Lecture 06 Process Management

Dr. Tushar, Mosaddek Hossain Kamal
Professor

Computer Science and Engineering, University of Dhaka,

BSc Third Year, Semester 2 (Sep – Feb), Academic Year: 2024

CSE3201: Operating Systems

Oct 31, 2024

Outline

Learning Outcome

Learning Outcome

- **Understand the Operating System Process**
  - Process and Thread
  - Process and Thread context, data Structure
  - PCB and TCB
- **Understand the Process Management**
  - Context Switch and Dispatcher
  - Process Scheduling
  - Process and Thread Synchronization tools
  - Deadlock
- **Realtime process**
  - Concept of Soft ane Hard real time process
  - Reatime Process Scheduling

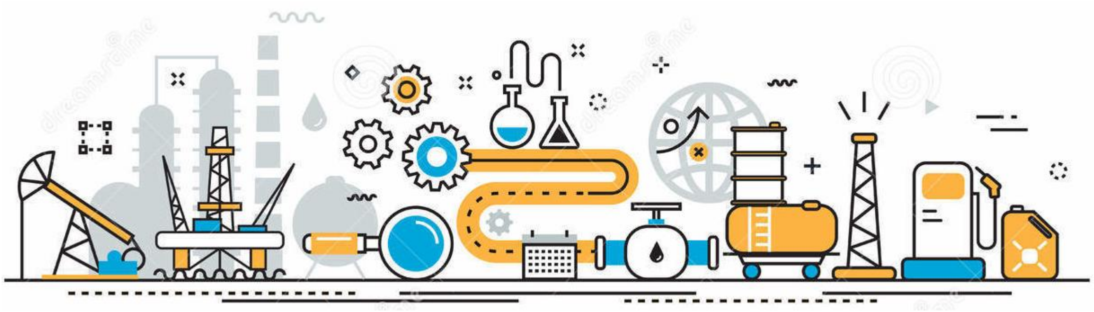
Major Requirements of an Operating System

OS Major Requirement

- Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support interprocess communication and user creation of processes
- Provide a convenient environment to the developer community
- Present an easy usable, and understandable system hiding complex hardware

Process and Thread

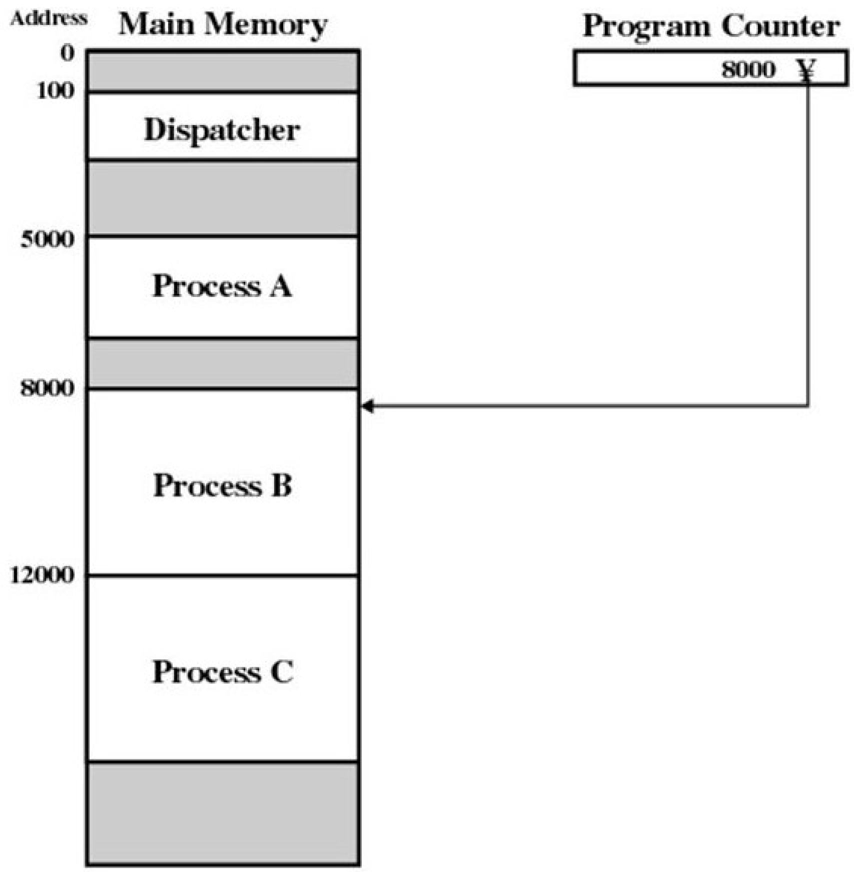
- Processes:
  - Also called a task or job
  - Execution of an individual program
  - “Owner” of resources allocated for program execution
  - Encompasses one or more threads
- Threads:
  - Unit of execution
  - may be a part of a process
  - Can be traced and minimize switching overhead
- List the sequence of instructions that execute
  - Belongs to a process



Process Execution

Execution snapshot of three single-threaded processes (No Virtual Memory)

- Dispatcher (SysTick, PendSV)
- Program Counter (PC)
- Stack Pointer
- Memory Allocation



Logical Execution Trace

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
0x00005000	0x00008000	0x00012000
0x00005001	0x00008001	0x00012001
0x00005002	0x00008002	0x00012002
0x00005003	0x00008003	0x00012003
0x00005004		0x00012004
0x00005005		0x00012005
0x00005006		0x00012006
0x00005007		0x00012007
0x00005008		0x00012008
0x00005009		0x00012009
0x0000500A		0x0001200A
0x0000500B		0x0001200B

0x00005000 = Starting address of program of Process A
0x00008000 = Starting address of program of Process B
0x00012000 = Starting address of program of Process C

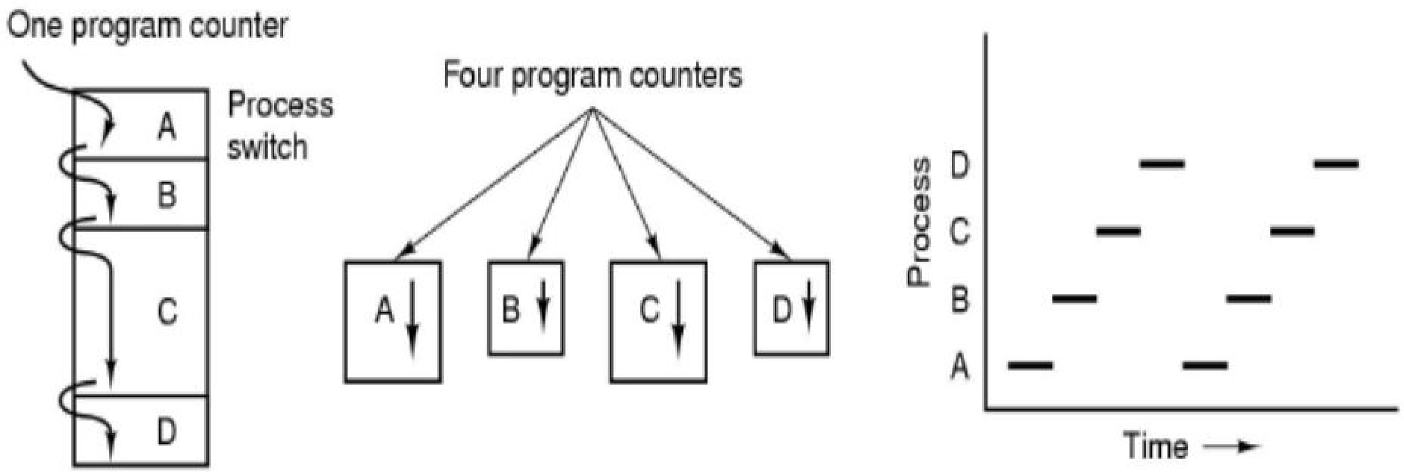
Combined Traces CPU Instructions

- Combined CPU Traces (Actual CPU Instruction) or execution sequence
- What are the Pink Shaded Area?
- 0x00000100 = Starting Address of Dispatcher Program.
- Shaded area indicates execution of the dispatcher program.
- First and third columns present the count of instruction cycle.
- Second and fourth columns show the address of instruction being executed.

