

# Processes and Threads



# Learning Outcomes

- An understanding of fundamental concepts of processes and threads
- An understanding of the typical implementation strategies of processes and threads
  - Including an appreciation of the trade-offs between the implementation approaches
    - Kernel-threads versus user-level threads
- A detailed understanding of “context switching”



# Major Requirements of an Operating System

- 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



# Processes and Threads

- 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
  - Can be traced
    - list the sequence of instructions that execute
  - Belongs to a process



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

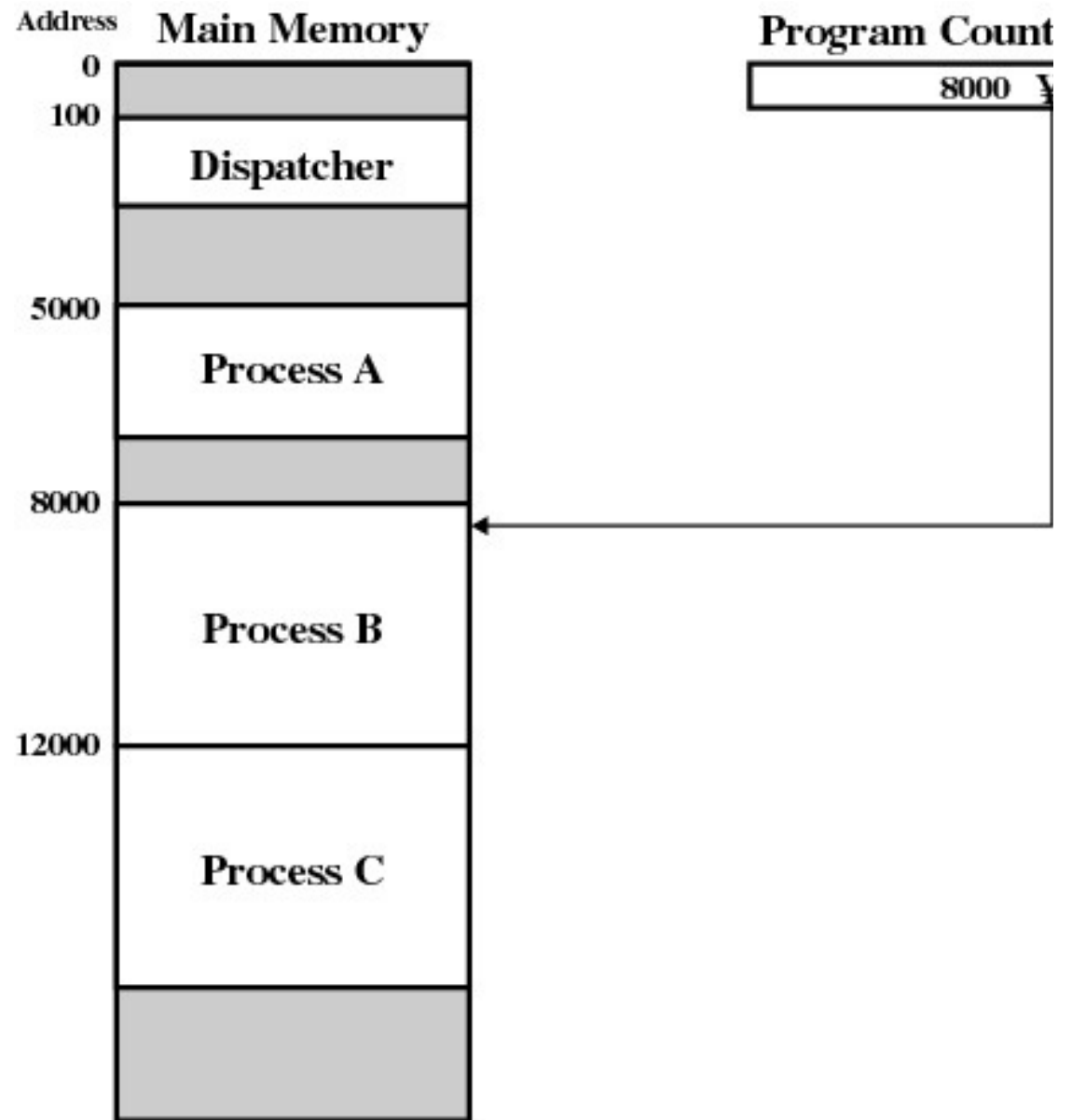


Figure 3.1 Snapshot of Example Execution (Figure 3 at Instruction Cycle 13

# Logical Execution Trace

5000  
5001  
5002  
5003  
5004  
5005  
5006  
5007  
5008  
5009  
5010  
5011

8000  
8001  
8002  
8003

12000  
12001  
12002  
12003  
12004  
12005  
12006  
12007  
12008  
12009  
12010  
12011

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A  
8000 = Starting address of program of Process B  
12000 = Starting address of program of Process C

**Figure 3.2** Traces of Processes of Figure 3.1

## Combined Traces

(Actual CPU  
Instructions)

What are the  
shaded sections?

1	5000		
2	5001		
3	5002		
4	5003		
5	5004		
6	5005		
-----Time out			
7	100		
8	101		
9	102		
10	103		
11	104		
12	105		
13	8000		
14	8001		
15	8002		
16	8003		
-----I/O request			
17	100		
18	101		
19	102		
20	103		
21	104		
22	105		
23	12000		
24	12001		
25	12002		
26	12003		
27	12004		
28	12005		
-----Time out			
29	100		
30	101		
31	102		
32	103		
33	104		
34	105		
35	5006		
36	5007		
37	5008		
38	5009		
39	5010		
40	5011		
-----Time out			
41	100		
42	101		
43	102		
44	103		
45	104		
46	105		
47	12006		
48	12007		
49	12008		
50	12009		
51	12010		
52	12011		
-----Time out			

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;

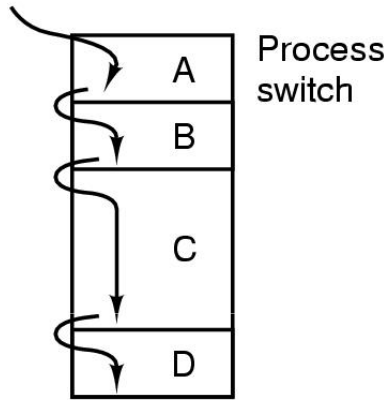
first and third columns count instruction cycles;

second and fourth columns show address of instruction being executed

**Figure 3.3 Combined Trace of Processes of Figure 3.1**

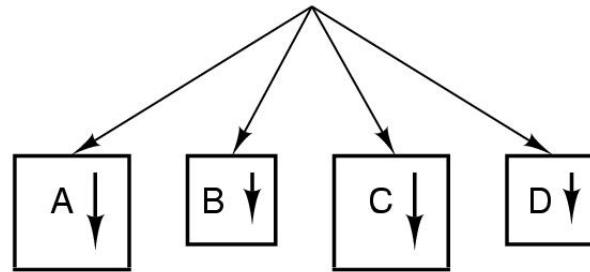
# Summary: The Process Model

One program counter

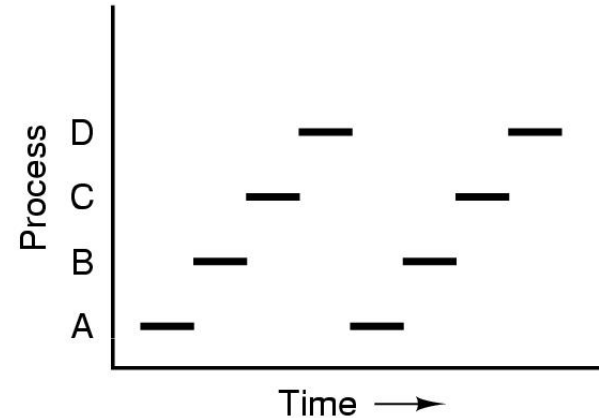


(a)

Four program counters



(b)

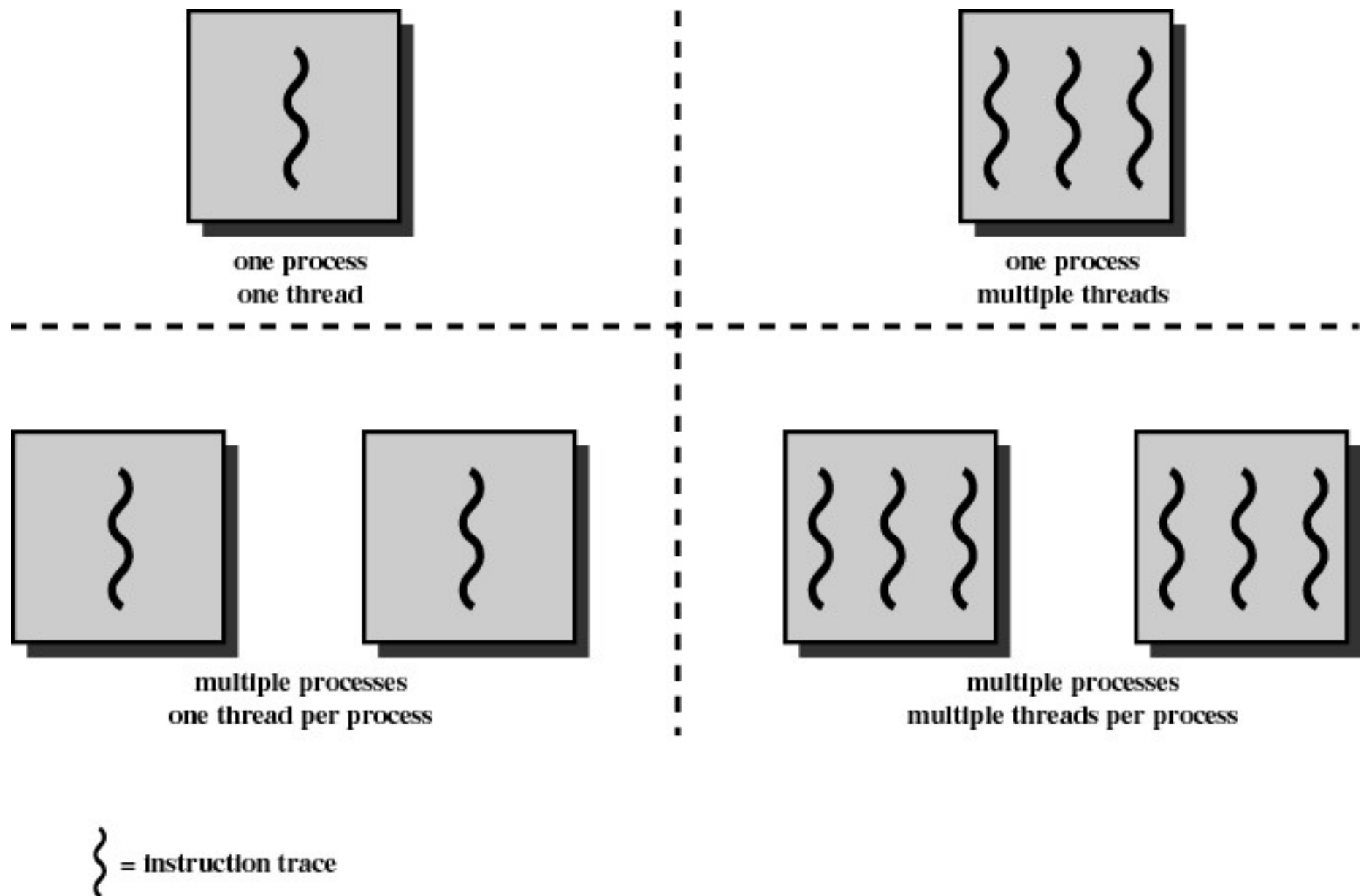


(c)

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







**Figure 4.1** Threads and Processes [ANDE97]

# Process and thread models of selected OSes

- Single process, single thread
  - MSDOS
- Single process, multiple threads
  - OS/161 as distributed
- Multiple processes, single thread
  - Traditional unix
- Multiple processes, multiple threads
  - Modern Unix (Linux, Solaris), Windows 2000

Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)



# Process Creation

## Principal events that cause 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/ssh 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.



