운영체제의 기초: Processes and Threads

2023년 3월 23, 28, 30일, 4월 4, 6일

홍 성 수

sshong@redwood.snu.ac.kr

SNU RTOSLab 지도교수

서울대학교 전기정보공학부 교수

Seoul National University RTOS Lab

Agenda

I. Process Concepts
II. Process Scheduling
III. Context Switching
IV. Process Creation and Termination
V. Multithreading

I. Process Concepts

Process Concepts (1)

- What is a process and why is it useful?

- Why?
  - With many things happening at once in a system, need some way of separating them all out cleanly
  - Important concept: "*Decomposition*"
    - Solve a hard problem by chopping it into several simpler problems that can be solved separately

Process Concepts (2)

- What?
  - Definition of a process
    - *Program in execution*, or
    - An *execution stream* in the context of a particular *process state*
  - What is an "*execution stream*" and what is a "*process state*"?
    - Process state is everything that can affect, or be affected by the process
      - code, data values, open files, etc.
    - Execution stream is a sequence of instructions performed in a process state
      - Key simplifying feature of a process
      - Only one thing happens at a time within a process

Process Concepts (3)

- Process *state* or *context*
  - Collection of three types of contexts
    - Memory context
      - Code segment, data segment, stack segment, heap
    - Hardware context
      - CPU registers, I/O registers
    - System context
      - Process table, open file table, page table
  - *Realization* of the notion of process

Process Concepts (4)

- Multiprogramming vs. multiprocessing
  - *Uniprogramming*
    - Only one process in memory at a time
    - Mostly old PC OS
    - Makes some parts of OS easier, but others hard
  - Multiprogramming
    - Multiple processes in memory
    - Most systems support multiprogramming
  - Multiprocessing
    - Multiple processes are running together at the same time
    - CPU is multiplexed

Process Concepts (5)

- *Design-time* entity vs. *run-time entity*
  - *System design* is an activity of
    - Accepting the *system requirements*
    - Generating a collection of *tasks*
      - Design by decomposition
  - Task is a design-time entity
  - Process is a run-time entity
    - Target of CPU scheduling and resource allocation
  - *Implementation* is
    - A mapping from design-time entities onto run-time entities

Process Control Block

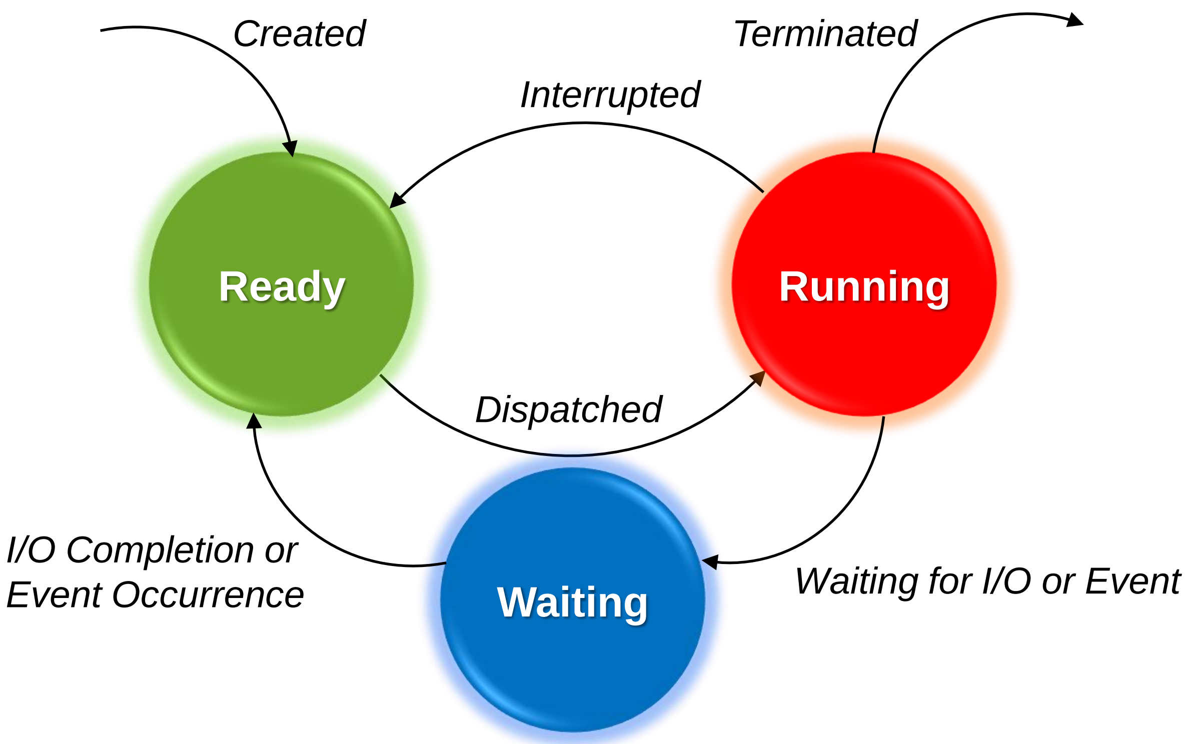
- With multiprocessing, OS must keep track of processes
  - For each process, a process control block (PCB) holds
    - Execution state (saved registers, etc.)
    - Scheduling information (priority)
    - Accounting and other misc. information (open files)
  - System-wide table of PCB
    - Process table
  - Unix
    - Fixed-size array of PCB's

State Transition (1)

- As a process executes, it changes state
  - New
    - Process is being created
  - Running
    - Instructions are being executed
  - Waiting
    - Process is waiting for some event to occur
  - Ready
    - Process is waiting to be assigned to CPU
  - Terminated
    - Process has finished execution

State Transition (2)

- State transition diagram



State Transition (3)

- State transitions and scheduling queues
  - Queues in different states
    - Ready queue
      - Set of all processes residing in main memory, ready and waiting to execute
    - Device queues (I/O waiting queues)
      - Set of processes waiting for an I/O device
  - State transition
    - Migrating processes between various queues

II. Process Scheduling

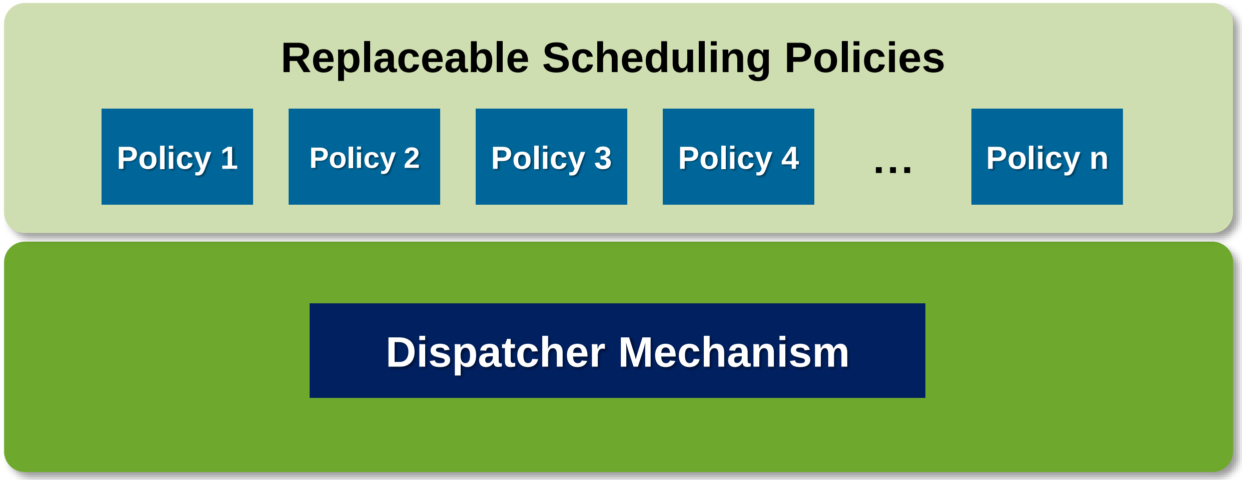
Process Scheduling

- Goal
  - For several processes to share a CPU, OS must select a process to run next

- Constraints
  - OS must offer
    - Fair scheduling
      - Make sure each process gets a chance to run
    - Protection
      - Making sure processes don't trash each other

Scheduler Design Principle

- Principle in designing system software
  - *Separation of policy and mechanism*
    - Separation of *scheduling policies* and *dispatching mechanisms*
  - Leads to two-level architecture



Dispatcher (1)

- Inner-most portion of OS that runs processes

```
loop forever
{
    run the process for a while
    stop it and save its state
    load state of another process
}
```

Dispatcher (2)

- Challenges
  1. How does the dispatcher regain control?
     - CPU can only be doing one thing at a time
     - User process running means that dispatcher isn't.
  2. Which process is executed next?
     - Need to locate runnable processes efficiently