1	0x00005000
2	0x00005001
3	0x00005002
4	0x00005003
5	0x00005004
6	0x00005005
	——— Timeout
7	0x00000100
8	0x00000101
9	0x00000102
10	0x00000103
11	0x00000104
12	0x00000105
13	0x00008000
14	0x00008001
15	0x00008002
16	0x00008003
	——— I/O Request
17	0x00000100
18	0x00000101
19	0x00000102
20	0x00000103
21	0x00000104
22	0x00000105
23	0x00012000
24	0x00012001
25	0x00012002
26	0x00012003

27	0x00012004
28	0x00012005
	——— Timeout
29	0x00000100
30	0x00000101
31	0x00000102
32	0x00000103
33	0x00000104
34	0x00000105
35	0x00005006
36	0x00005007
37	0x00005008
38	0x00005009
39	0x0000500A
40	0x0000500B
	——— Timeout
41	0x00000100
42	0x00000101
43	0x00000102
44	0x00000103
45	0x00000104
46	0x00000105
47	C0x00012006
48	0x00012007
49	0x00012008
50	0x00012009
51	0x0001200A
52	0x0001200B
	——— Timeout

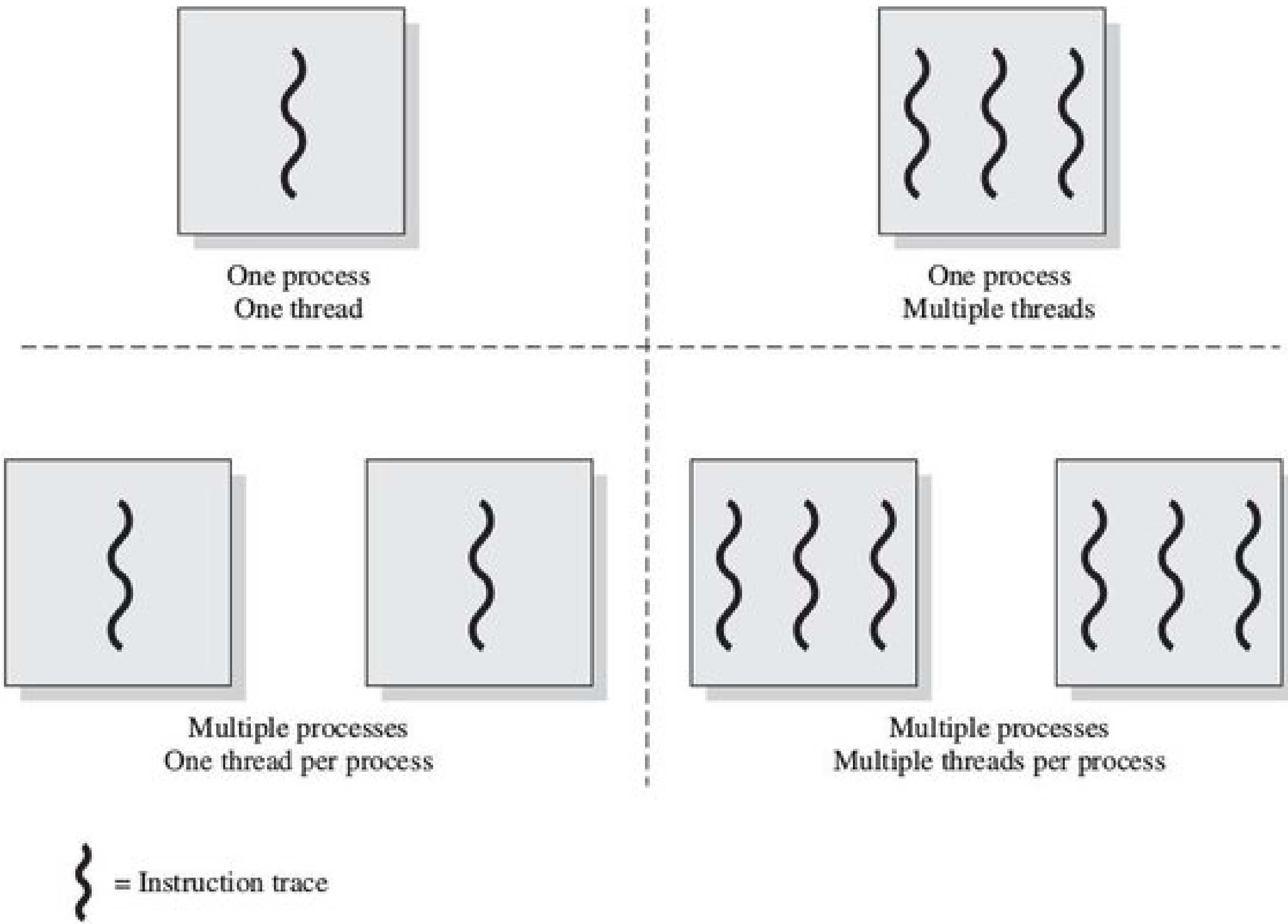
Summary of Process Model



(a) (b) (c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant

Thread and Process



Process and thread models of selected OSes

Process and thread models of selected OSes

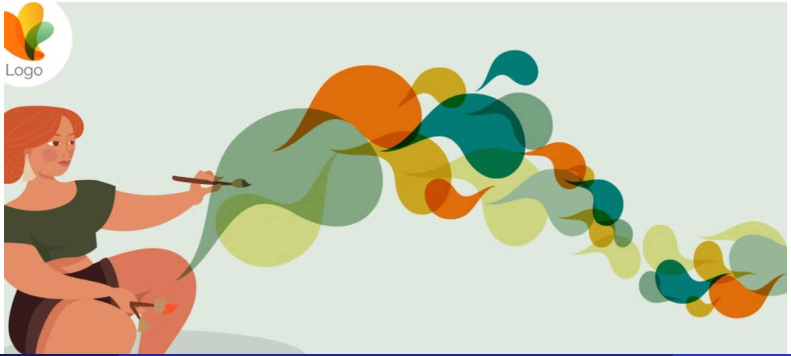
- Single process, single thread
  - MSDOS – still exist the core of windows OS in a modified form
- Single process, multiple threads
  - Harvard University: OS/161 as distributed (OS courses)
  - Envision for DU CSE OS
- Multiple processes, single threads
  - Traditional unix
- Multiple processes, multiple threads
  - Modern Unix (Linux, Solaris), Modern Windows OS
- Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)

Process Creation

Principal That Causes Process Creation

1. System initialization
   - Foreground processes (interactive programs)
   - Background processes
     - Email server, web server, print server, etc.
     - Called a daemon (unix) or service (Windows)
2. Execution of a process creation system call by a running process
   - New login shell for an incoming telnet connection
3. User request to create a new process
4. Initiation of a batch job

Note: Technically, all these cases use the same system mechanism to create new processes.