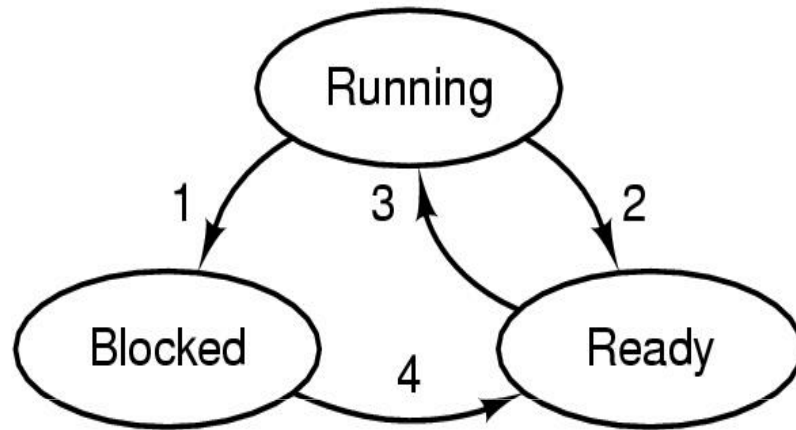
# Process Termination

Conditions which terminate processes

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



# Process/Thread States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process/thread states
  - running
  - blocked
  - ready
- Transitions between states shown



# Some Transition Causing Events

## Running ▷ Ready

- Voluntary `yield()`
- End of timeslice

## Running ▷ Blocked

- Waiting for input
  - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available

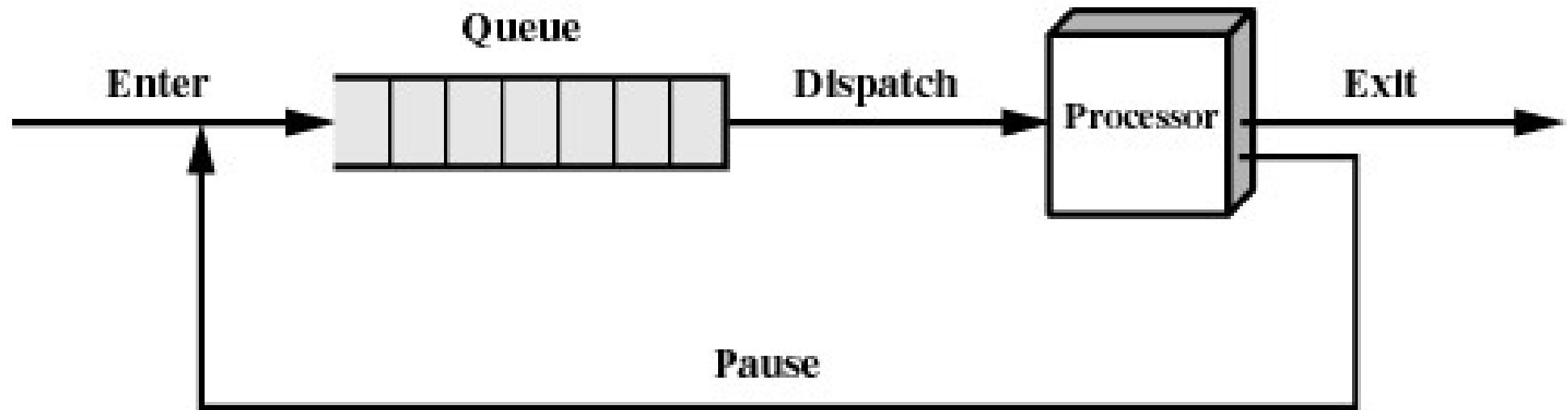


# Dispatcher

- 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



# The Ready Queue



(b) Queuing diagram

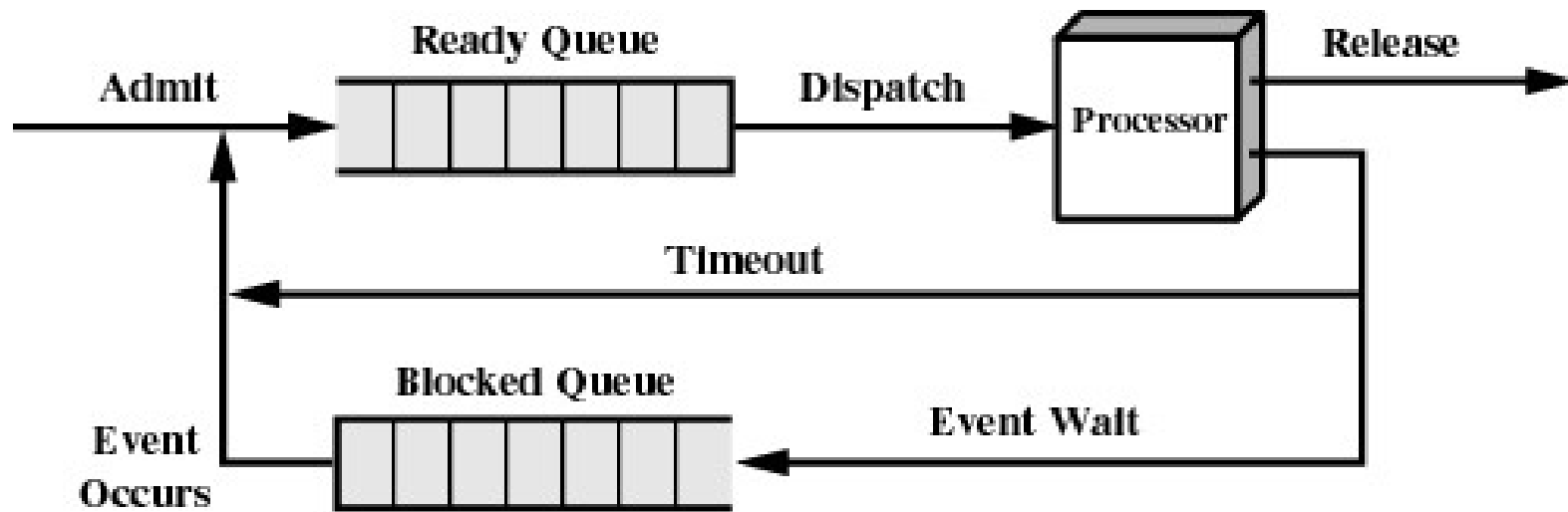


# What about blocked processes?

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

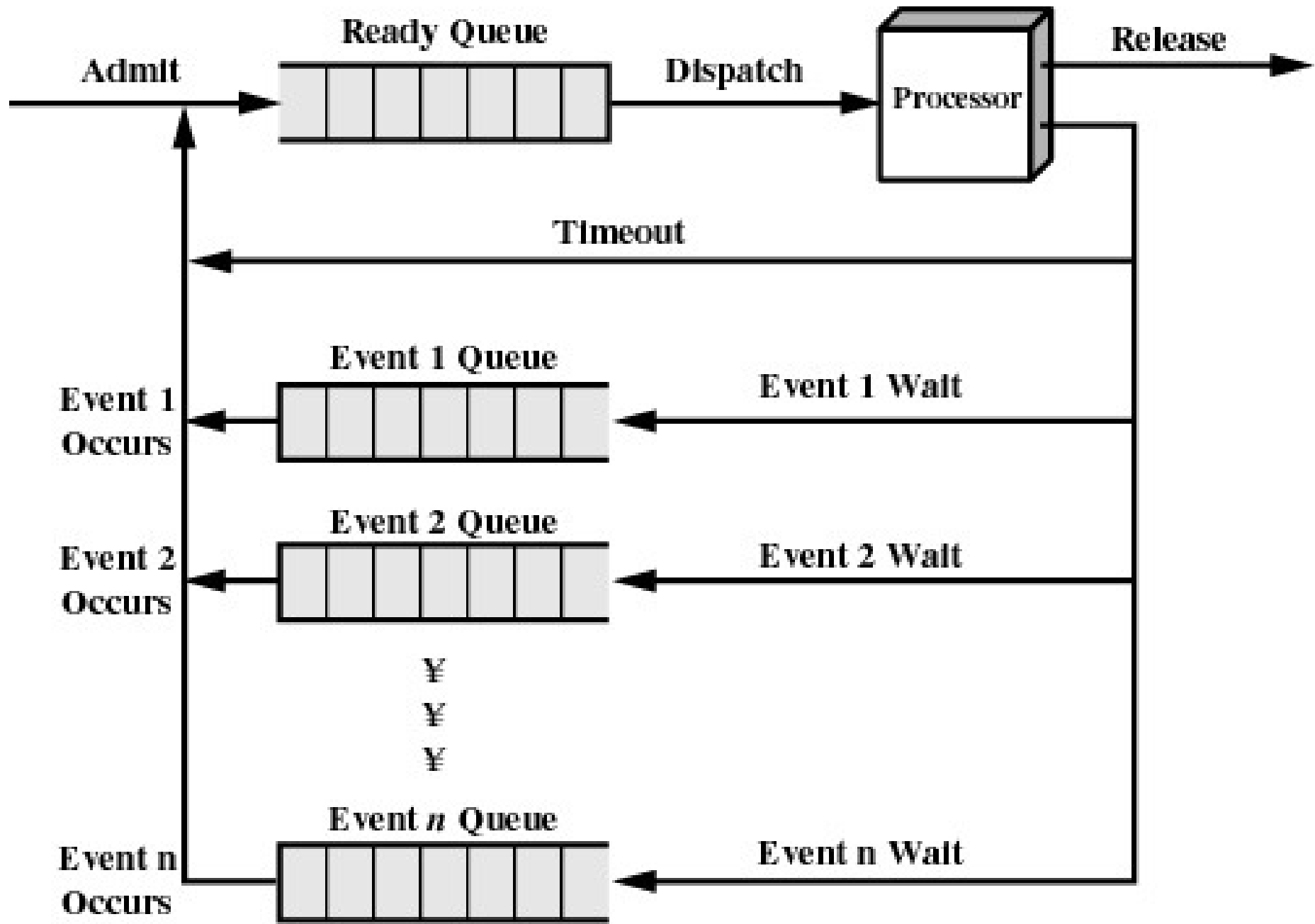


# Using Two Queues



(a) Single blocked queue





(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 *trapframe* in OS/161
  - Reality is much more complex (hashing, chaining, allocation bitmaps,...)