1. Entering and Leaving the Kernel (1)

- How does the dispatcher regain control?
  - Trust the process to wake up the dispatcher
    - On a voluntary basis – "*non-preemptive*" way
    - Problem: Sometimes processes misbehave
  - Provide the dispatcher with an alarm clock
    - On a compulsory basis – "*preemptive*" way
    - Timer hardware and interrupts

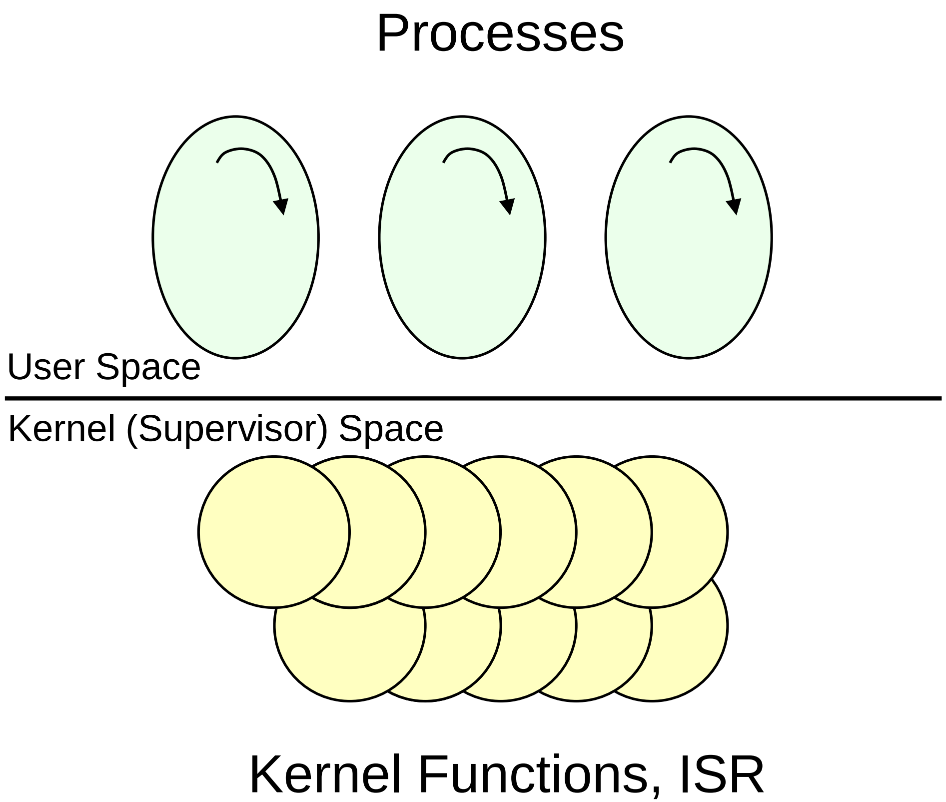
1. Entering and Leaving the Kernel (2)

- "*Misconceptions*" about the kernel
  - Like a user process, the kernel is an active and independent entity possessing a thread of control
  - The kernel is continuously monitoring user processes while they are running

- In reality
  - The kernel is a passive entity consisting of *kernel functions* and *interrupt service routines*
  - It's like a library

1. Entering and Leaving the Kernel (3)

- In reality (cont'd)
  - A *collection of functions* running in kernel space



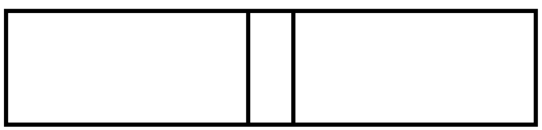
1. Entering and Leaving the Kernel (4)

- Kernel space (mode)
  - Has elevated system state compared to normal user applications
    - Protected memory space
    - Full access to the hardware
  - *Elevated system state + unrestricted memory access*
- User space (mode)
  - Has restricted system state compared to the kernel
    - A subset of the machine's available resources
    - Limited privilege
      - Unable to perform certain system functions
  - *Restricted system state + restricted memory access*

1. Entering and Leaving the Kernel (5)

- Execution modes in protected MMU machine

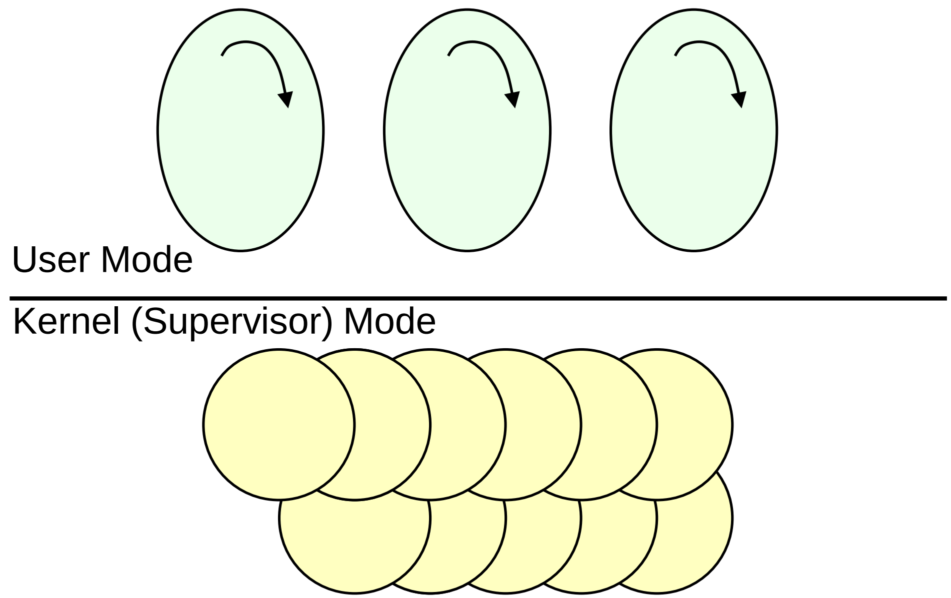
Processor Status Word (PSW)



Mode bit = 0: kernel mode

1: user mode

Processes



Kernel Functions, ISR

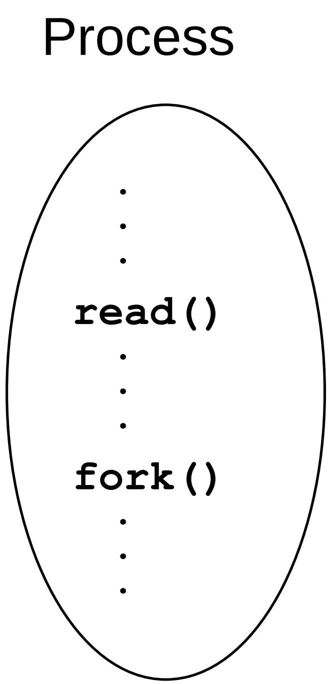
1. Entering and Leaving the Kernel (6)

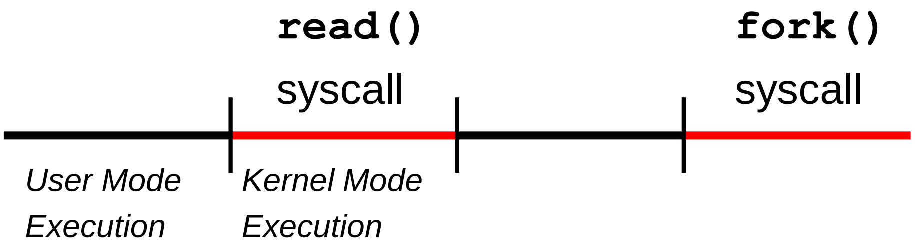
- Dispatcher is a kernel function after all

- Control returns to OS on
  - Traps: events internal to user processes
    - System calls
    - Errors (illegal instructions, address error, etc)
    - Page faults
  - Interrupts: events external from user processes
    - Character typed at a terminal
    - Completion of a disk transfer
    - Timer to make sure OS eventually gets control

1. Entering and Leaving the Kernel (7)

- Mode change of a process





1. Entering and Leaving the Kernel (8)

- *System call* vs. *function call*
  - Common properties
    - *Transfer* control to another routine
    - *Maintain* the context of the process
  - Differences
    - Syscall incurs *mode change* but function call doesn't
    - Syscall is more *expensive* than function call

2. Scheduling Policy (1)

- Once the dispatcher gets control, how to decide who's next?
  - Possibilities
    - Scan process table for first runnable process:
      - Might spend much time searching
      - Results in weird priorities: Small PIDs better
      - Question: How do you know a process is runnable?
    - Link together the runnable processes into a queue
      - Dispatcher takes from the head of the queue
      - Runnable processes are inserted at back of queue
      - Called "*ready list*" or "*run queue*"
    - Assign priorities to processes
      - Keep the queue sorted by priority
      - Separate queue per priority