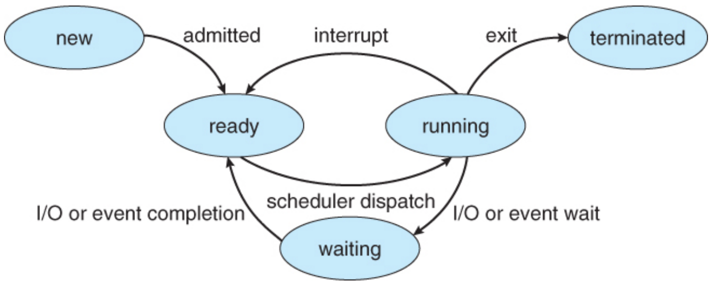


Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process and Thread State



- Process blocks for input
- Scheduler picks another process
- Scheduler picks this process
- Input become available

- Possible process/thread states
  - New
  - **Running**
  - **Blocked (waiting)**
  - **Ready**
  - terminated
- Transitions between states shown
  - Dispatch (CPU, Memory, and Resources Scheduling)

Some Transition Causing Events

- Running → Ready
  - Voluntary **Yield()**
  - End of timeslice
- Running → Blocked
  - Waiting for input
    - File, Nework, Keyboard
  - Waiting for a timer (alarm signal)
  - Waiting for a resource to become available

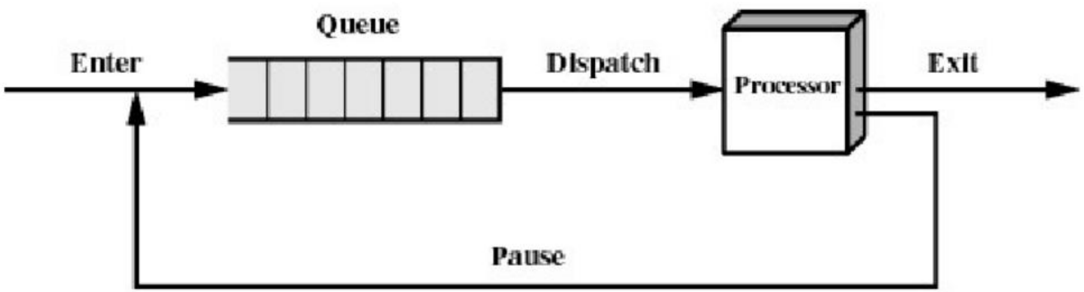
"Now that you've taken over the ship, let's discuss a transitional plan."

Dispatcher

Dispatcher

- Sometimes also called the scheduler
  - The literature is also a little inconsistent on this point
- Has to choose a Ready process to run
  - How?
  - It is inefficient to search through all processes

Ready Queue

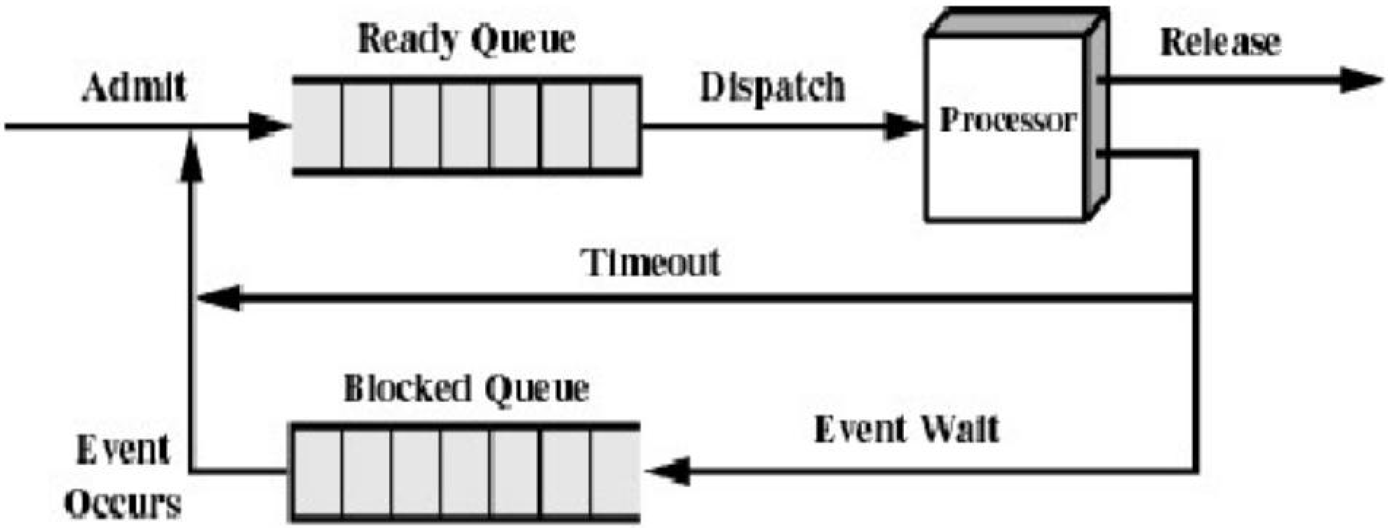


(b) Queuing diagram

What about blocked processes?

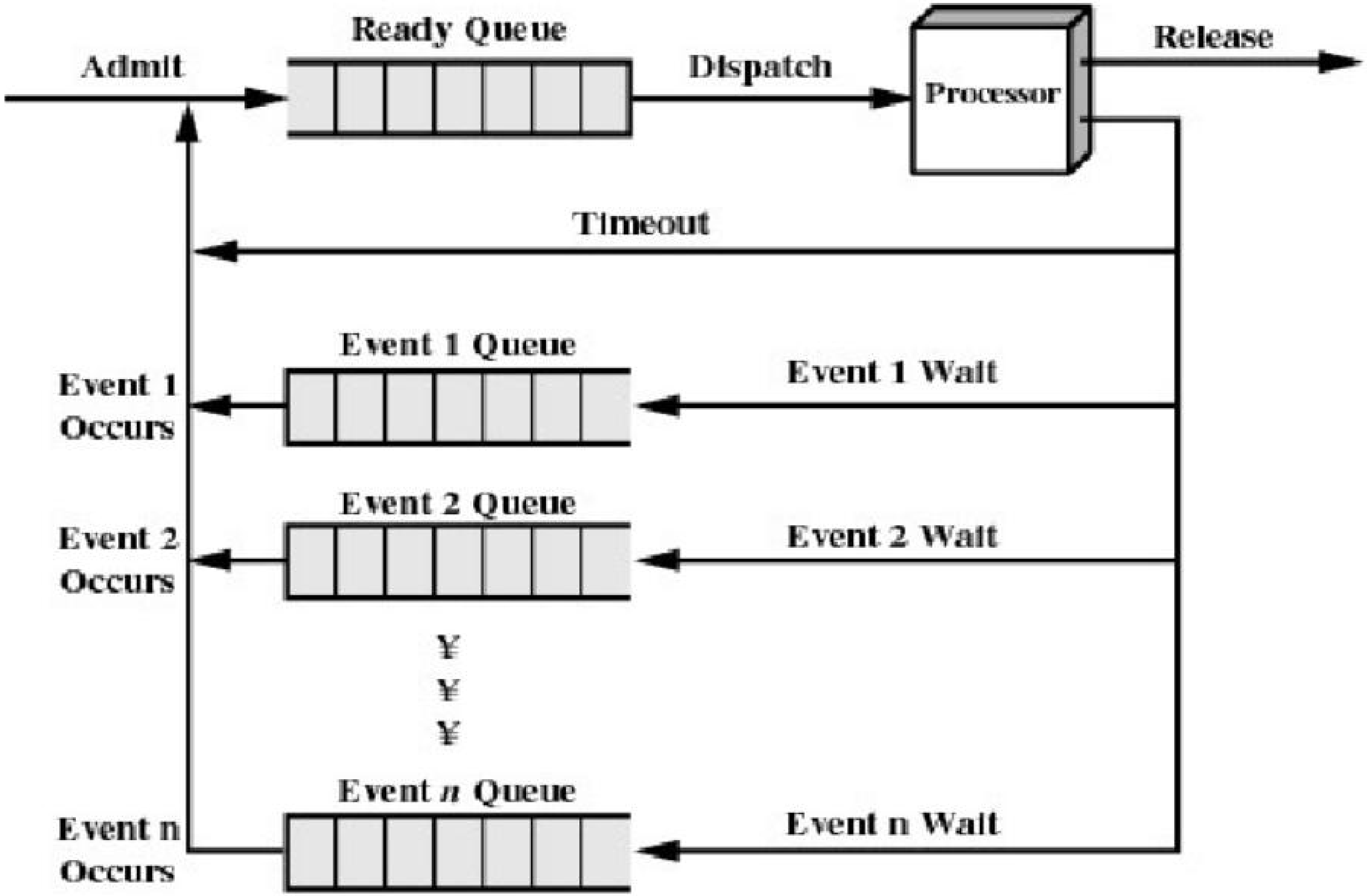
- When an unblocking event occurs, we also wish to avoid scanning all processes to select one to make Ready