P7
P6
P5
P4
P3
P2
P1
P0



# Implementation of Processes

<b>Process management</b>	<b>Memory management</b>	<b>File management</b>
Registers Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

## Example fields of a process table entry



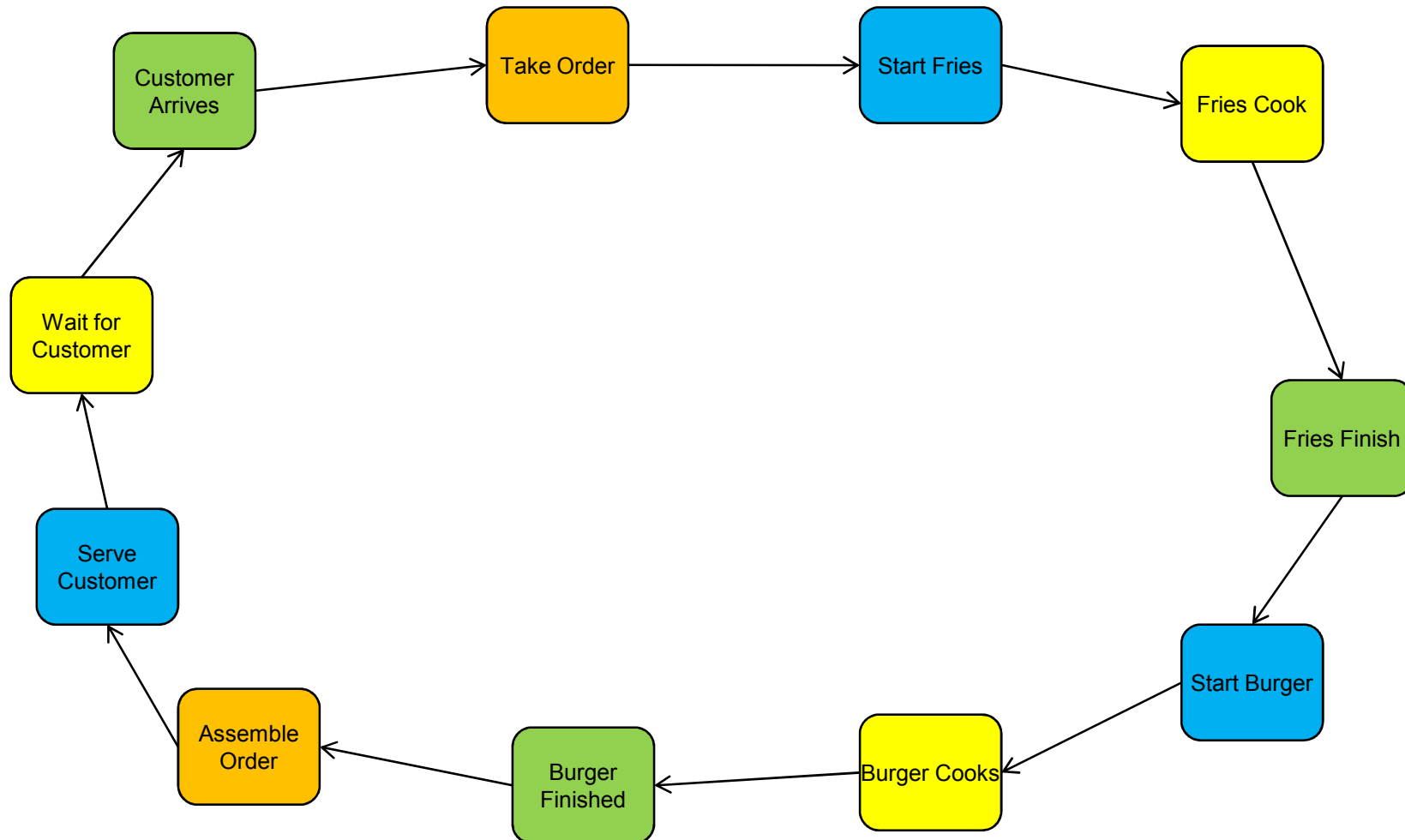
# Threads Analogy



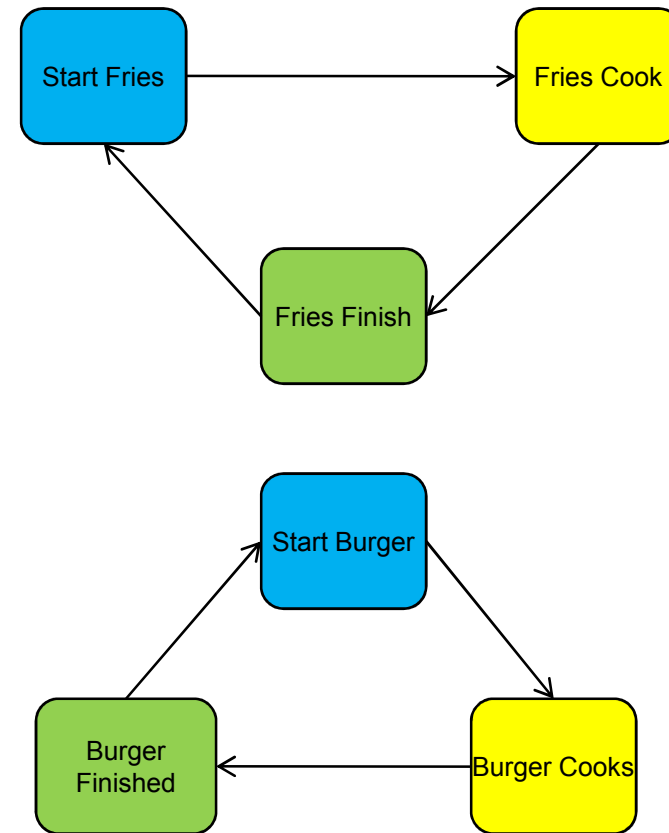
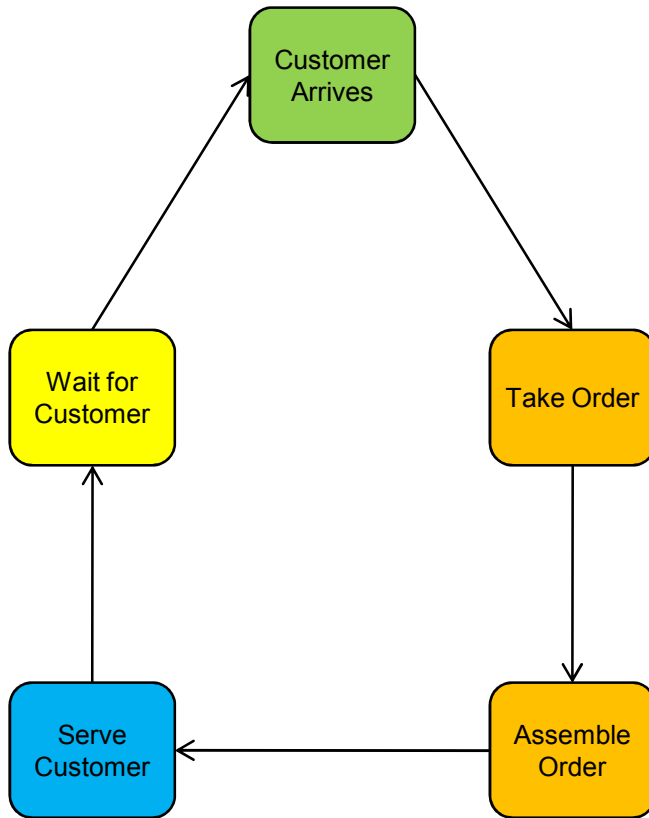
## The Hamburger Restaurant



# Single-Threaded Restaurant



# Multithreaded Restaurant

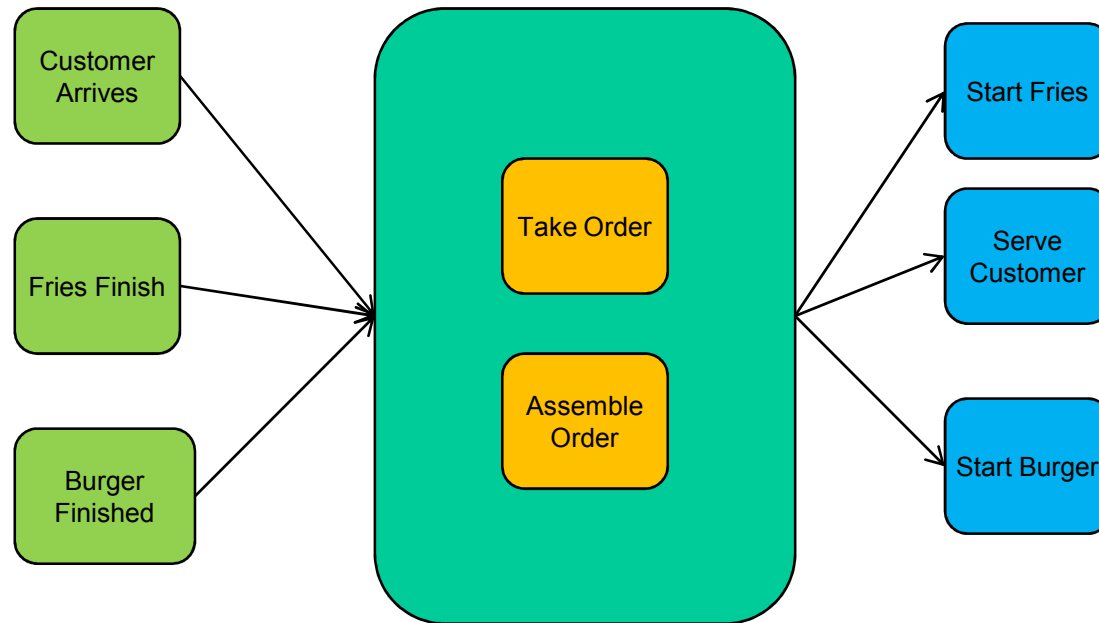




# Finite-State Machine Model

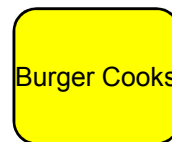
(Event-based model)