2. Scheduling Policy (2)

- Who decides priorities and how are priorities chosen?
  - Who?
    - Separate part of OS: the scheduler
  - Question: Why not by the dispatcher?
    - Concept: Separation of policy and mechanism
  - How? Subject of the next topic

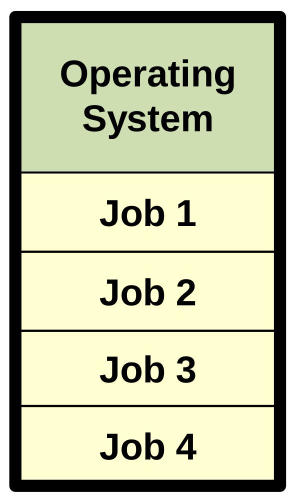
III. Context Switching

Context Switching (1)

- How does the dispatcher save and restore state?
  - Mechanism: "*context switch*"
- What must get saved?
  - Everything that next process could or will damage:
    - Program counter
    - Processor status word (condition codes, etc.)
    - General purpose registers, floating-point registers
    - All of memory?
      - Swapping
        - Memory could be large so saving it could be expensive

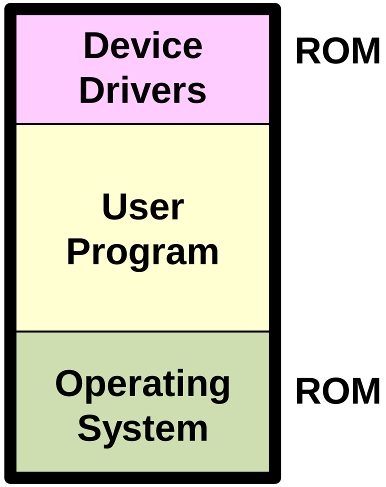
Context Switching (2)

- What must get saved? (cont'd)
  - Possibilities:
    1. Don't save memory at all
       - No dynamic memory management
         - Memory is allocated to entire batch
       - Old batch processing system: multiprogrammed batch monitor
       - Context switching in multithreaded process



Context Switching (3)

- What must get saved? (cont'd)
  - Possibilities: (cont'd)
    2. Save all memory to disk (roll-in/roll-out swapping)
       - Bringing in each process entirely, running it and then putting it back on the disk, so that another program may be loaded into that space
       - Early personal computer/workstation: DOS
       - Effective but very slow



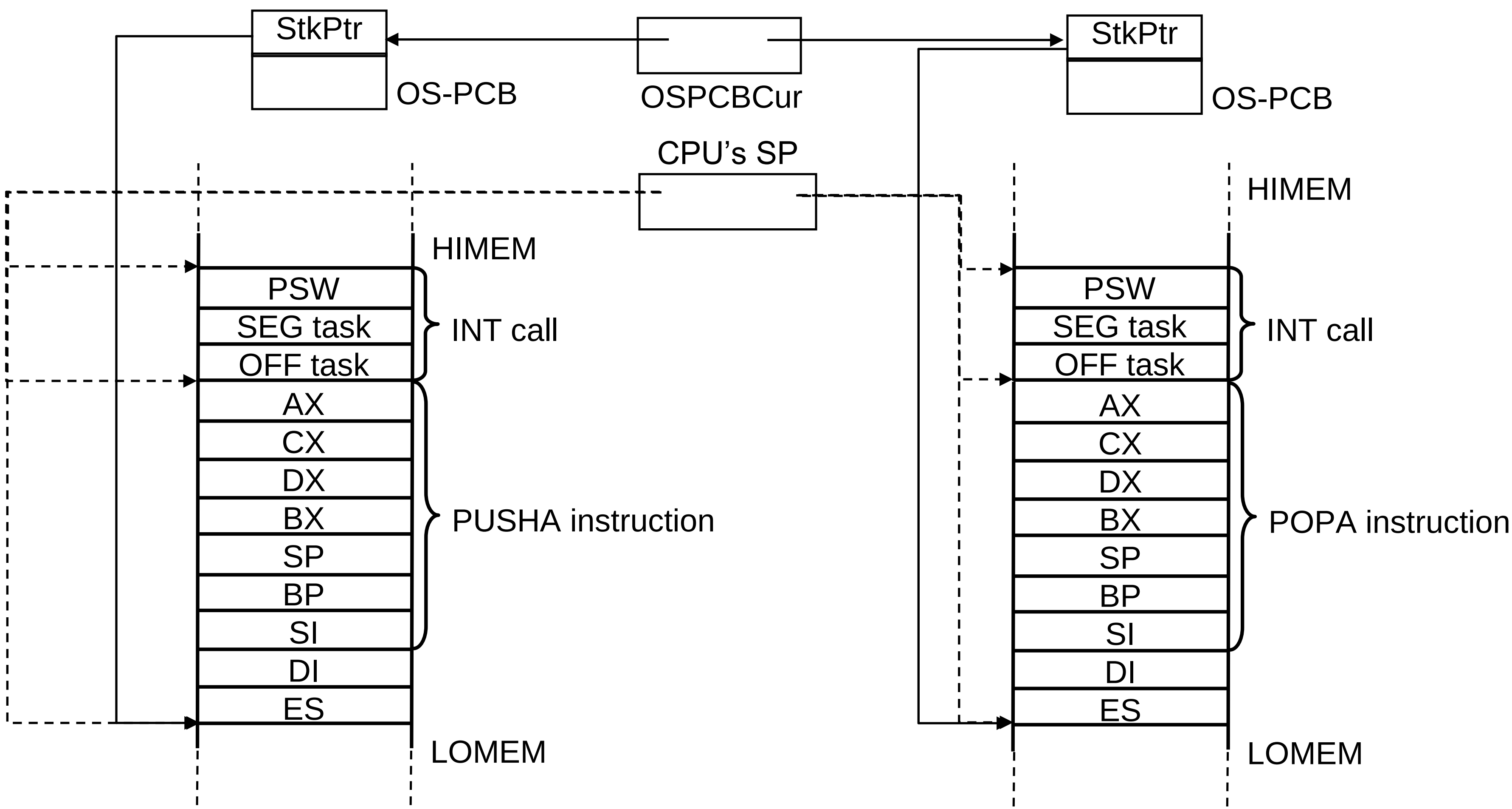
Context Switching (4)

- What must get saved? (cont'd)
  - Possibilities: (cont'd)
    3. Save some part memory to disk (swapping)
       - Moving memory blocks of process between RAM and disk
         - Swap file, swap device
       - Implemented with memory complex management mechanisms
       - Used in most of the modern OSes
         - Unix or Unix-like systems: Linux, OS X

Implementation

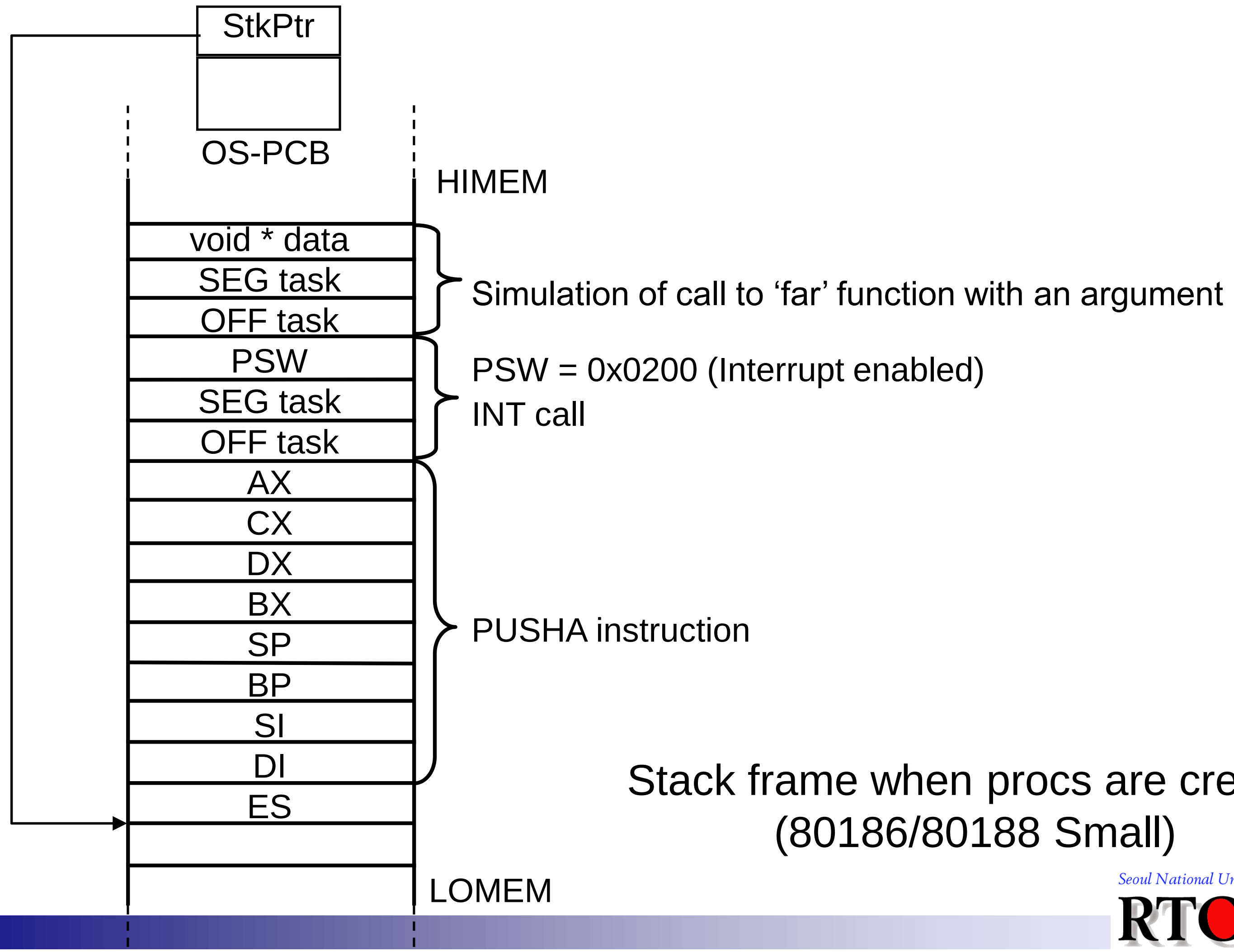
- Machine dependent
  - Different for MIPS, SPARC, x86, etc.
- Tricky
  - OS must execute code to save state without changing the process' state
- Requires some special hardware support
  - Example: "*Save PC and PSR on trap or interrupt*"