Using Two Queues



(a) Single blocked queue

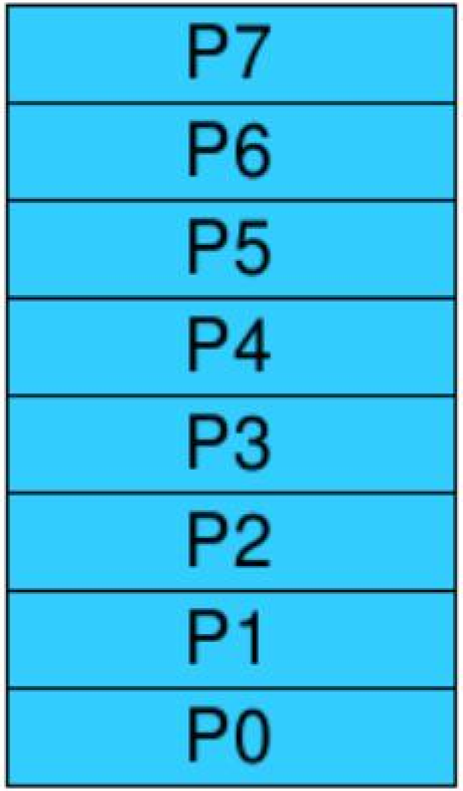
Multiple Blocked Queue



(b) Multiple blocked queues

Implementation of Processes

- A processes' information is stored in a process control block (PCB)
- The PCBs form a process table
  - Sometimes the kernel stack for each process is in the PCB
  - Sometimes some process info is on the kernel stack
    - e.g. registers in the stackframe in OS



Implementation of Processes

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 1: Example fileds of a process table – PCB

PCB – Process Control Block

Thread Model

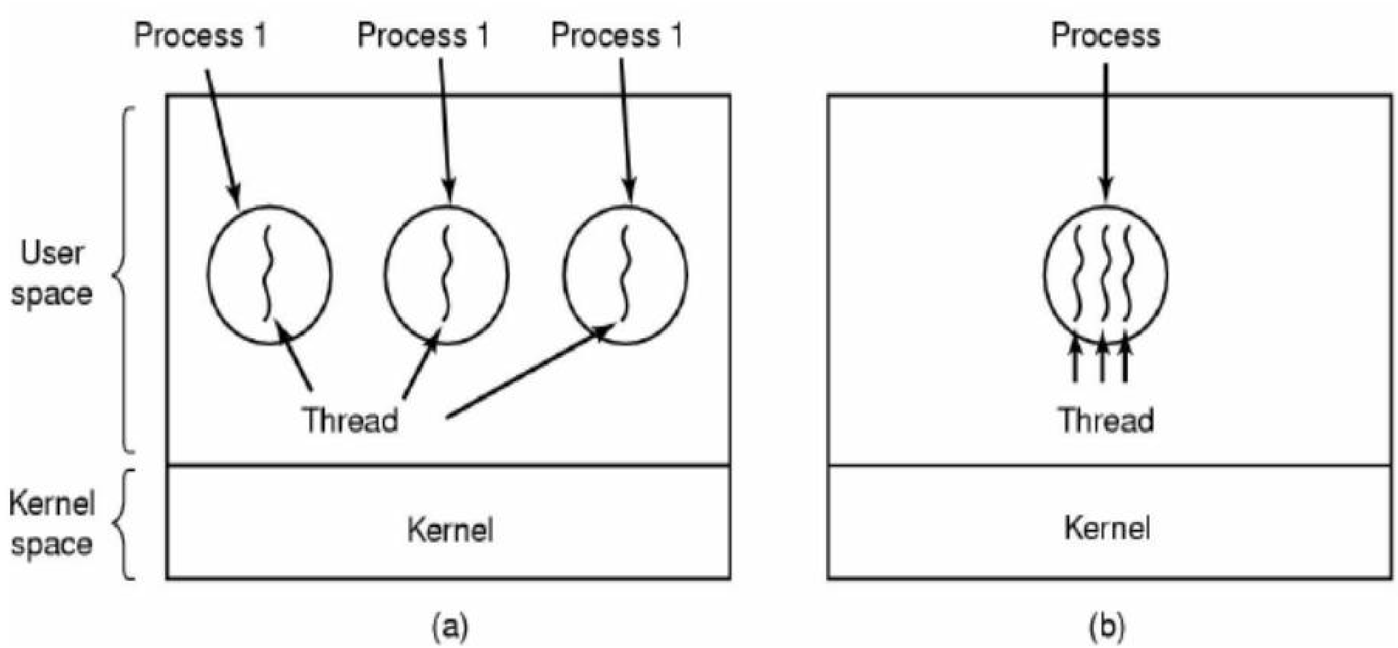


Figure 2: (a) Three process each one thread (b) One process with three thread

The Thread Model

TCB – Thread Components/attributes

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 3: OS Thread fileds (Col1) Item shared by all thread, (Col2) Item private to each thread