Input  
Events



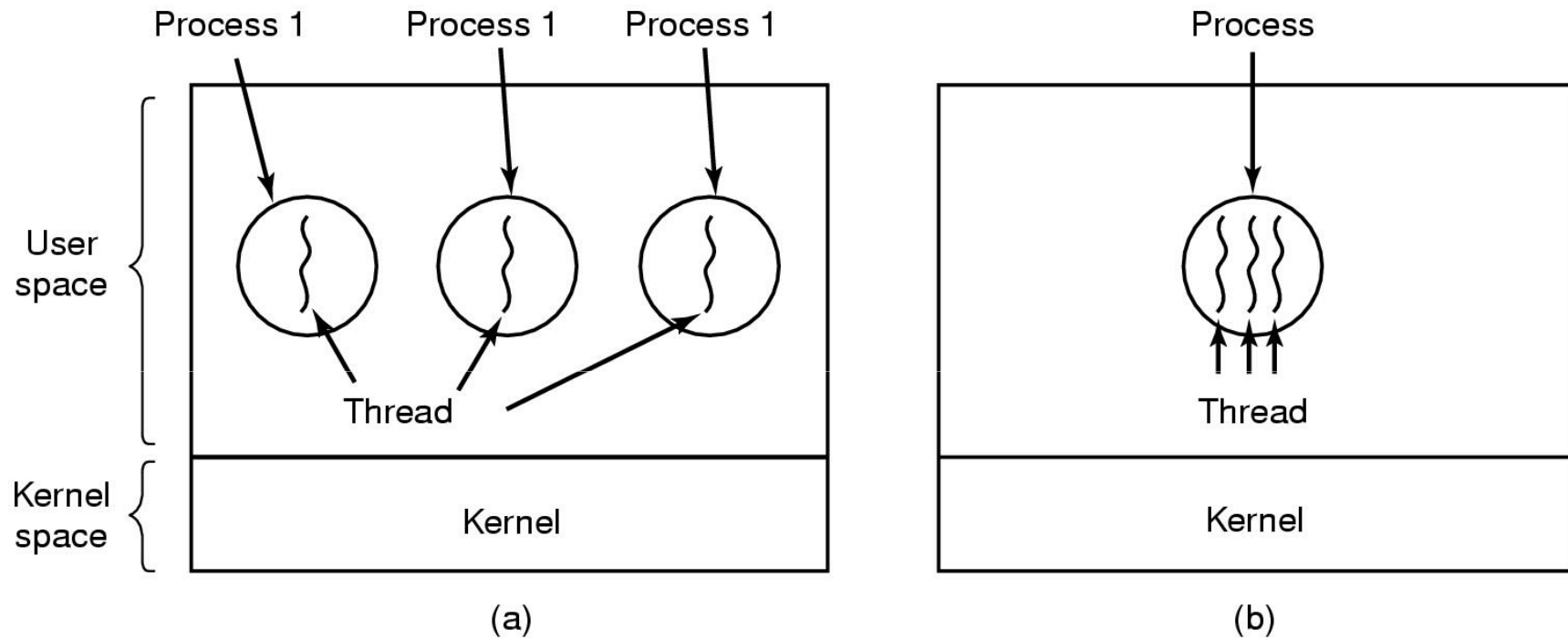
Non-  
Blocking  
actions

External  
activities



# Threads

## The Thread Model



(a) Three processes each with one thread

(b) One process with three threads



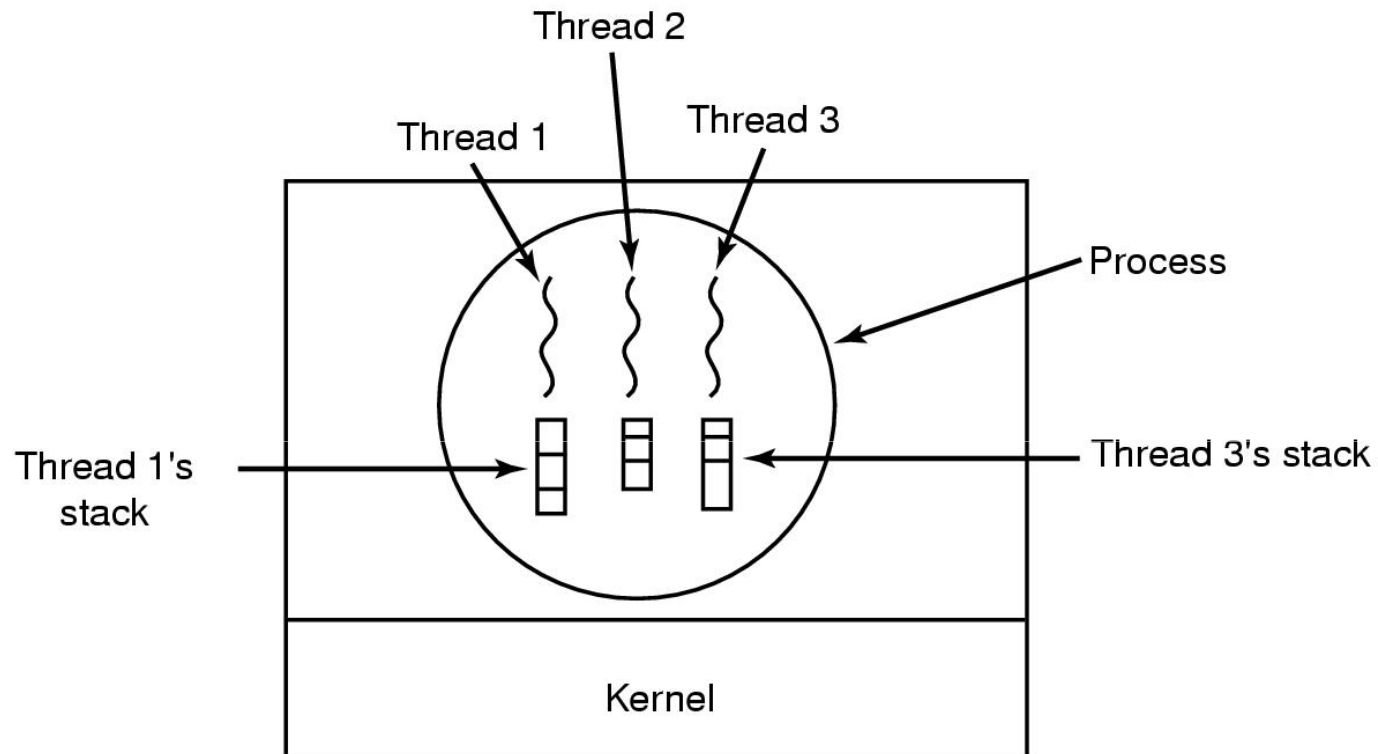
# The Thread Model

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

- Items shared by all threads in a process
- Items private to each thread



# The Thread Model



Each thread has its own stack

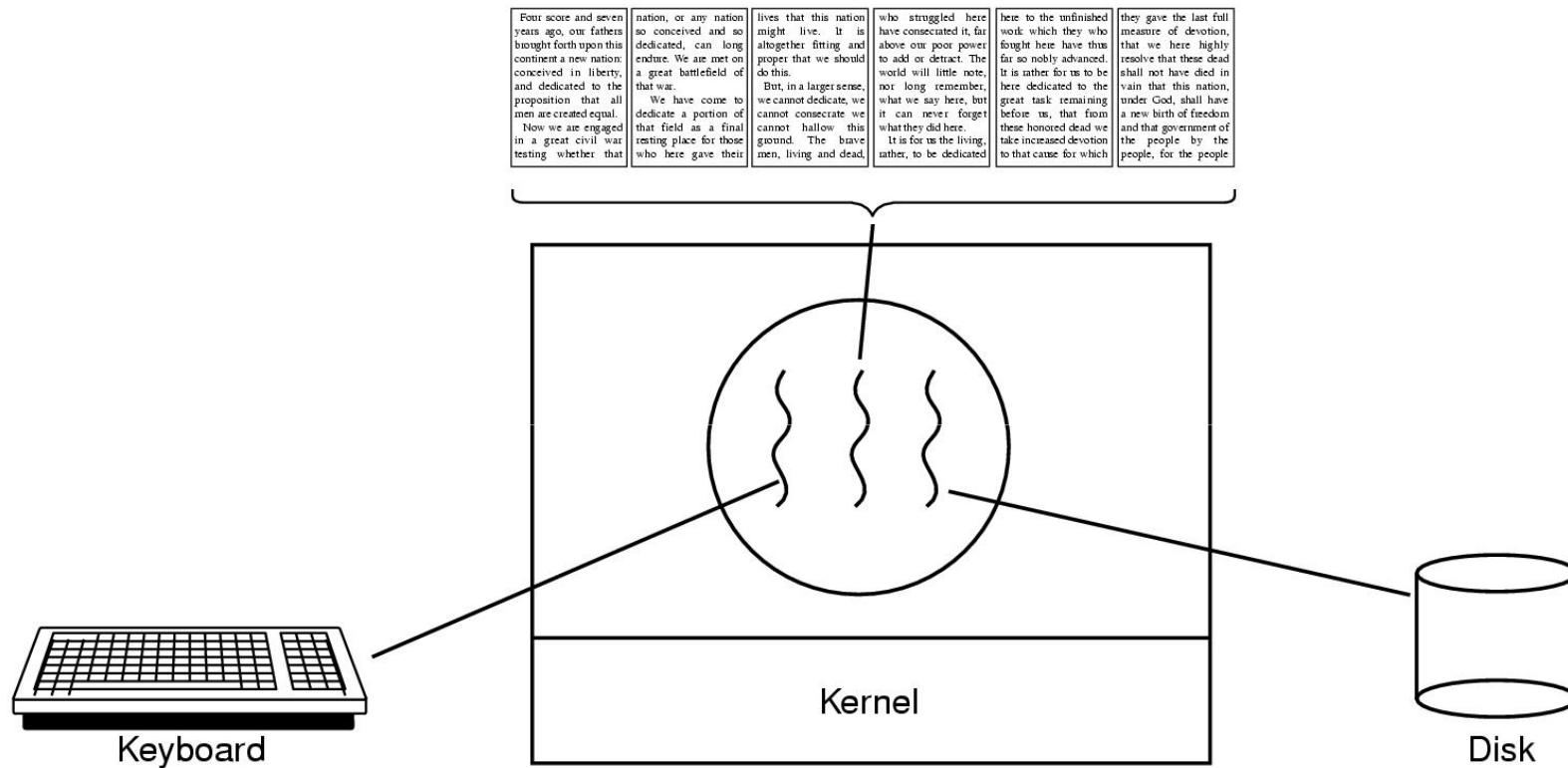


# Thread Model

- 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)