Mechanism (1)



Stack frames during a context Switch (80186/80188 Small)

Mechanism (2)



Stack frame when procs are created
(80186/80188 Small)

IV. Process Creation and Termination

Process Creation (1)

- Creating new processes in a full-fledged OS
  - Build one from scratch (Ex: Unix Process 0)
  - Clone an existing one (Ex: Unix **`fork()`** syscall)

Process Creation (2)

- From scratch
  1. Load code and data into memory
  2. Create (empty) call stack
  3. Create and initialize a process control block
  4. Put the process on ready list

  - Intuitive and natural – This is what Windows OS does

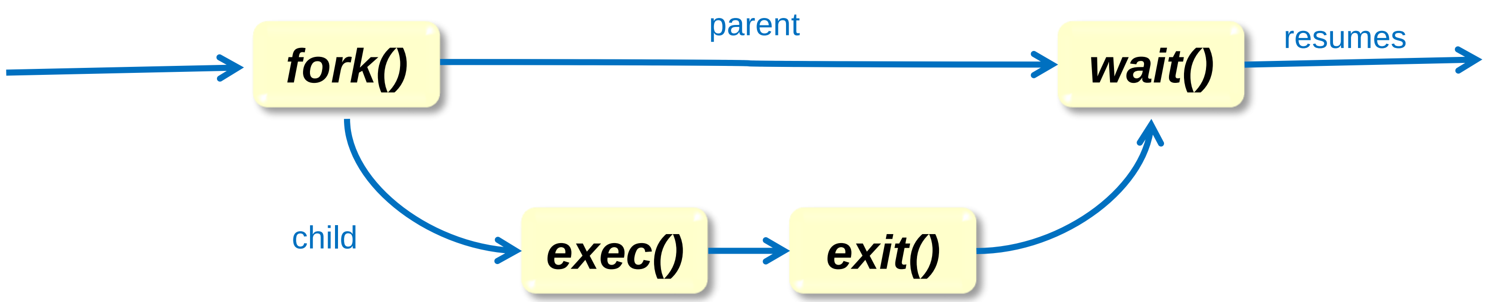
Process Creation (3)

- Cloning
  1. Stop the current process and save its state
  2. Create a new one by making a copy of code, data, stack, and PCB
  3. Put the new process on ready list

  - Not quite right – What's missing?
    - Process creation in Unix with **`fork()`** and **`exec()`**

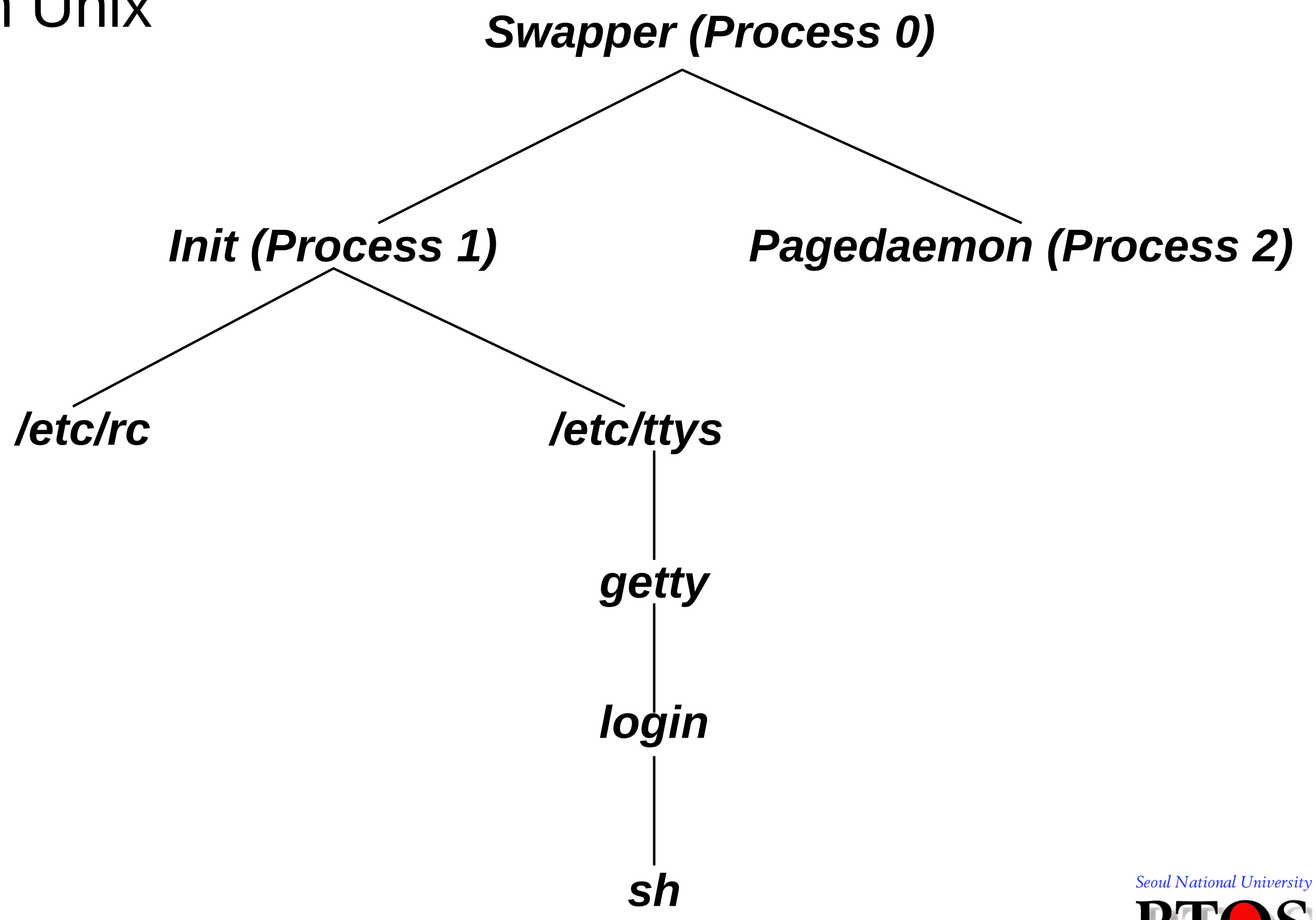
Process Creation (4)

- Process life cycle in Unix



Process Creation (5)

- In Unix



Process Creation (6)