TCB – Thread Control Block

Thread Model

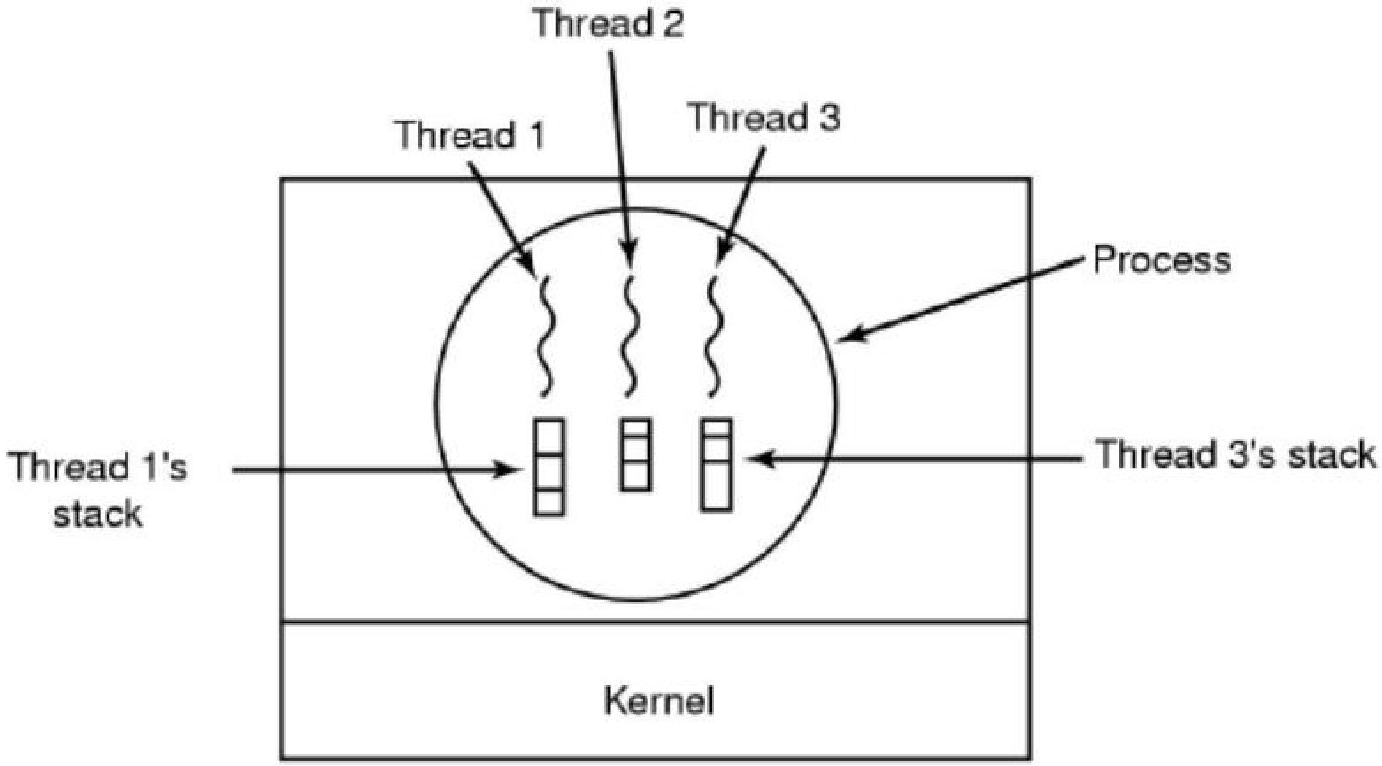


Figure 4: **Each thread has its own stack**

Thread Model

- Local variables are per thread
  - Allocated on the stack
- Global variables are shared between all threads
  - Allocated in data section
  - Concurrency control is an issue
- Dynamically allocated memory (malloc) can be global or local
  - Program defined (the pointer can be global or local)

Thread Usage

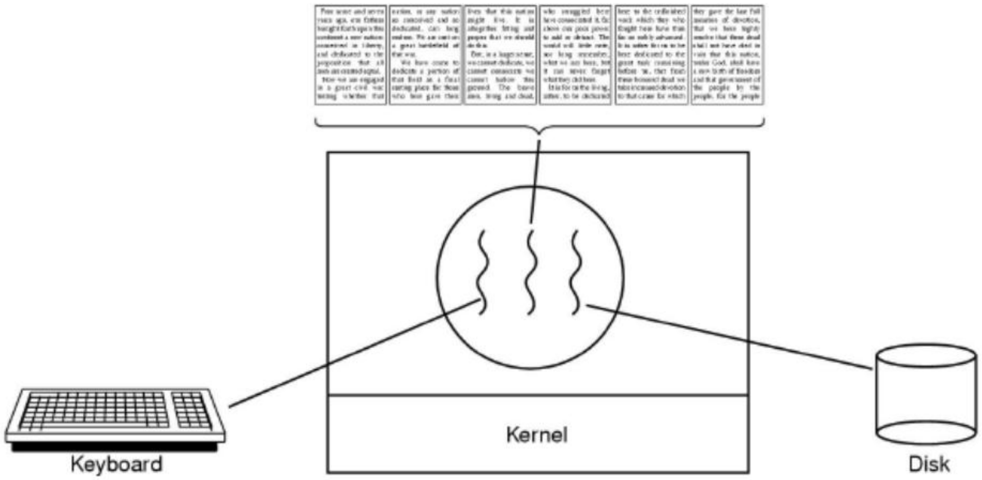


Figure 5: A word Processor with Three Threads

Thread Usage

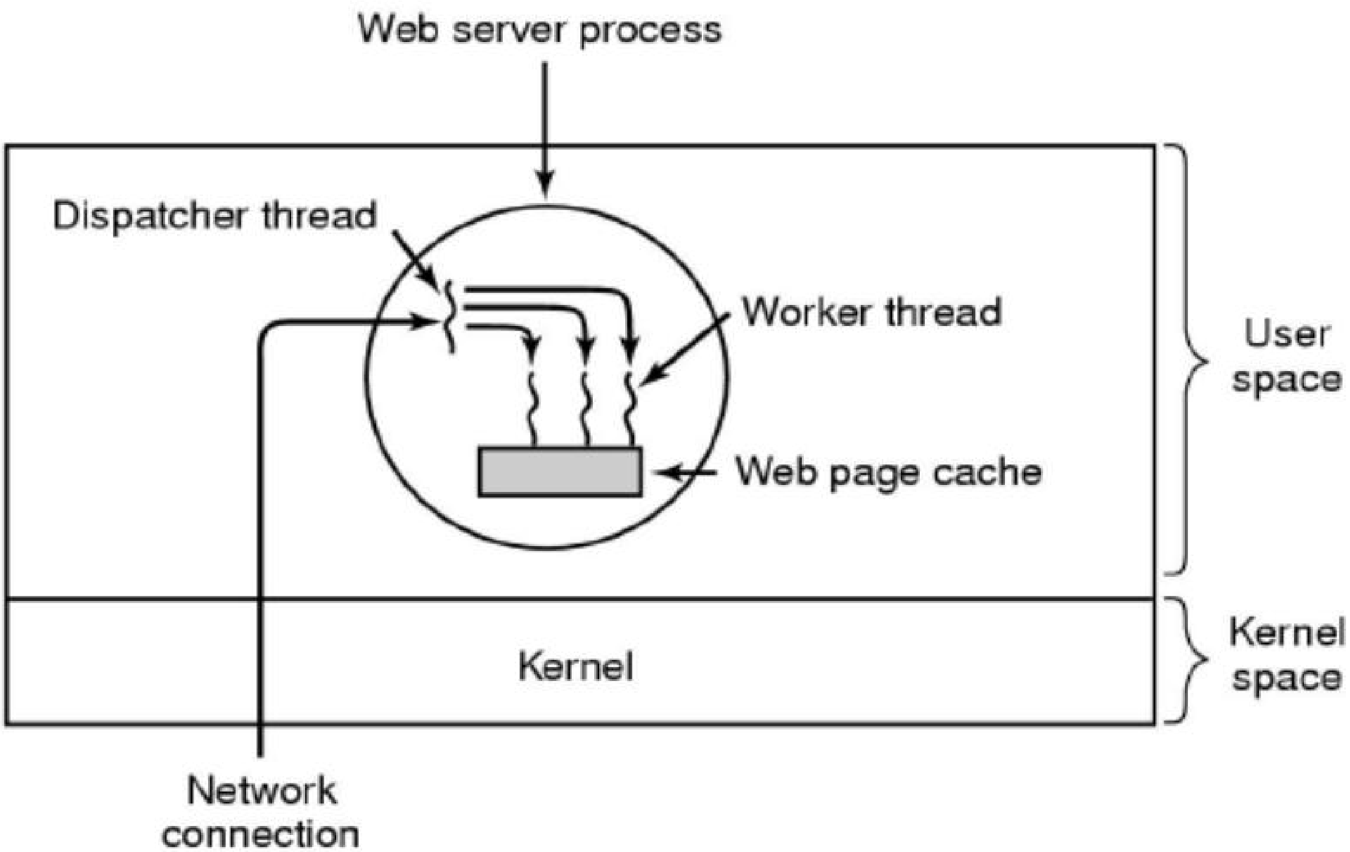


Figure 6: A multi-threaded Web Server

Thread Usage

Rough outline of code for previous slide

(a) Dispatcher thread
(b) Worker thread

```
while(TRUE){
 get_next_request(&buf);
 handoff_work(&buf);
}
```

(a) Dispatcher Thread

```
while(TRUE){
 wait_for_work(&buf);
 look_fpr_page_in_cache(&buf,&page);
 if(page_no_in_cache(&page))
  read_page_from_disk(&buf,&page);
 return_page(&page);
}
```

(b) Worker Thread

Thread Usage

Model	**Characteristics**
Threads	Parallelism, Blocking System Call
Single-threaded Process	No Parallelism, blocking System Calls
Finite-State machine	No Parallelism, nonblocking System Calls, interrupts

Three ways to construct a server

Summarizing "Why Threads?"

