Slide 1

Week 3

COMP3231 Operating Systems

2005/S2

➜ Threads and Processes
- What is a process? What is a thread?
- OS services to support processes, threads
- Thread switching
- Kernel level versus user level threads
- Threads in SVR4 Unix, Linux and Windows 2000

➜ Concurrency control
- Mutual exclusion (software, hardware, OS)

Slide 2

Threads

Do threads need OS support or is it possible to implement them as user-level library?

We need four features to implement multi-threading:

① Context switching
② Preemption
③ Scheduling
④ Handling of blocking system calls

Slide 3

Context Switching

➜ User-level operation `sigsetjmp` saves context and the set of blocked signals

➜ `siglongjmp` restores the environment (and jumps to the instruction the saved pc is pointing to)

Slide 4

Preemption and Scheduling

➜ System call `signal` allows the user to install alternative signal handler for some signals (timer)

➜ `alarm` sets the timer signal.

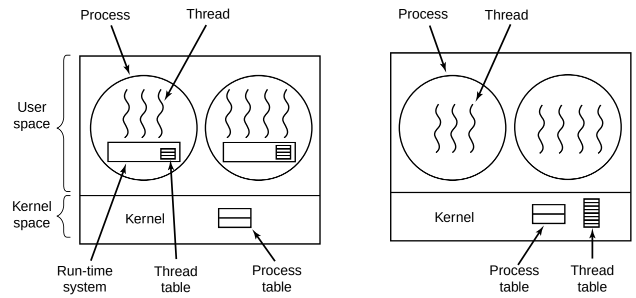
➜ Can be used to implement preemption and scheduling: timer signal activates scheduler, which picks next thread, sets alarm clock, then activates thread.

Slide 5

Blocking System Calls

- ➜ wrapper around each potentially blocking system call
- ➜ introduces fairly high overhead
- ➜ page faults can still lead to blocking of whole process

Slide 7



Slide 6

User-level Threads

- ➜ All thread management is done by the application (library)
- ➜ Runs in usermode
- ➜ The kernel is not aware of the existence of threads
  - User-level threads are not scheduled by the kernel
  - Pure application/library level construct
- ➜ Used to enhance modularisation
- ➜ Also called co-routines

Slide 8

User-level Threads vs Kernel Level Threads

- ✔ Scheduling policy tailored to specific application
- ✔ Extremely low overhead
- ✘ Process blocks if one of its threads blocks
  - for system calls, it can be avoided by using wrapper functions for all possibly blocking operations, introduces extra system calls
  - process blocks on page fault
- ✘ No inter-process parallelism possible on machines with multiple CPUs

Slide 9

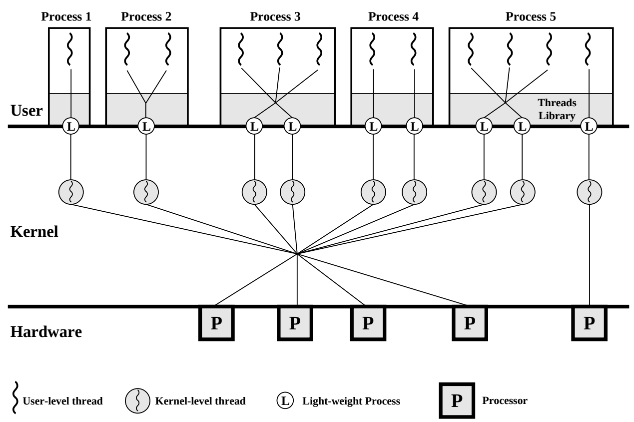
Combined Approaches

Try to get best of both worlds!

- ➜ Library offers (user-level) thread interface
- ➜ OS supports kernel-level threads
- ➜ Thread library provides API for binding one or more user-level threads to a kernel-level thread
- ➜ Most thread management done explicitly at user level

Slide 10

Example: Solaris Thread Architecture



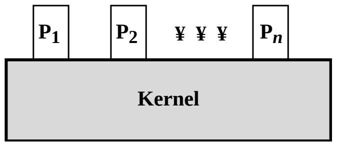
`

Slide 11

Execution Context of the Operating System

Non-process Kernel:

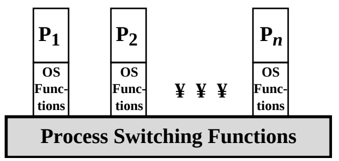
- ➜ Execute kernel outside of any process context
- ➜ Separate context for execution of OS code
  - OS context may be automatically switched by hardware
- ➜ Traditional model



Slide 12

Execution Within User Processes:

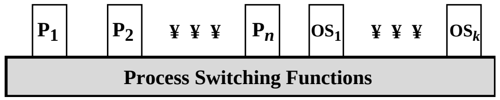
- ➜ OS software executes within context of a user process
- ➜ Process in privileged mode while executing OS code
  - has access to additional (kernel) memory
- ➜ E.g: UNIX



Slide 13

Execution Within Separate Process(es):

- ➜ Process-based (server-based) OS
- ➜ Separate process for major OS functions
- ➜ Clients use message-passing IPC to invoke services
  - Aids distribution
- ➜ OS processes may execute in unprivileged (user) mode
- ➜ E.g: Windows-2000



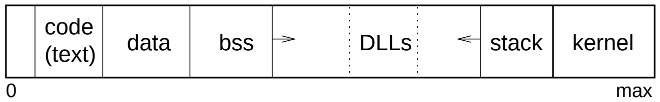
Slide 14

UNIX SVR4 Process Management

- ➜ Mostly follows in-process model
- ➜ In addition has "kernel processes" (daemons)
- ➜ Several parts of kernel data:
  - user-level context (text, data, stack)
  - register context
  - system-level context
    - process table entry:
      - ∗ global table, always entirely accessible by kernel
      - ∗ process state, IDs, prio, links to other data
    - "U area": process resources

Slide 15

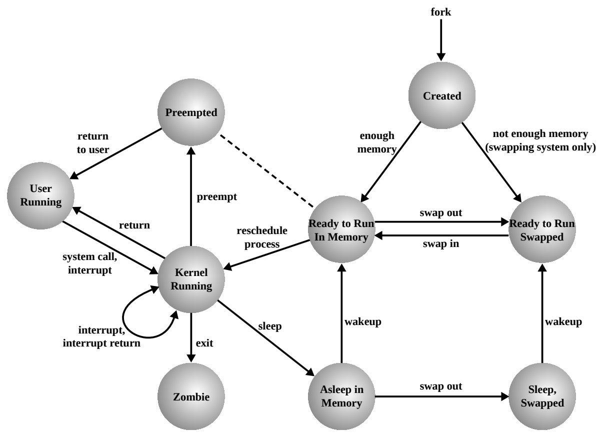
Typical Address Space Layout (UNIX):



- ➜ 0-th page typically not used
- ➜ text segment is read-only
- ➜ data segment is initialised data (part R/O)
- ➜ bss segment is uninitialised data (heap), can grow
- ➜ shared libraries (DLLs) allocated in free middle region
- ➜ stack at top of user space, grows downward
- ➜ kernel space is in reserved (shared) region

Slide 16

UNIX process states (single-threaded):



Slide 17

Main process system calls:

`fork()`	Creates process with copy of parent's process image
`exec()`	Replaces image of calling process with new executable file
`exit()`	Terminate calling process. Return exit status to parent
`wait()`	Wait for (specific or any) child to terminate. Collect exit status
`kill()`	Sends signal to process. Default action for many signals is to kill recipient, but may install signal handler

Slide 18

Typical use:

```
#include <stdio.h>
int pid;
pid = fork();
 if (pid < 0) {
   perror("fork() failed");
   exit(1);
}  else  if (pid > 0)            /* We're the parent! */
    printf("child PID=%d\n", pid);
}  else {                         /* We're the child! */
   execve(file);
   sprintf(STDERR, "Exec failed!");
   exit(1);
}
```


Slide 19

Threads in Linux:

- Linux has no real threads (with reduced context)
- Provides `clone()` system call
  - generalisation of Unix `fork()`
  - creates new process
  - parent and child can share (part of) address space
  - effectively a shortcut for `fork(); mmap()`
  - child has complete process context
  - `LinuxThreads` lib implemented using `clone`
- PThreads package provides user-level threads
  - can be bound to Linux "threads"
  - gives some approximation of lightweight threads
  - similar to Solaris
    - but Solaris' "lightweight processes" are faster

Slide 20

LINUX `clone` SYSTEM CALL

- `CLONE_PARENT`: (Linux 2.4 onwards) parent of new task same as parent of caller
- `CLONE_FS`: share file system information
- `CLONE_FILES`: share file descriptor table
- `CLONE_SIGHAND`: share table of signal handler
- `CLONE_VFORK`: parent suspended until child releases vm resources (`exit()`, `execve()`)
- `CLONE_VM`: parent and child run in the same memory space
- `CLONE_THREAD`: (Linux 2.4 onwards) parent and child share the same thread id

The Thread and Process Model in Windows 2000

Slide 21

Four important concepts in Windows 2000:

➜ Jobs
- collection of processes bundled together
- share quotas: max. number of processes, CPU time, memory usage, security restrictions

➜ Processes
- unit of resource ownership

➜ Threads
- units visible to scheduler

➜ Fibres
- user-level threads
- created using Win32 API calls, which do not trigger a system call

Slide 22