- Shell example

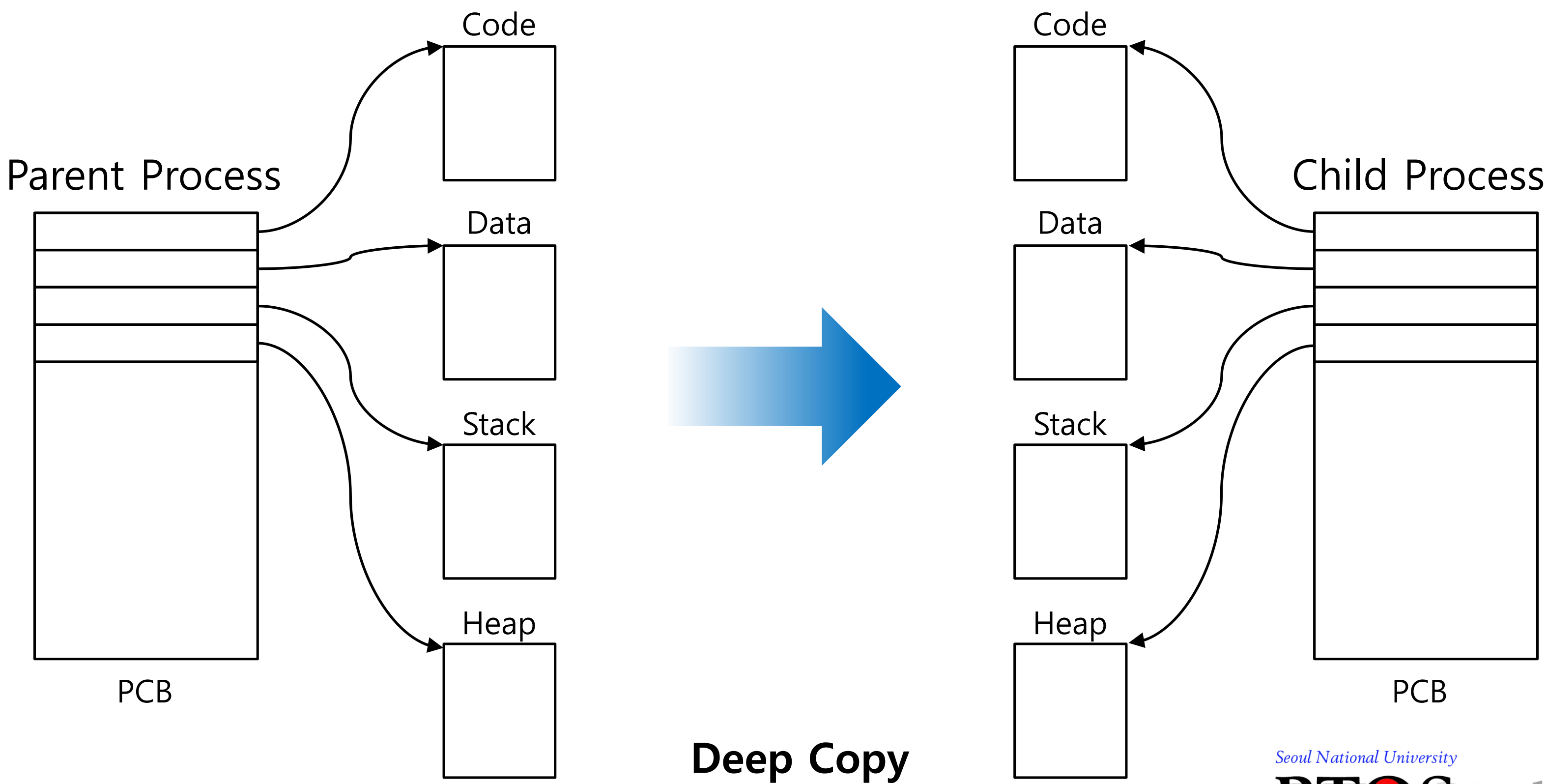
```
for(;;) {
    cmd = readcmd();
    pid = fork();
    if(pid < 0){
      perror("fork failed");
      exit(-1);
    } else if(pid == 0) {
      // Child - Setup environment
      if(exec(cmd) < 0) perror("exec failed");
      exit(-1); // Exit on exec failure
    } else {
      // Parent - Wait for command to finish
      wait(pid);
    }
 }
```

Process Creation (7)

- Questions surrounding the **`fork()`** mechanism
  1. What were the drawbacks of the original **`fork()`**?
  2. Why did early Unix adopt it after all?
  3. Why is it still used in Linux?

Process Creation (8)

1. What were the drawbacks of the original `fork()`?
   - *Deep copy*-based cloning was simply too expensive



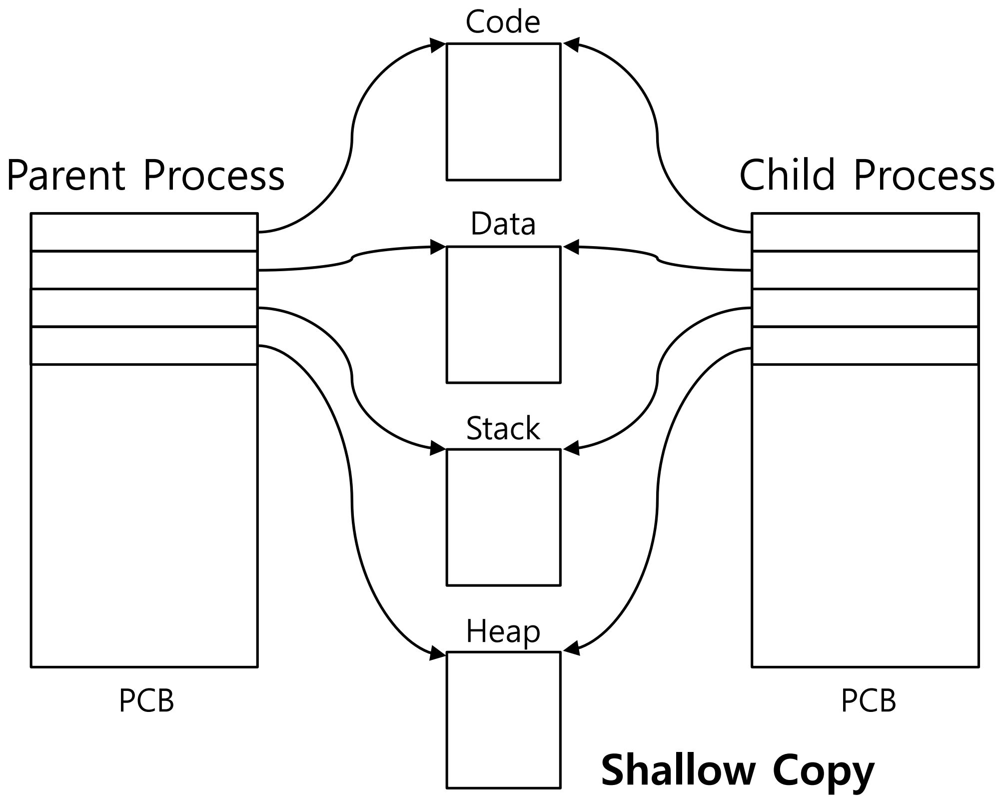
Process Creation (9)

2. Why did early Unix adopt **`fork()`** after all?
   - Due to the lack of inter-process communication mechanisms

```
ipc_proc()
{
    fd = open("./fifo_pipe", O_RDWR);

    pid = fork();
    if(pid > 0){
      // Parent - write data to the pipe
      write(fd, data, size);
    } else if(pid == 0) {
      // Child - read data from the pipe
      read(fd, data, size);
    }
}
```

Process Creation (10)

3. Why is it still used in Linux?
   - Thanks to *shallow copy* and *copy-on-write* (COW)



Process Termination

- Process executes last statement and asks the OS to decide it (**`exit()`**)
  - Outputs data from child to parent (via **`wait()`**)
  - Deallocates process' resources
- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - OS does not allow child to continue if its parent terminates
    - Cascading termination

V. Multithreading

Traditional Process Model

- Two characteristics in one process
  - Unit of *resource ownership*
    - Assigned virtual address space to hold the process image
    - Has the control of some resources (files, I/O devices, ...)
  - Unit of *dispatching*
    - Has a thread of control
    - Has execution state and dispatching priority
      - Process execution may be interleaved with other processes

- *How about separating the two*?