We need thread because:

- Simpler to program than a state machine
- Less resources are associated with them than a complete process
  - Cheaper to create and destroy
  - Shares resources (especially memory) between them
- Performance: Threads waiting for I/O can be overlapped with computing threads
  - Note if all threads are compute bound, then there is no performance improvement (on a uniprocessor)
- Threads can take advantage of the parallelism available on machines with more than one CPU (multiprocessor)

Implementing Threads in User Space

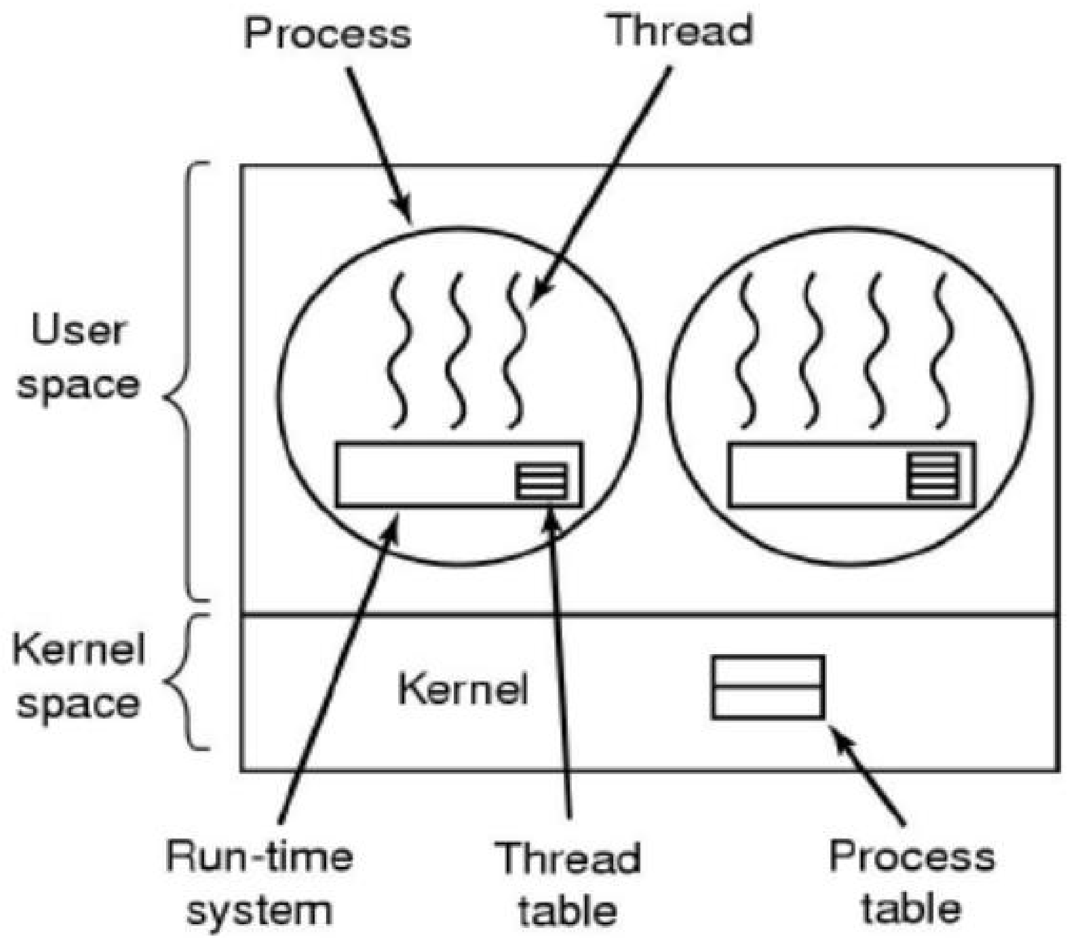


Figure 7: A user-level threads package

User-level Threads

Implementation at user-level

- User-level Thread Control Block (TCB), ready queue, blocked queue, and dispatcher
- Kernel has no knowledge of the threads (it only sees a single process)
- If a thread blocks waiting for a resource held by another thread, its state is save and the dispatcher switches to another ready thread
- Thread management (create, exit, yield, wait) are implemented in a runtime support library

User-Level Thread

Pros

- Thread management and switching at user level is much faster than doing it in kernel level
  - No need to trap into kernel and back to switch
- Dispatcher algorithm can be tuned to the application
  - E.g. use priorities
- Can be implemented on any OS (thread or non- thread aware)
- Can easily support massive numbers of threads on a per-application basis
  - Use normal application virtual memory
  - Kernel memory more constrained. Difficult to efficiently support wildly differing numbers of threads for different applications.

User-Level Thread

Cons

- Threads have to yield() manually (no time interrupt deliver to user-level)
  - Co-operative multithreading
    - A single poorly design/implementation thread can monopolise the available CPU time
  - Threr are work-arounds (e.g., a timer signal per second to enable pre-emptive multithreading), they are course grain and a kludge
- Does not take advantage of multiple CPUs (in reality, we still have a single threaded process as far as the kernel is concerned)

User-Level Threads – Cons

If a thread makes a blocking system call (or takes a page fault), the process (and all the internal threads) blocks

- Can't overlap I/O with computation
- can use wrappers as a work around
  - Example: wrap the read() call
  - use select() to itself if read system call would block
    - Example: wrap read call
    - use select() then read()
    - Only call read if it won't block
    - Otherwise schedule another thread
  - Wrapper requires 2 system calls instead one
    - Wrappers are needed for environments doing lots of blocking system calls?
- Can change to kernel to support non-blocking system call
  - Lose "on any system" advantage, page faults still a problem