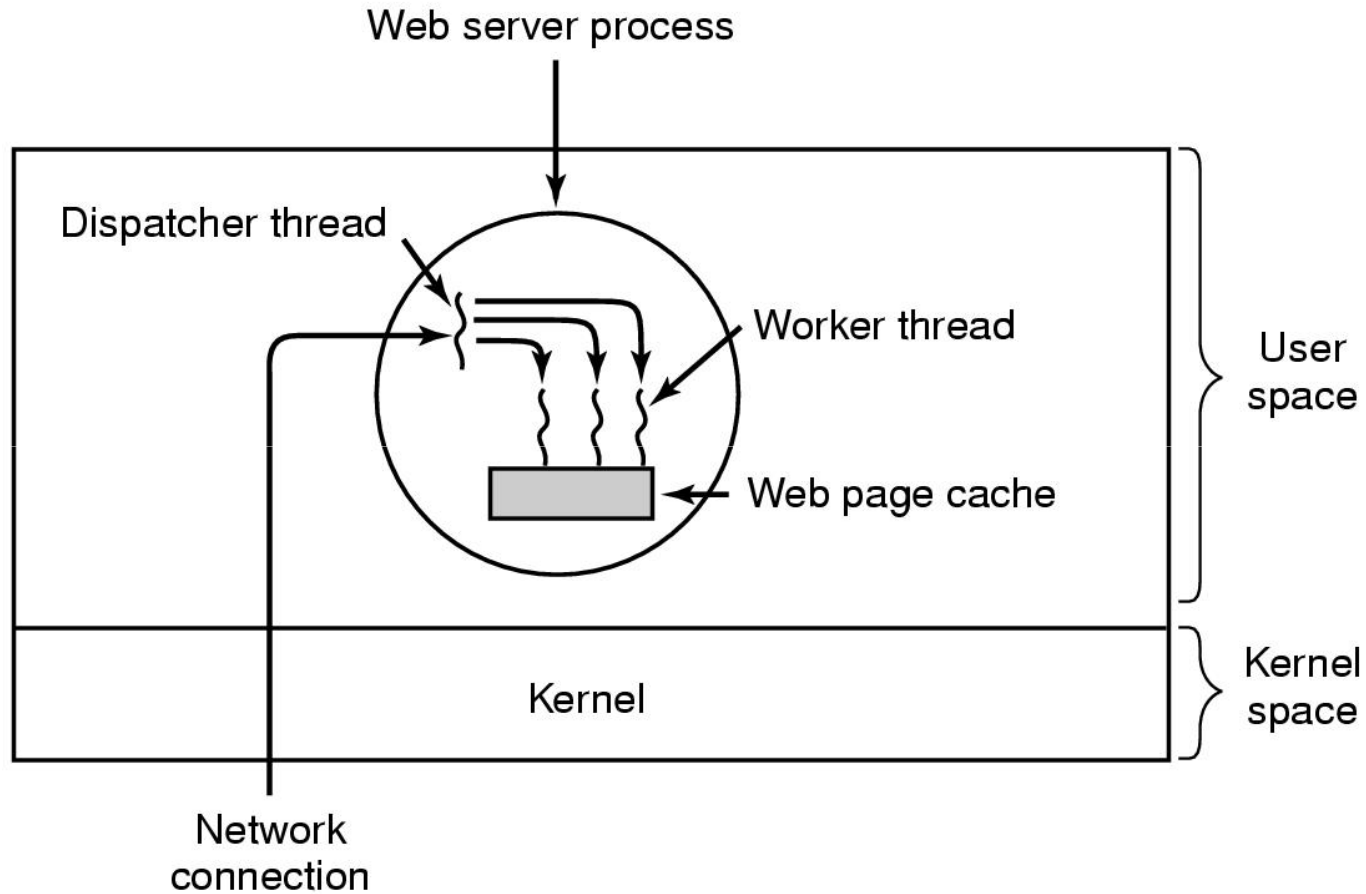
# Thread Usage



A word processor with three threads



# Thread Usage



## A multithreaded Web server



# Thread Usage

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

(a)

```
while (TRUE) {  
    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);  
}
```

(b)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread





# Thread Usage

<b>Model</b>	<b>Characteristics</b>
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

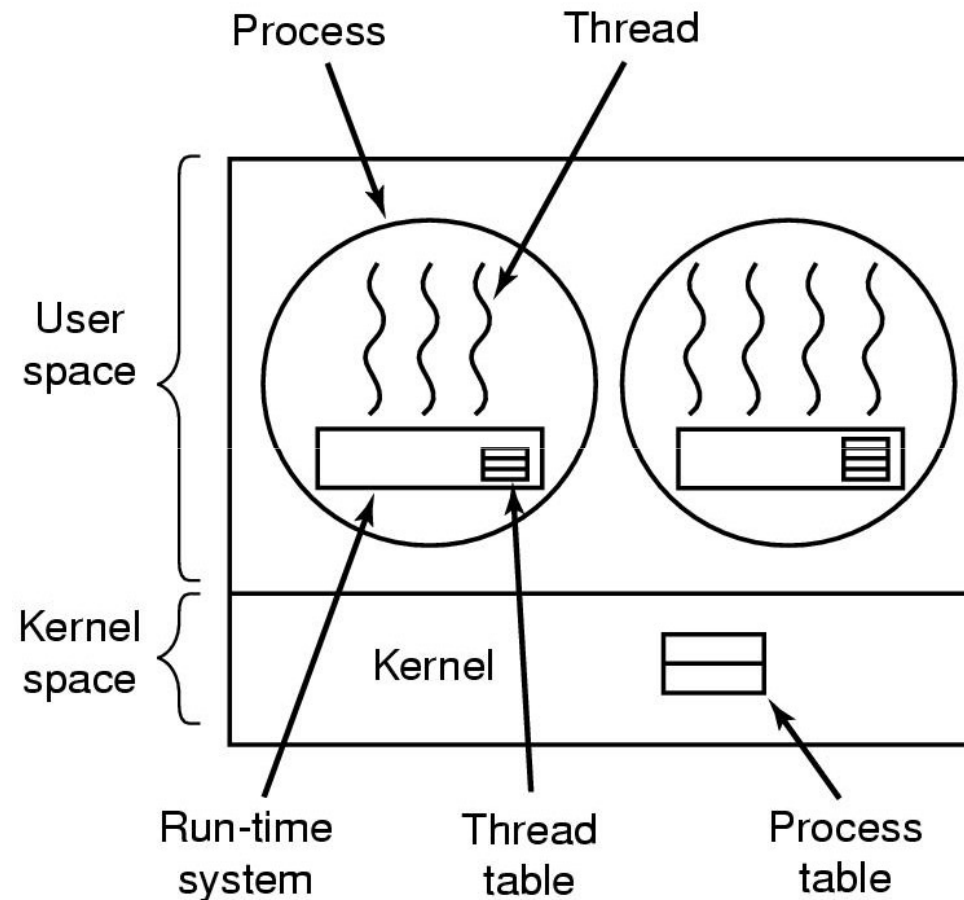


# Summarising “Why Threads?”

- 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



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 saved and the dispatcher switches to another ready thread
  - Thread management (create, exit, yield, wait) are implemented in a runtime support library



# User-Level Threads

- Pros
  - Thread management and switching at user level is much faster than doing it in kernel level
    - No need to trap (take syscall exception) 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 Threads

- Cons
  - Threads have to yield() manually (no timer interrupt delivery to user-level)
    - **Co-operative multithreading**
      - A single poorly design/implemented thread can monopolise the available CPU time
      - There 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)

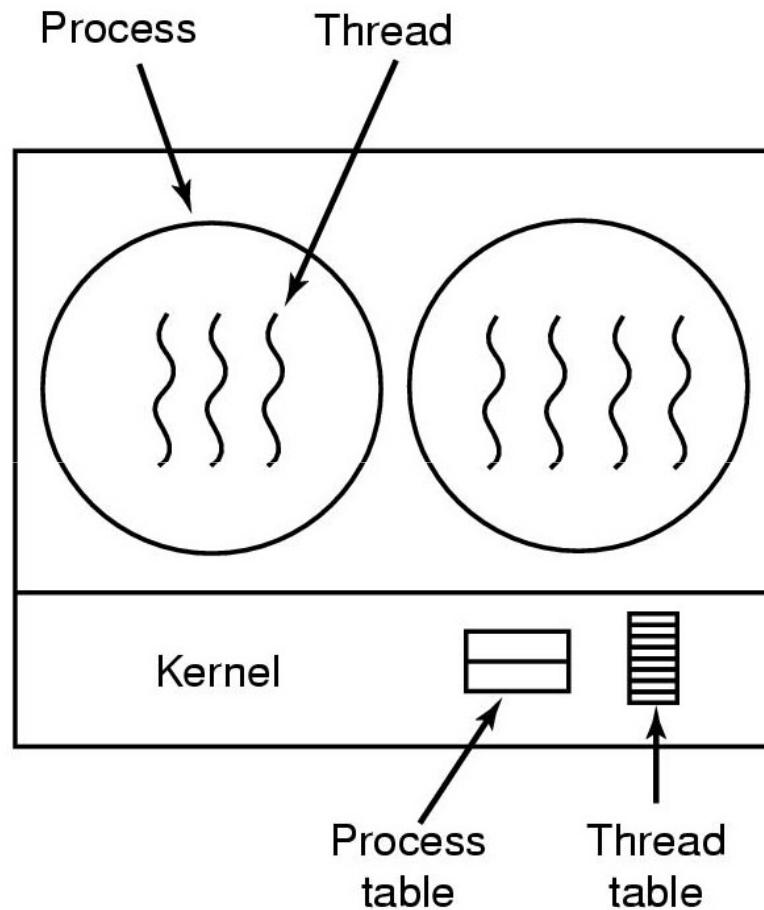


# User-Level Threads

- 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 test if read system call would block
        - » `select()` then `read()`
        - » Only call `read()` if it won't block
        - » Otherwise schedule another thread
      - Wrapper requires 2 system calls instead of 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



A threads package managed by the kernel





# Kernel Threads

- Threads are implemented in the kernel
  - TCBs are stored in the kernel
    - A subset of information in a 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 calls
    - E.g. create, wait, exit

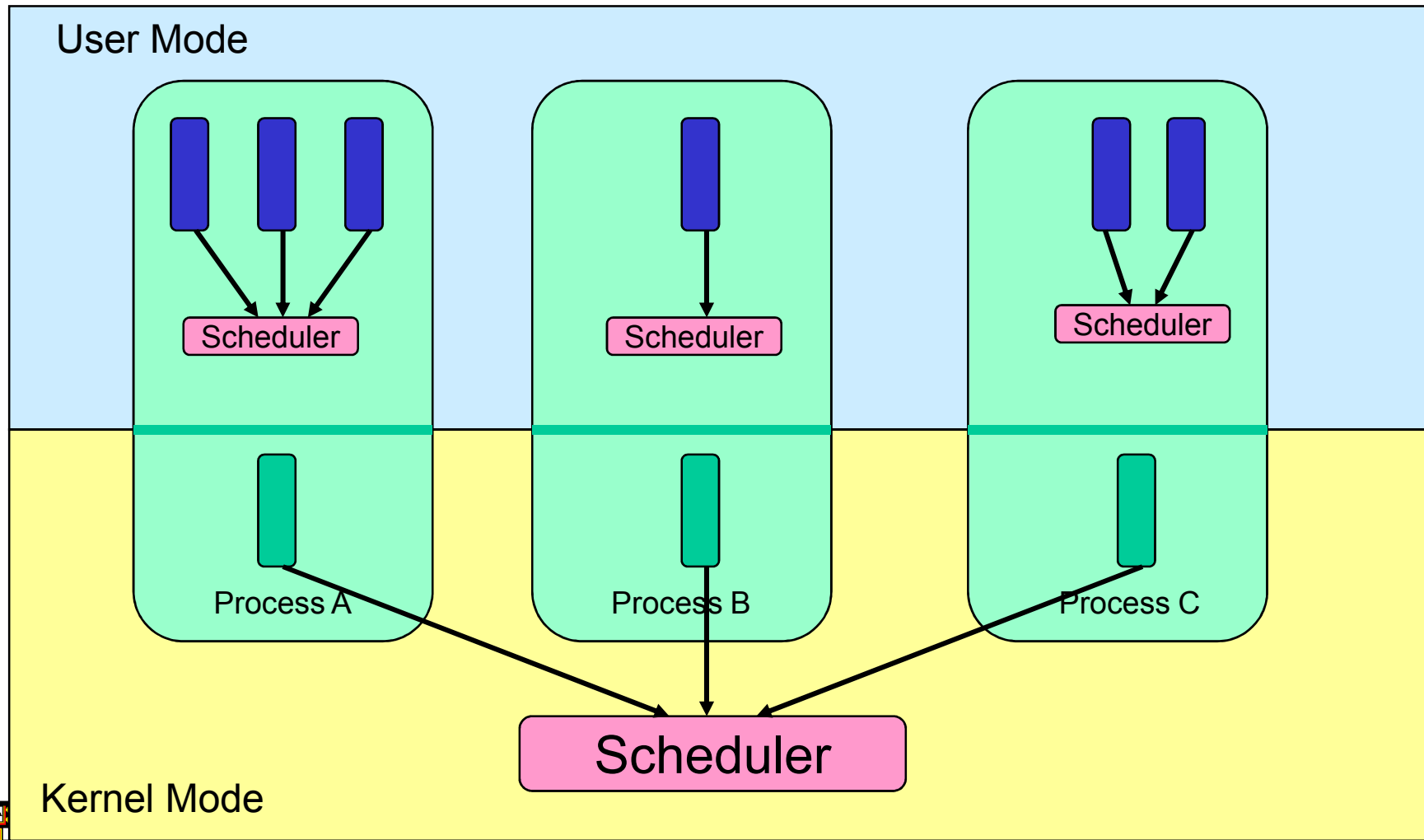


# Kernel Threads

- Cons
  - Thread creation and destruction, and blocking and unblocking threads requires 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 multiprocessor