Multithreaded Process Model

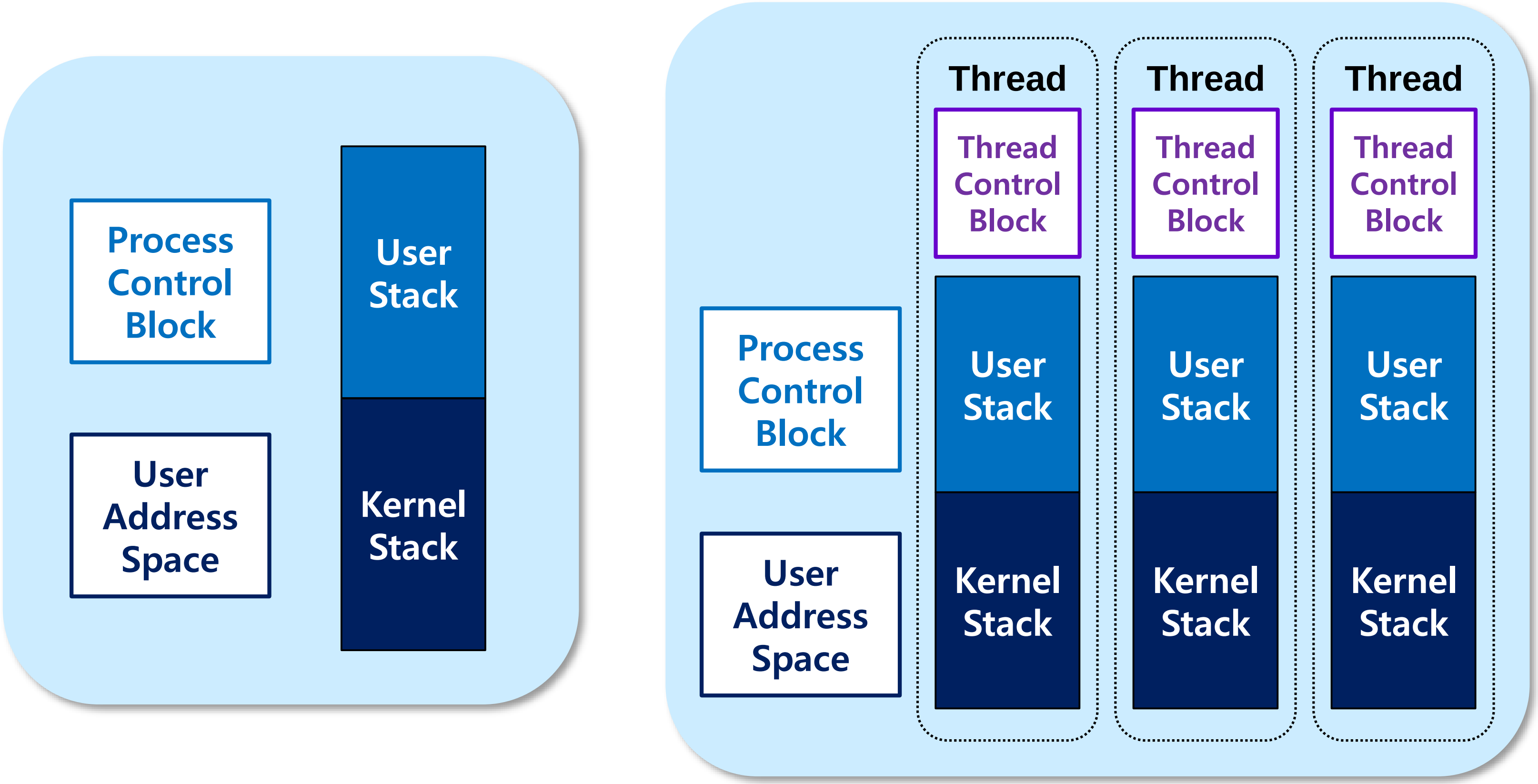
- Most modern OSes treat these two characteristics independently
  - Unit of resource ownership is usually referred to as "*process*" or "*task*"
  - Unit of dispatching is usually referred to as "*thread*" or "*lightweight process*"

Multithreading: Basics (1)

- Characteristics of threads
  - Has an execution state (running, ready, stopped)
    - Saves thread context when not running
  - Has a *runtime stack* for local variables and some *per-thread static memory*
  - Has access to the memory address space and resources of its process
    - All threads of a process share these
    - When one thread alters a (non-private) memory item, all other threads (of the process) see that
    - A file opened by a thread is available to others

Multithreading: Basics (2)

- Single threading vs. multithreading



Multithreading: Basics (3)

- Various thread supports in OS
  - MS-DOS
    - Supports a single user process and a single thread
  - Old Unix
    - Supports multiple user processes
      but only supports one thread per process
  - Solaris
    - Supports multiple user processes possessing multiple threads

Multithreading: Basics (4)

- State transition of threads
  - Three key states
    - Running, ready, blocked
  - They have no *suspended* (i.e., swapped-out) state
    - All threads of the same process share the same address space
    - Suspending a single thread involves suspending all threads in the same address space
  - Termination of a process terminates all threads within the process

Why Multithreading? (1)

- Effective concurrent programming (original goal)
  - Straightforward mapping from threads onto multiprocessors
- Resource sharing
  - Can pass data via shared memory
    - No need for IPC
    - Need to synchronize the activities of various threads so that they do not obtain inconsistent views of the data

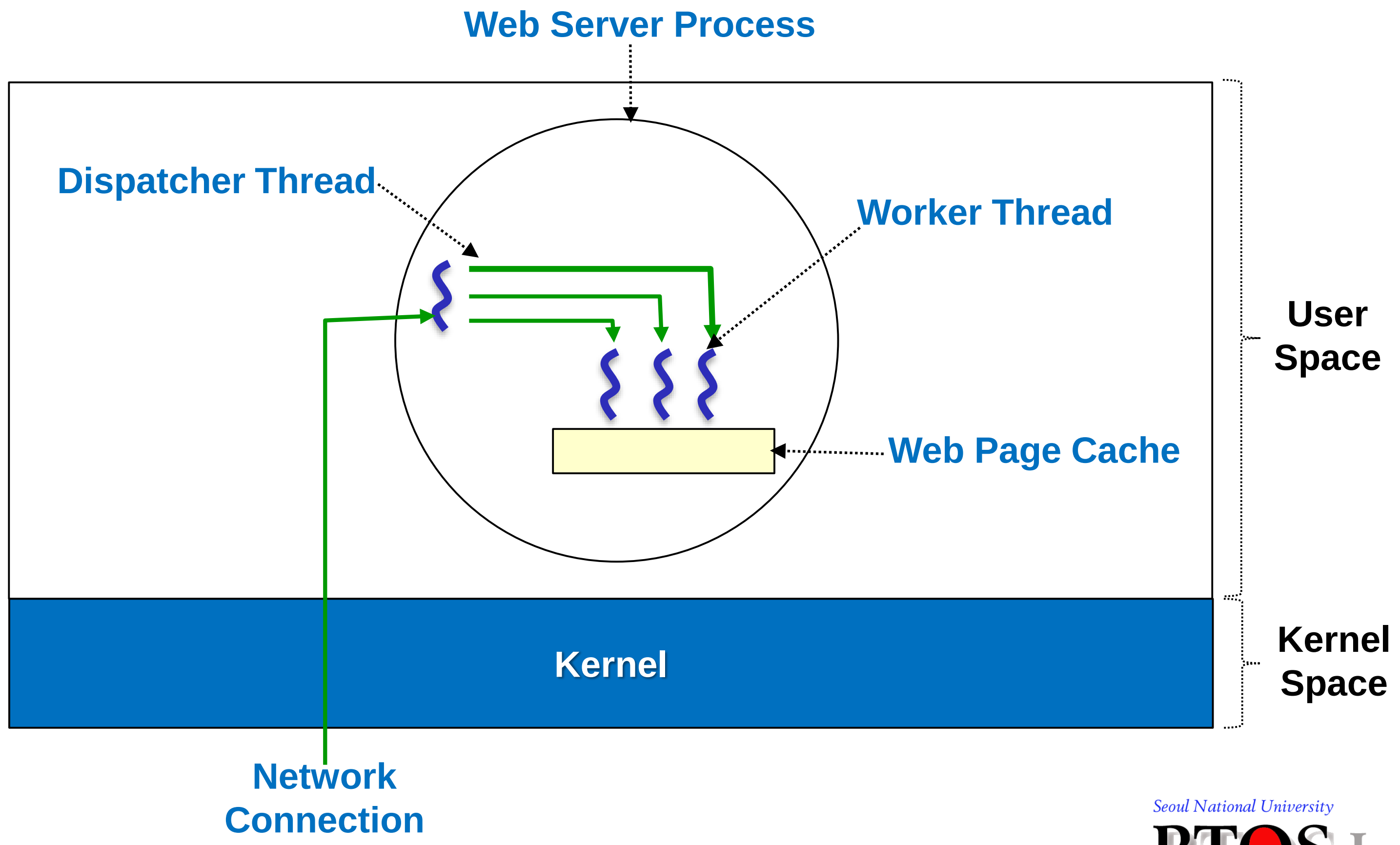
Why Multithreading? (2)

- Economy – cheap to implement
  - Takes less time
    - To create a new thread than a process
    - To terminate a thread than a process
    - To switch between two threads within the same process
  - Uses very little resource
    - Only stack and per-thread static memory

- Agility in responses (good for reactive systems)
  - *Concurrent server* architecture for interactive applications
    - A process has one server thread and multiple worker threads
      - Even if one worker blocks, possibly on a read, others still continue executing and produce outputs to users

