriented, other**
  - [Predictability] job should run about the same amount of time regardless of load
- **system-oriented, performance-related**
  - [Throughout] jobs completed per unit time
  - [Processor Utilization] percentage of time that processor is busy
- **system-oriented, other**
  - [Fairness] jobs should be treated the same
  - [Balancing Resources] keep all resources busy

Algorithm Evaluation

- **[deterministic modeling]**
  - take a particular predefined workload and evaluation algorithm(s) (like gantt charts); gives exact numbers -- easy to compare; limited applicability (too specific); {this is what we'll focus on}
- **[queuing models]**
  - use *distribution* of characteristics of arrivals (arrival times and execution times); math can sometimes be difficult; too many independent assumptions
- **[simulation]**
  - programming a model of the computer system; random-number generators; can be expensive to develop
- **[implementation]**
  - actual implement it and evaluate it in "real operation"; difficulties: cost, what if environment changes?

Nonpreemptive Scheduling Policies

Important: Assume jobs are independent unless otherwise stated

- **First-in First-Out (FIFO): execute jobs to completion in the order of their arrival**
- **Example: assume context switch time of 0 seconds; if 2 jobs arrive as the same time, then job with "lower number" arrived first**

Job	*Arrival*	*Duration*	*Response*	*Turnaround*
1	0	50		
2	0	40		
3	10	30		
4	10	20		
5	20	10		

Advantage: **simple**

Disadvantages: **short jobs may wait behind long jobs; may lead to poor overlap of I/O and CPU; not appropriate for timesharing situations**

What does the Gantt chart look like?