# User-level Threads

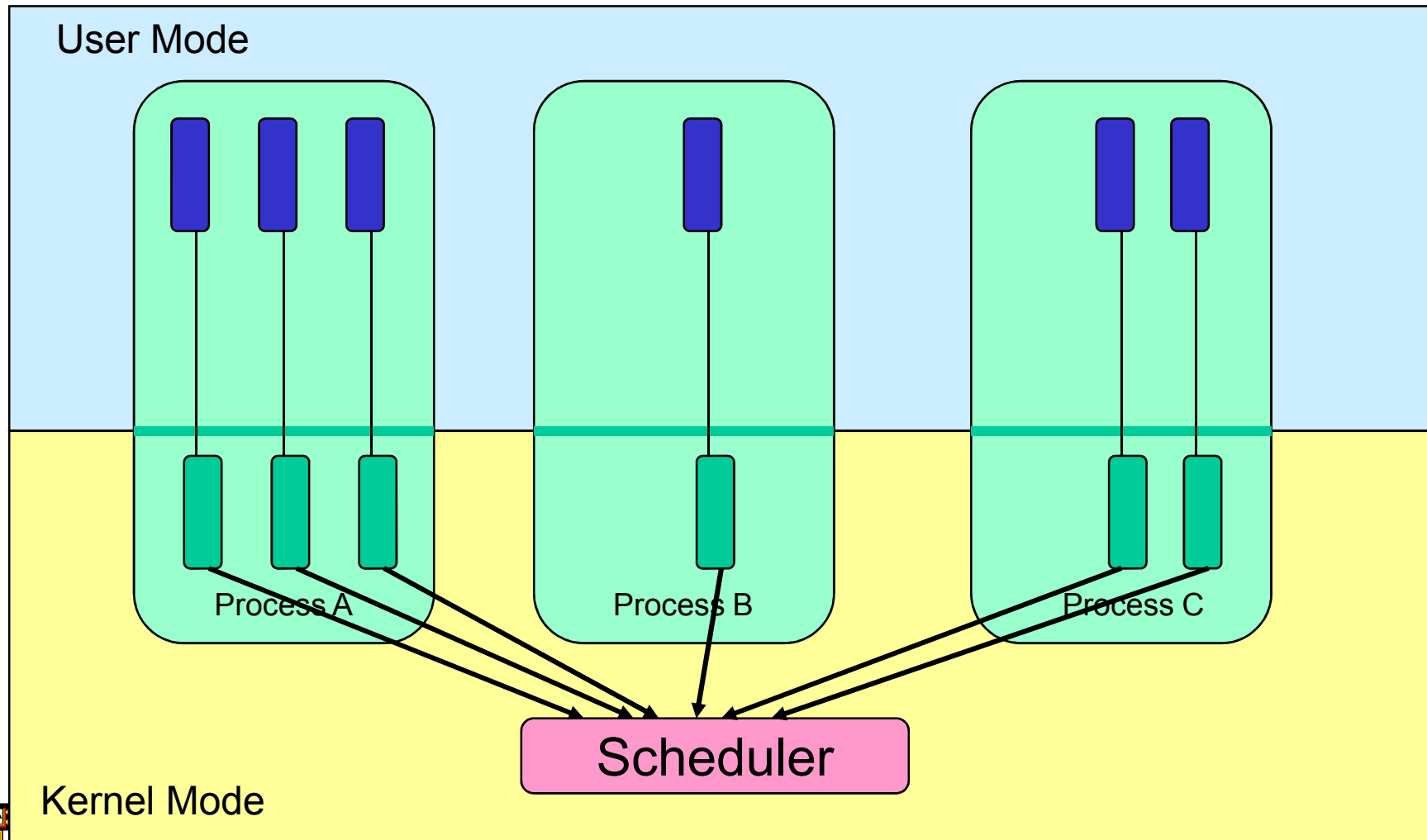


# User-level Threads

- ✓ Fast thread management (creation, deletion, switching, synchronisation...)
- ✗ Blocking blocks all threads in a process
  - Syscalls
  - Page faults
- ✗ No thread-level parallelism on multiprocessor



# Kernel-Level Threads



# Kernel-level Threads

- ✘ Slow thread management (creation, deletion, switching, synchronisation...)
  - System calls
- ✓ Blocking blocks only the appropriate thread in a process
- ✓ Thread-level parallelism on multiprocessor



# Multiprogramming Implementation

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

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



# Thread Switch

- A switch between threads can 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 interrupt
    - Triggering a dispatch is the main purpose of the *timer interrupt*

A thread switch can happen between any two instructions

Note instructions do not equal program statements



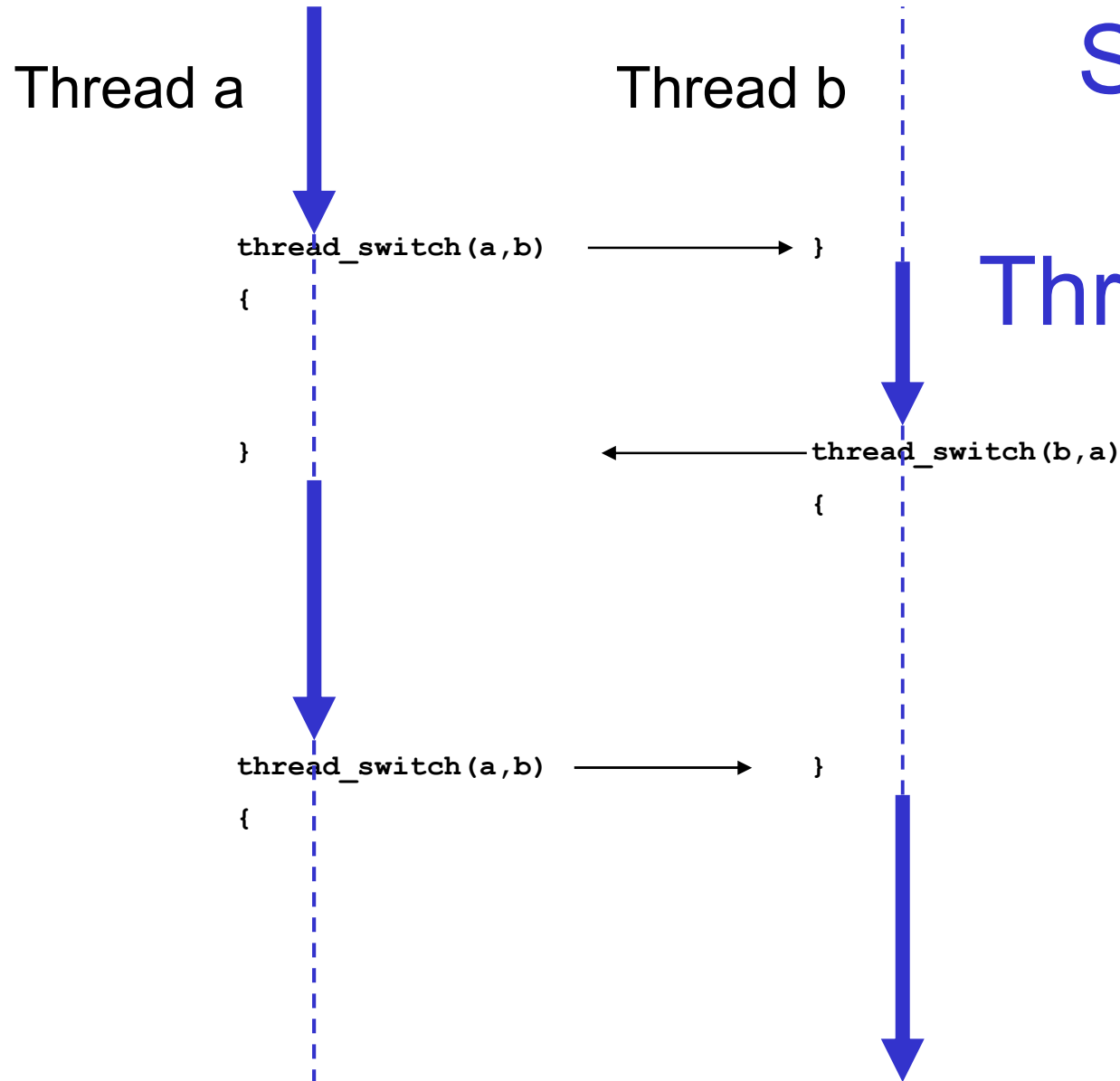


# Context Switch

- Thread switch must be *transparent* for threads
    - When dispatched again, thread should not notice that something else was running in the meantime (except for elapsed 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



# Example Context Switch

- Running in user mode, SP points to user-level activation stack

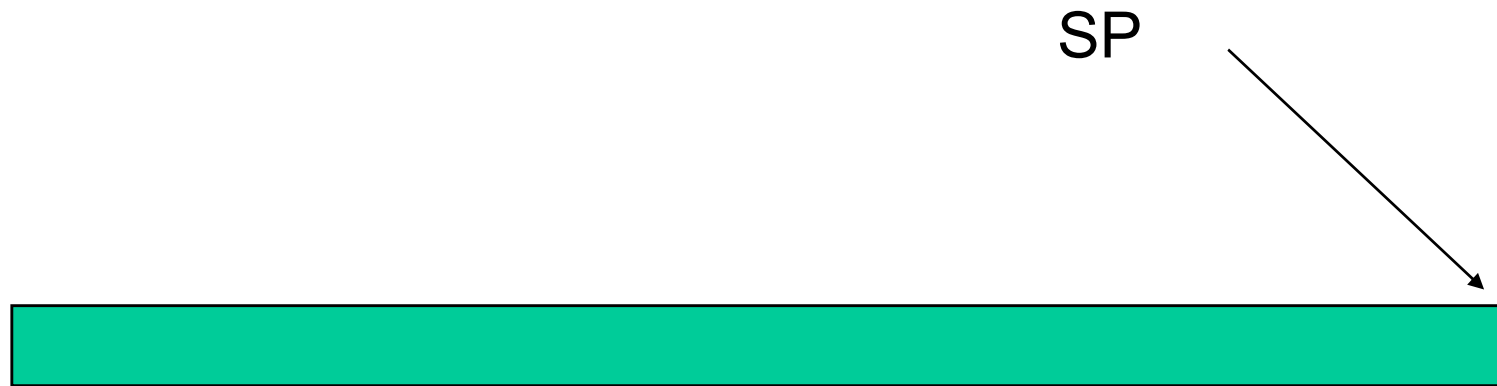
Representation of  
Kernel Stack  
(Memory)