Why Multithreading? (3)

- Multithreading fits for concurrent server architecture



Why Multithreading? (4)

- Pseudocode for concurrent server architecture

```
while (TURE){
    get_next_request(&buf);
    dispatch_work(&buf);
}
```

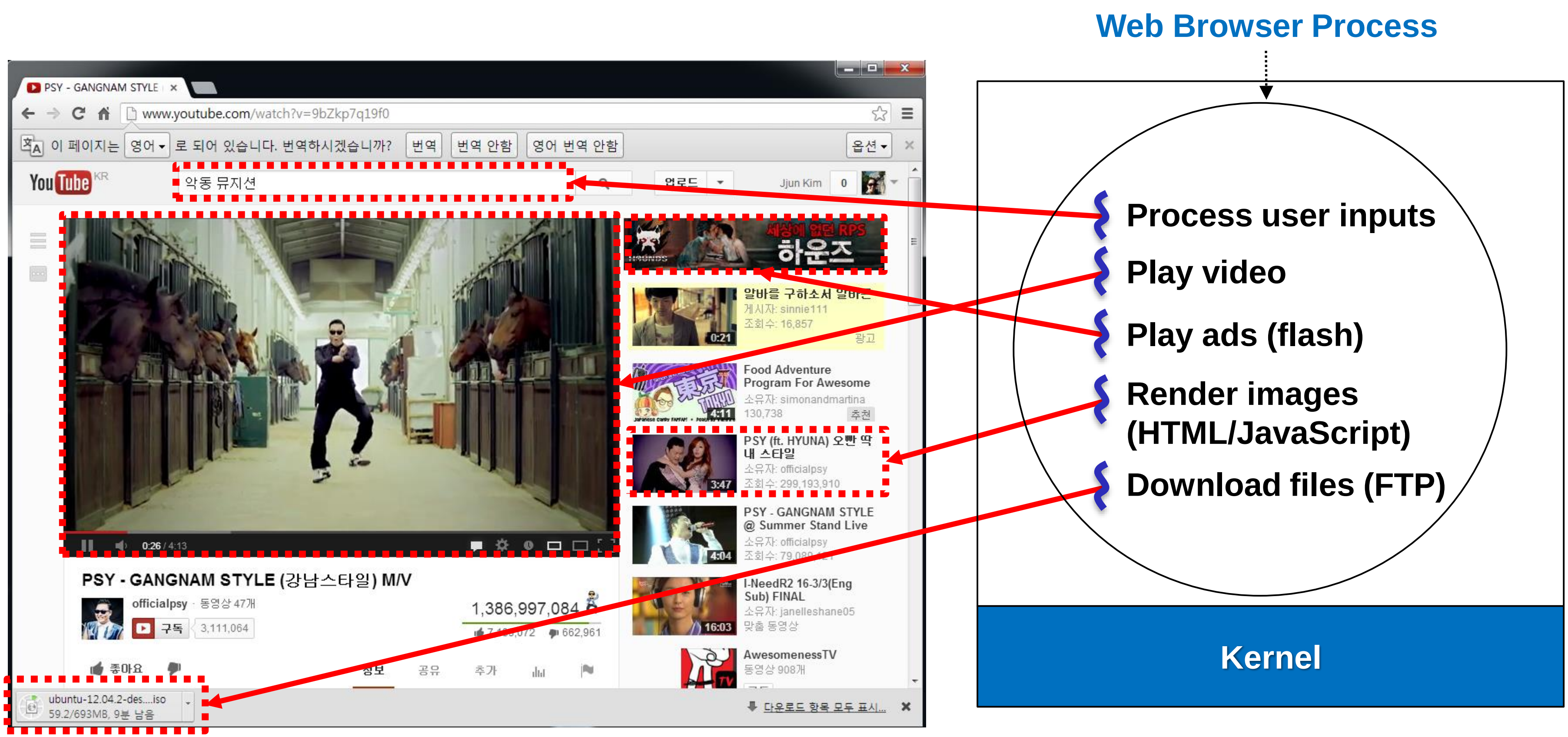
Dispatcher Thread

```
while (TURE){
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Worker Thread

Why Multithreading? (5)

- Multithreaded Web browser



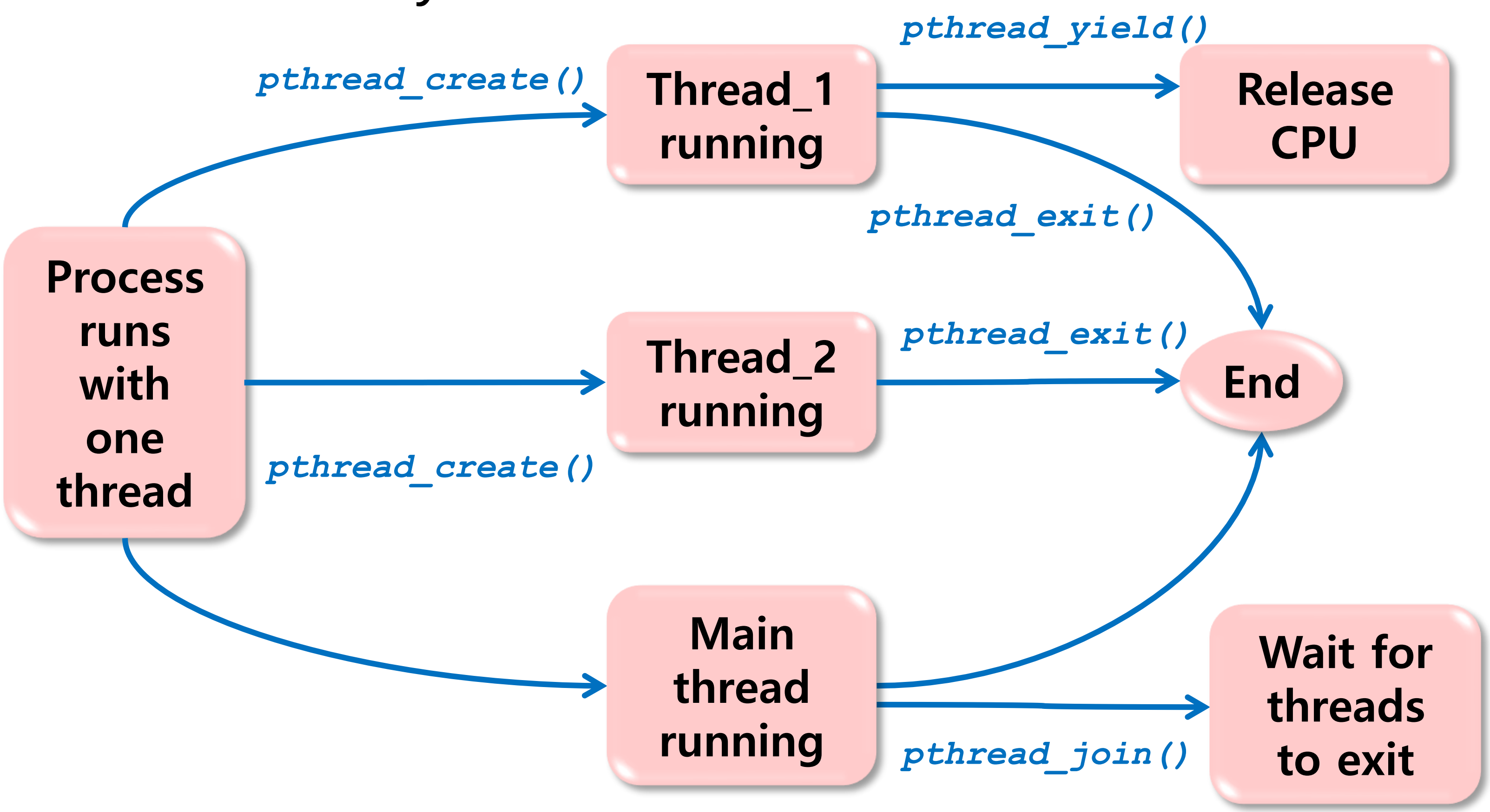
Pthreads Programming Model (1)

- Pthreads: POSIX standard for threads
  - Defines API for creating and manipulating threads
  - Implementations of the API are available on many Unix-like OSes such as Linux and Mac OS X
- Selected Pthreads functions

Pthreads API	Description
`pthread_create()`	Create a new thread
`pthread_exit()`	Terminate the calling thread
`pthread_join()`	Wait for a specific thread to exit
`pthread_yield()`	Release CPU to let another thread run

Pthreads Programming Model (2)

- Thread life cycle



Pthreads Programming Model (3)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     5

void *ThreadCode(void *argument)
{
   int tid;

   tid = *((int *)argument);
   printf("Hello World! It's me, thread %d!\n", tid);

   /* optionally: insert more useful stuff here */

   return NULL;
}
```