Implementing Threads in the Kernel

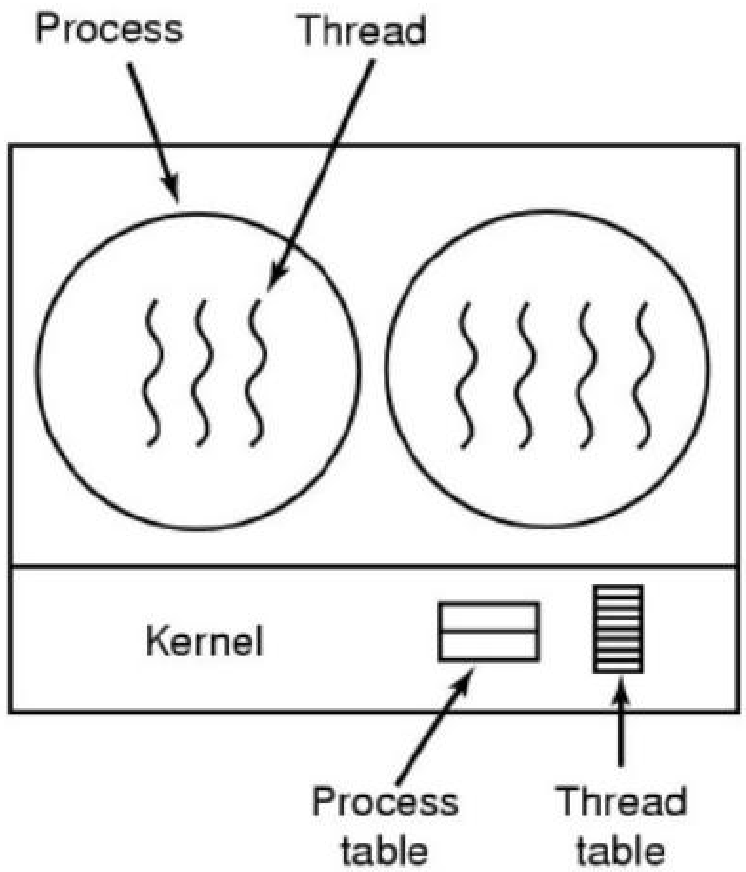


Figure 8: A thread Package Managed By Kernel

Kernel Threads

Threads are implemented in the kernel

- TCBs are stored in the kernel
  - A subset of information in the traditional PCB
    - The subset related to execution context
  - TCBs have a PCB associated with them
    - Resources associated with the group of threads (the process)
- Thread management calls are implemented as system call
  - e.g., create, wait, exit

Cons

- Thread creation and destruction, and blocking and unblocking threads required kernel entry and exit
  - More expensive than user-level equivalent

Pros

- Preemptive multithreading
- Parallelism
  - Can overlap blocking I/O with computation
  - Can take advantage of a multoprocessor

Multiprogramming Implementation

Multiprogramming Implementation

1. Hardware stack and program counter, etc.
2. Hardware loads new program counter from interrpt vector
3. assembly language procedure sets up new stack
4. C interrupt service runs (typically reads abd buffer input)
5. Scheduler decides which process is run next
6. C procedure returns to assembly code
7. Assembly language procedure starts up new current process

Skeleton of what lowest lwevel of OS does when an interrupt occurs – a thread/context switch

Thread Switch

A switch between threads can be happen any time the OS is invoked

- On a system call
  - Mandatory if system call blocks or on exit();
- On an exception
  - mandatory if offender is killed
- on an iterrupt/exception
  - Triggering a dispatch is the main purpose of the timer interrupt

A thread switch can happen between any two instructions

Note istructions donot equal program statement

Context switch

Thread switch must be transparent for threads

- When dispatch again, thread should not notice that something else was running in the mean time (except for elasped time)

OS must save all state that affects the thread

- This state is called the thread context
- Switching between threads consequently results in a context switch

Simplified Explicit Thread Switch

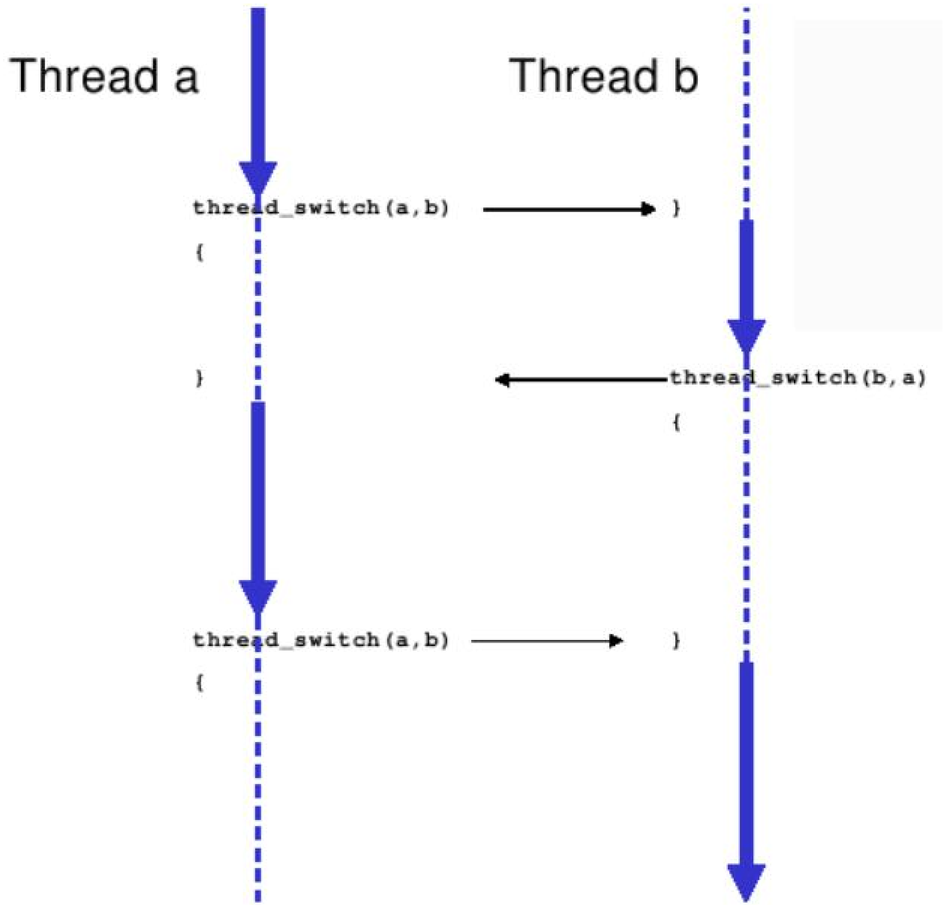
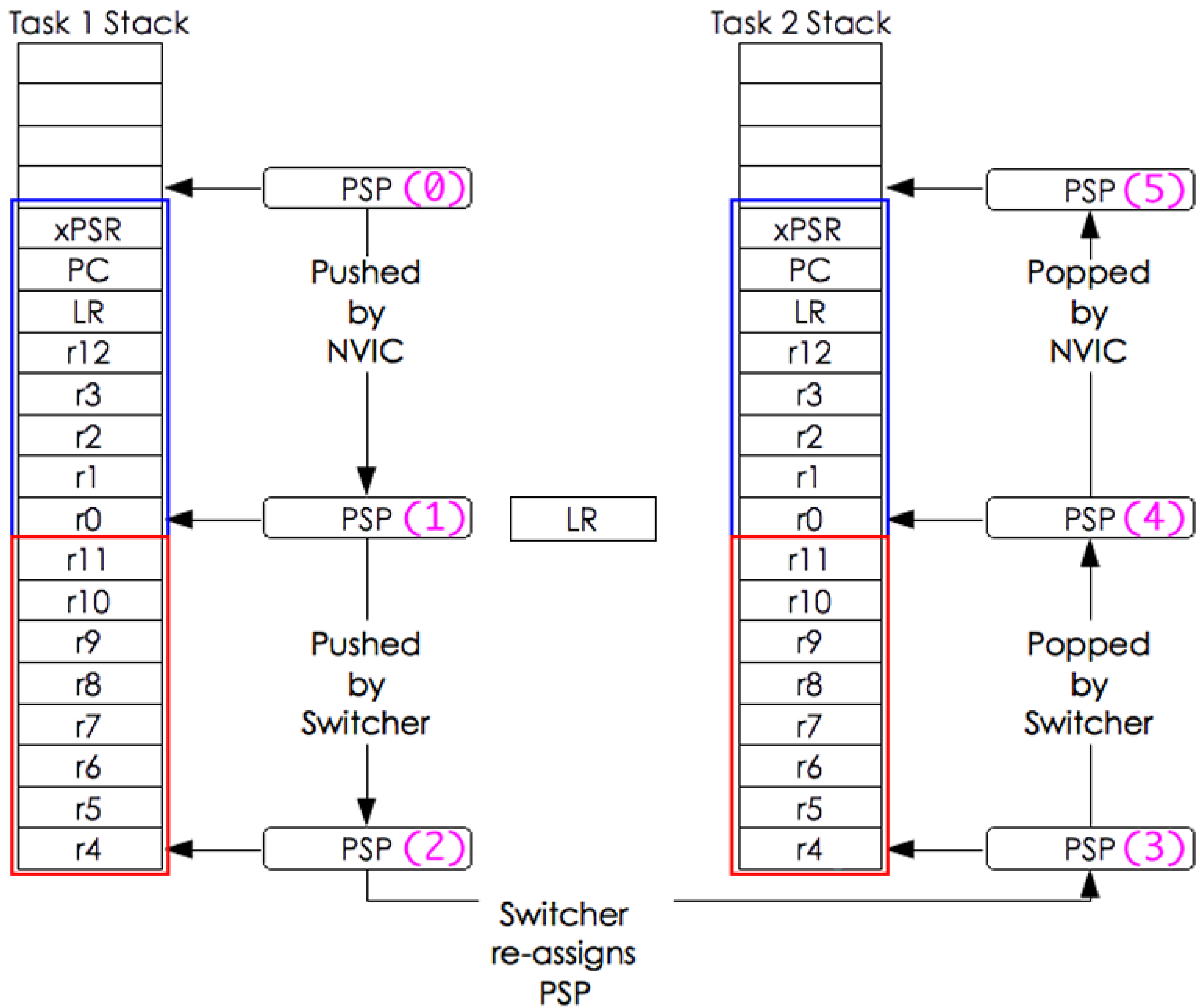