SP



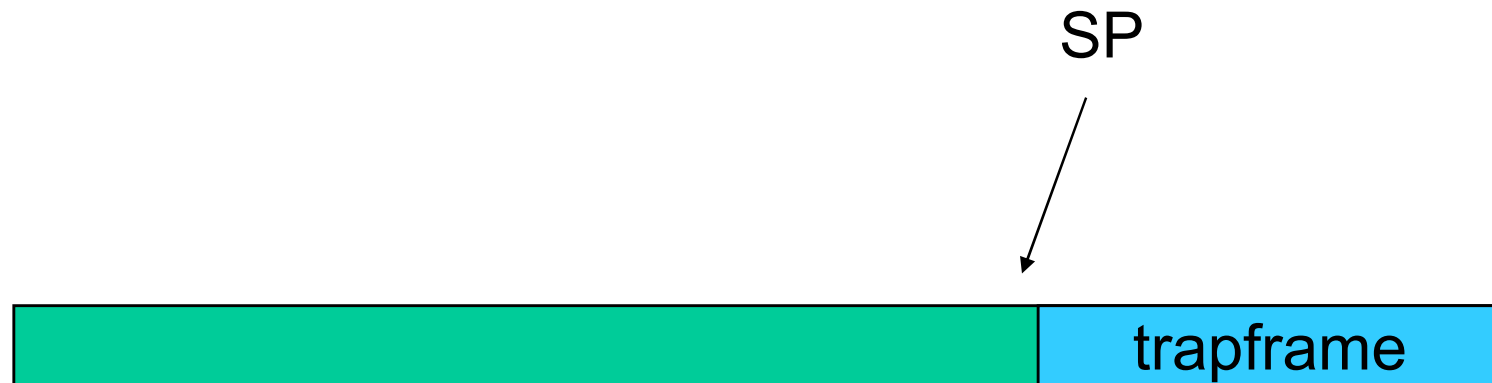
# Example Context Switch

- Take an exception, syscall, or interrupt, and we switch to the kernel stack



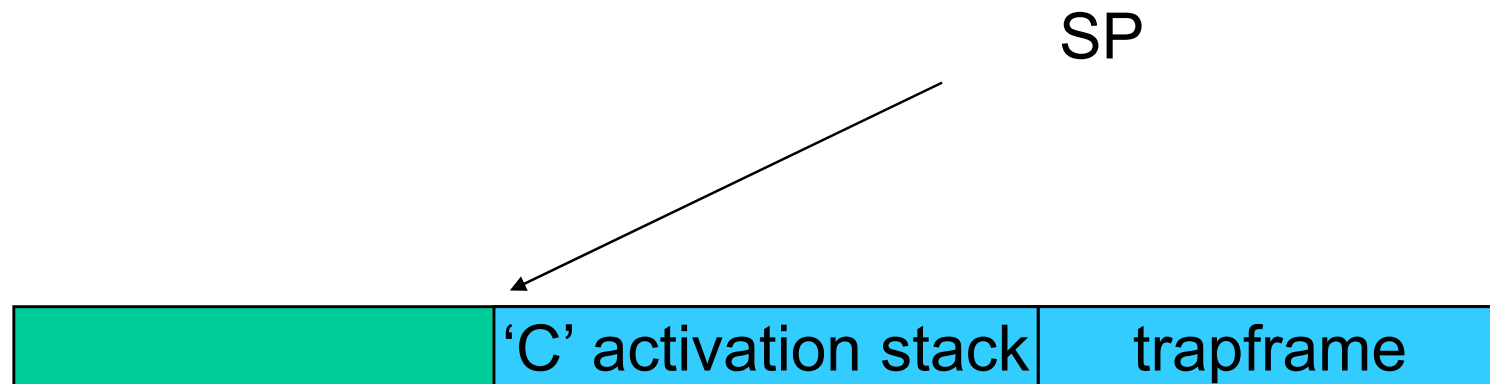
# Example Context Switch

- We push a *trapframe* on the stack
  - Also called *exception frame*, *user-level context*....
  - Includes the user-level PC and SP



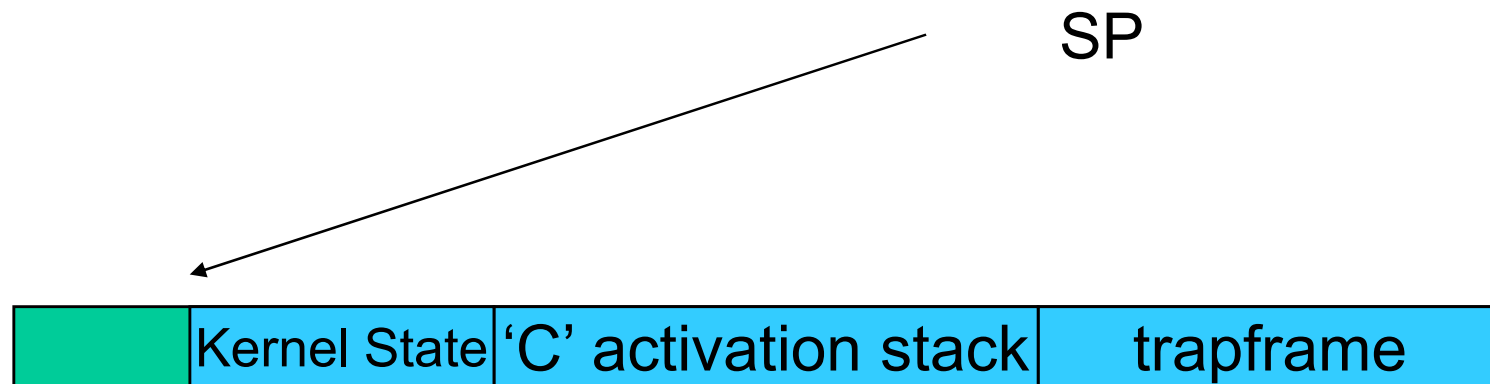
# Example Context Switch

- Call 'C' code to process syscall, exception, or interrupt
  - Results in a 'C' activation stack building up



# Example Context Switch

- The kernel decides to perform a context switch
  - It chooses a target thread (or process)
  - It pushes remaining kernel context onto the stack

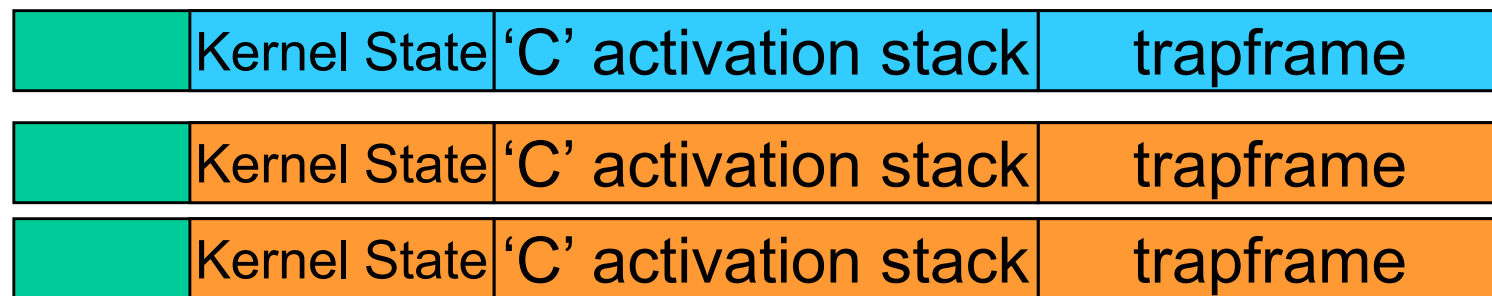


# Example Context Switch

- Any other existing thread must
  - be in kernel mode (on a uni processor),
  - and have a similar stack layout to the stack we are currently using