Source: http://en.wikipedia.org/wiki/POSIX_Threads

Pthreads Programming Model (4)

```
int main(void)
{
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

    /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, ThreadCode, (void *)&thread_args[i]);
      assert(0 == rc);
   }
    /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join(threads[i], NULL);
      assert(0 == rc);
   }
   exit(EXIT_SUCCESS);
}
```

Source: http://en.wikipedia.org/wiki/POSIX_Threads

Pthreads Programming Model (5)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

__thread int TLS_data;
int normal_data;

void *thread(void *param)
{
    int *data;

    data = (int *)param;
    TLS_data = *data;
    normal_data = *data;
    if (*data == 0) usleep(100000);

    printf("Thread %d, TLS_data %d, normal_data %d\n",
                            *data, TLS_data, normal_data);
}
```

Pthreads Programming Model (6)

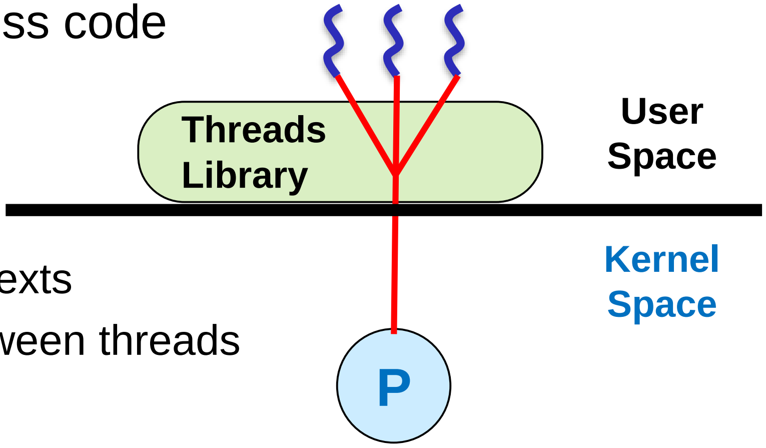
```
int main()
{
    pthread_t tcb[3];
    int thread_args[3];
    int i;

    for (i = 0; i < 3; i++)
    {
        thread_args[i] = i;
        pthread_create(&tcb[i], NULL, thread, (void *)&thread_args[i]);
    }

    for (i = 0; i < 3; i++)
    {
        pthread_join(tcb[i], NULL);
    }

    return 0;
}
```

Pthreads Programming Model (7)

```
sshong@ubuntu:~$ gcc tls.c -lpthread
sshong@ubuntu:~$ a.out
Thread 2, TLS_data 2, normal_data 2
Thread 1, TLS_data 1, normal_data 1
Thread 0, TLS_data 0, normal_data 1
```

Threads Implementation: User-Level Threads (1)

- Key entities
  - Thread:
    - 100% use-level entity
    - Kernel is not aware of the existence of threads
  - Threads library
    - User-level library linked to process code
    - Contains code for
      - Creating and destroying threads
      - Scheduling thread execution
      - Saving and restoring thread contexts
      - Passing messages and data between threads
  - Processor
    - Allocation on a process basis



Threads Implementation: User-Level Threads (2)

- Characteristics
  - Application does all the thread management via threads library
    - Thread switching does not require kernel mode privileges
    - Scheduling is application-specific
  - Kernel activities
    - Kernel is not aware of thread activity but still manages process activity
    - When a thread makes a blocking system call
      - The whole process will be blocked
      - For the thread library, that thread is still in the running state
    - Implication:
      - *Thread states are independent of process states*

Threads Implementation: User-Level Thread (3)

- Pros and cons of ULT

Advantages

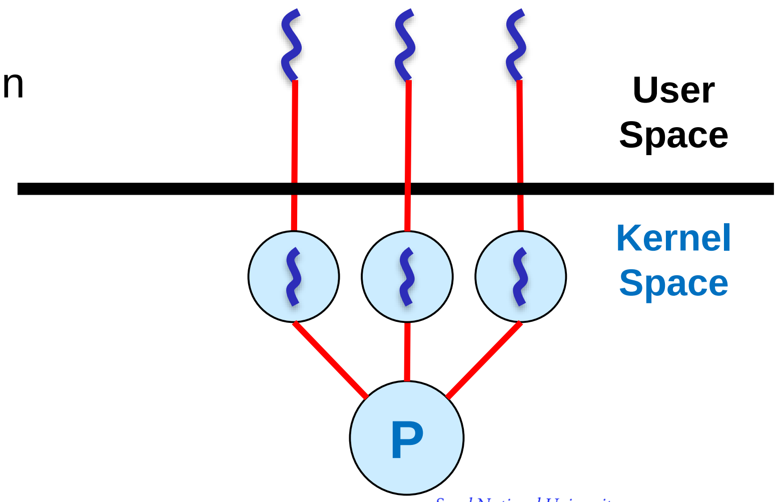
- Thread switching does not involve the kernel - no mode switching
- Scheduling can be application specific - can choose the best algorithm
- ULTs can run on any OS: only needs a thread library

Inconveniences

- Most system calls are blocking and the kernel blocks the process: all threads within the process will be blocked
- Kernel can only assign processes to processors: two threads within the same process cannot run simultaneously on two processors

Threads Implementation: Kernel-Level Threads (1)

- Key entities
  - Thread:
    - Both user-level and kernel-level entity
      - 1-to-1 mapping between user-level and kernel level thread
    - The kernel is completely aware of the existence of threads
  - System call API and kernel functions for thread facility
    - Code for
      - Maintaining context information for processes and threads
      - Switching between threads
      - Scheduling threads
      - Synchronization
  - Processor
    - Allocation on a thread basis



Threads Implementation: Kernel-Level Threads (2)

- Characteristics
  - No thread library but API to the kernel thread facility
    - Need kernel modification
  - Scheduling on a thread basis
    - Kernel-level threads are scheduling entities
  - Ex: Windows NT and OS/2

Threads Implementation: Kernel-Level Threads (3)

- Pros and cons of KLT

Advantages

- Kernel can simultaneously schedule many threads of the same process on many processors
- Blocking is done on a thread level
- Kernel routines can be multithreaded

Inconveniences

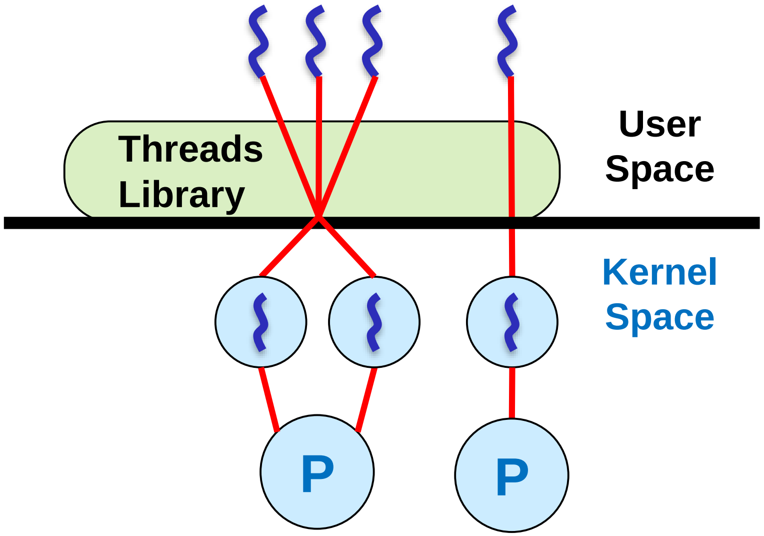
- Thread switching within the same process involves the kernel: we have two mode switches per thread switch
- This results in a significant slowdown

Threads Implementation: Combined UL/KL Threads (1)

- Key entities
  - User-level thread
    - The kernel is mostly unaware of the existence of threads
  - Kernel-level thread
    - Serves as virtual processor to user-level threads
    - Schedulable entity
  - Threads library
    - Contains code for
      - Creating/destroying user-level threads
      - Scheduling thread execution
      - Saving and restoring thread contexts
      - Passing messages and data between threads

Threads Implementation: Combined UL/KL Threads (2)

- Key entities (cont'd)
  - System call API and kernel functions for thread facility
    - Code for
      - Creating/destroying kernel-level threads
      - Mapping/unmapping between user-level and kernel level threads
      - Maintaining context information for processes and threads
      - Switching between threads
      - Scheduling threads
  - Processor
    - Allocation on a thread basis



Threads Implementation: Combined UL/KL Threads (3)

- Characteristics
  - Thread creation done in user space
  - User-level scheduling for sharing kernel-level threads
  - Kernel-level scheduling on a thread basis
  - Synchronization of threads done in user space
  - Programmer may adjust the number of KLTs
  - May combine the best of both approaches
  - Ex: Solaris