Figure 9: Explicit thread switch

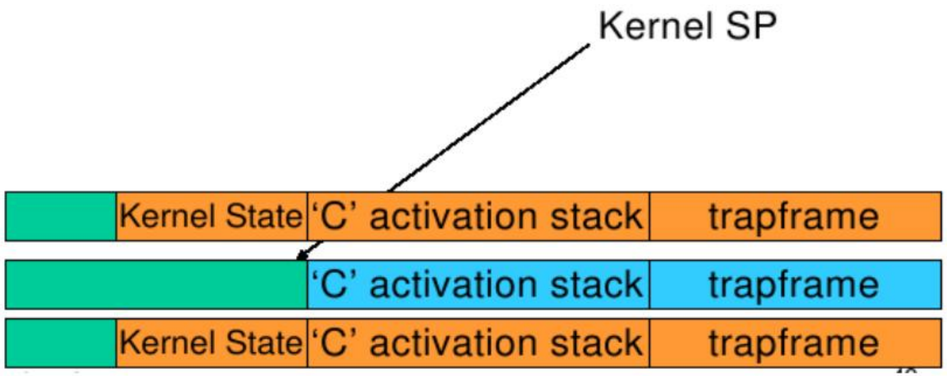
Thread Switch in OS – ARM Processor



Dig Deep into the context switching

Dig deep into the context switching: Question: Where do we put the stackframe? In kernel stack or process stack!!

- Does it resides in put into the kernel
  - A few OS description shows that it is in kernel



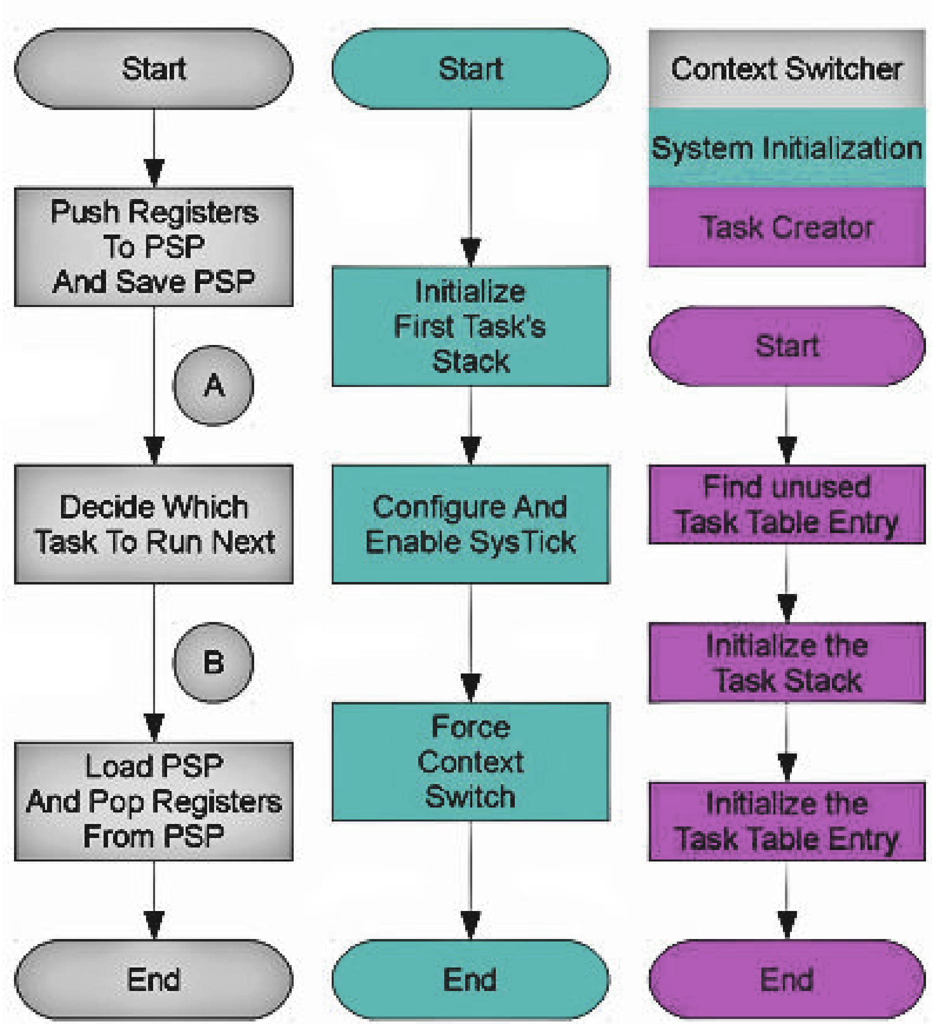
  - Some other shows that the stackframe in processes' stack
    - Shows in the previous slide
- Trapframe and stackframe are developed from the same idea
- We go with stackframe in the process stack
- However the OS needs vital information on process and thread to create, pause, resume and terminate a process

Stack Frame in Process/Thread Stack

Stack Frame in Process/Thread Stack

- Process header (PCB/TCB) in Scheduling Queue resides in kernel stack
  - Process Queue contains at least PSP of each of the process
  - any other information such as process/thread ID, state, statistics, own, and so on
  - Entry in the process queue deleted when a process terminates
  - New process/thread inserted into the queue
- Process stack contains stackframe or trapframe
  - Process/thread register and register values, xPSR and PC, resides in the process stack
  - Exception/Interrupt automatically (hardware) insert xPSR, PC LR and r12, r0-r3
  - If active FPU registers D0-D15 inserted into the process stack
  - OS scheduler (PendSV) insert other microprocessor-registers of and FPU

StackFrame in Process/Task Stack



StackFrame in Process/Task Stack

Example functions for task scheduling

```
typedef struct {
 void * sp; /*The task's current
        stack pointer*/
 int flags; /*Status flags
        includes activity
        status, parent task, etc*/
 } task_table_t;
int current_task;
task_table_t task_queue[MAX_TASKS];
```