Kernel  
stacks of  
other  
threads

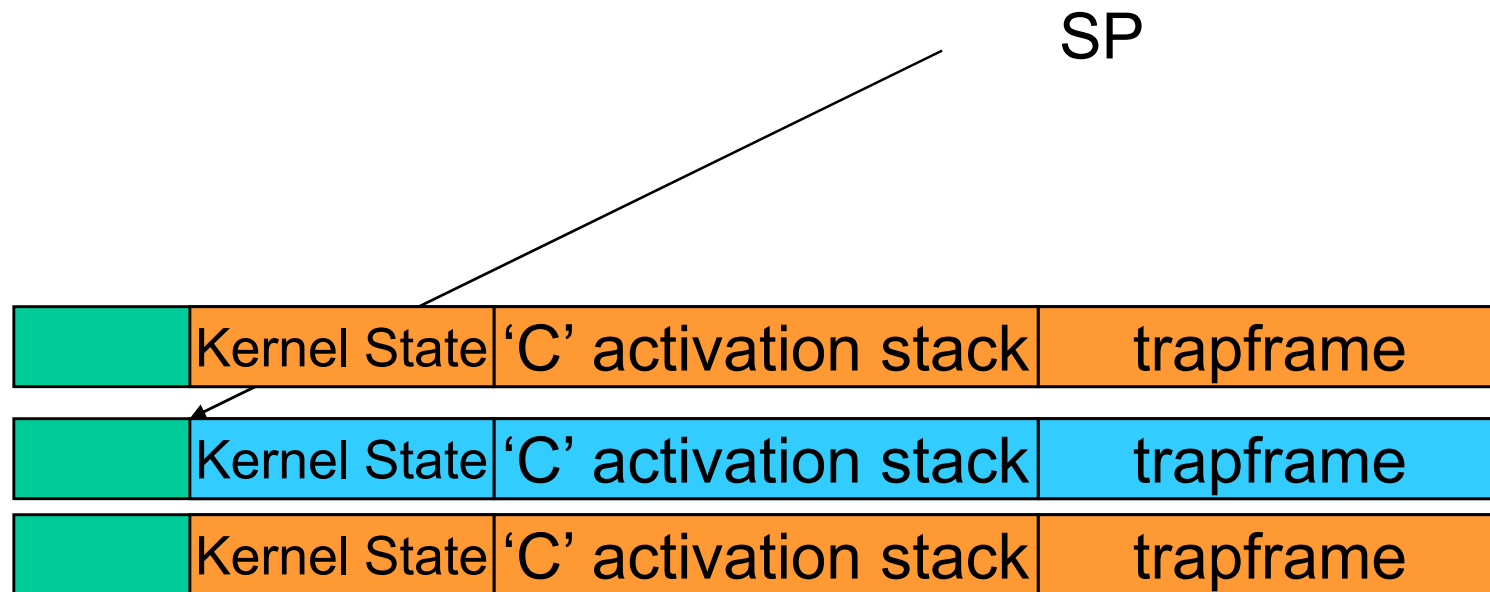
SP





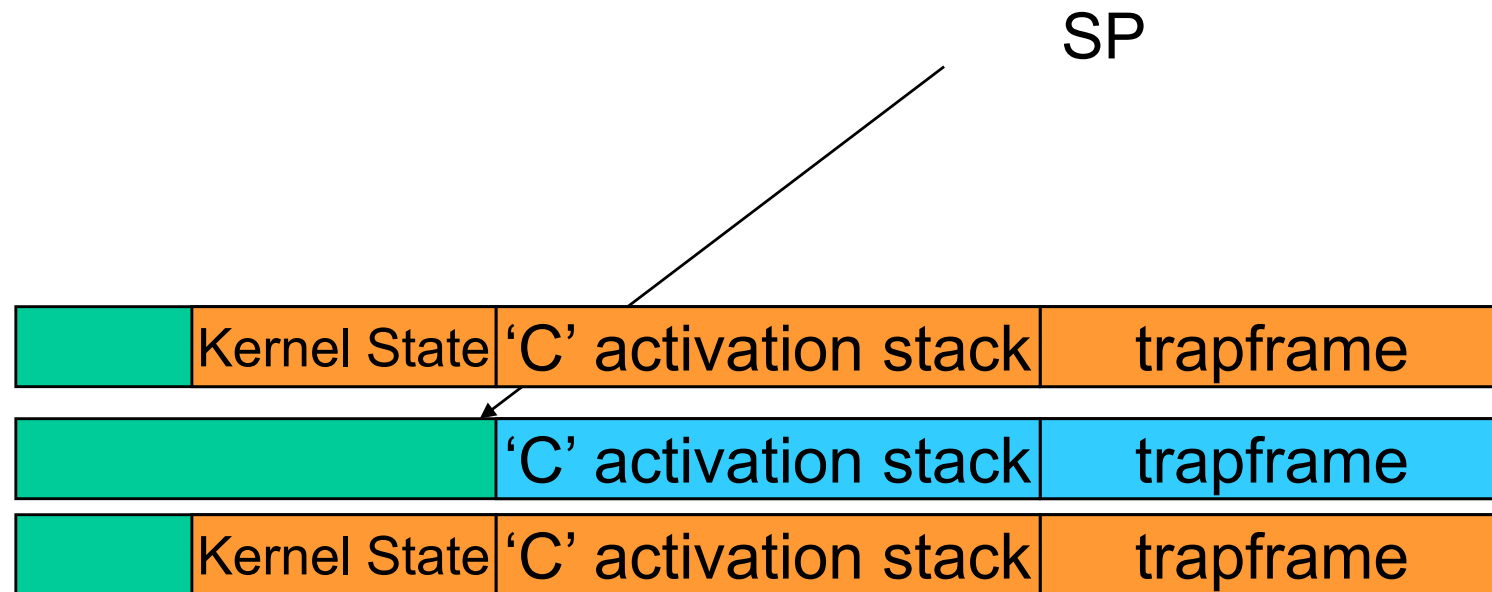
# Example Context Switch

- We save the current SP in the PCB (or TCB), and load the SP of the target thread.
  - Thus we have *switched contexts*



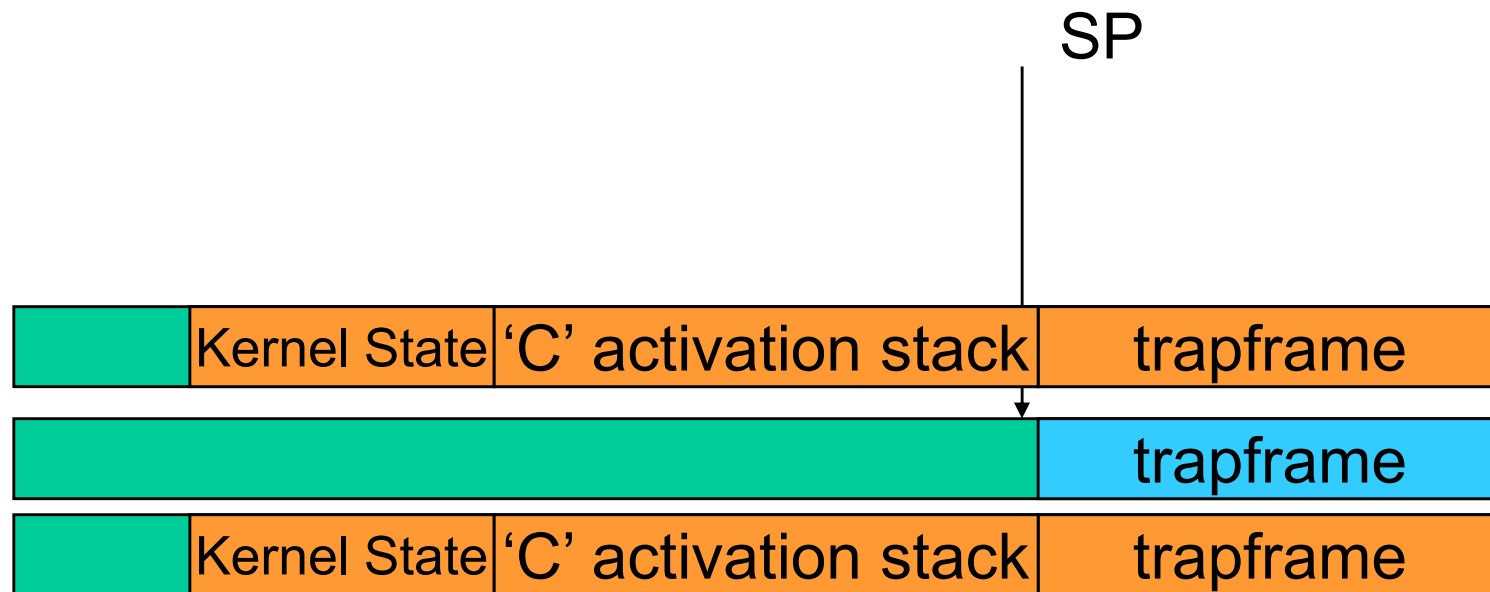
# Example Context Switch

- Load the target thread's previous context, and return to C



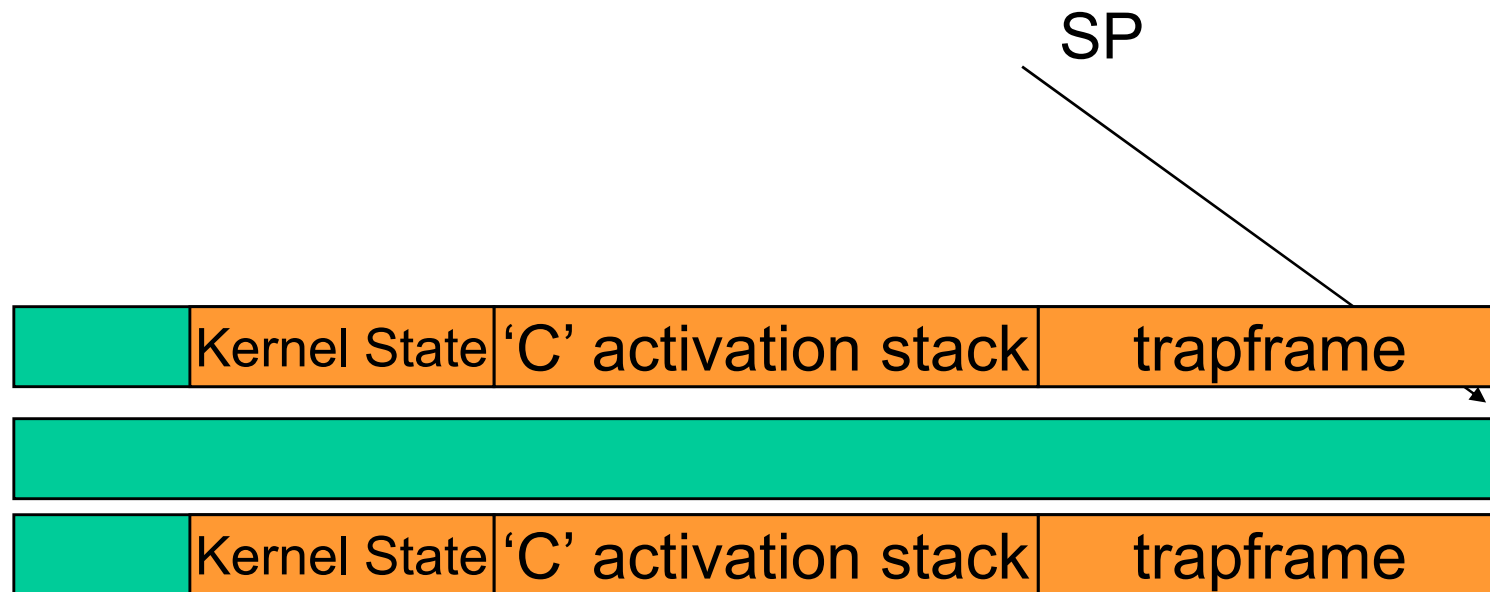
# Example Context Switch

- The C continues and (in this example) returns to user mode.



# Example Context Switch

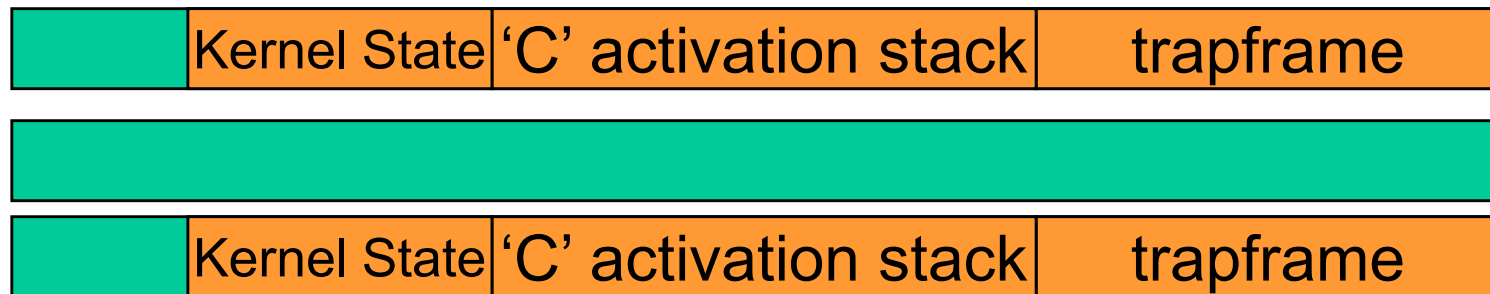
- The user-level context is restored



# Example Context Switch

- The user-level SP is restored

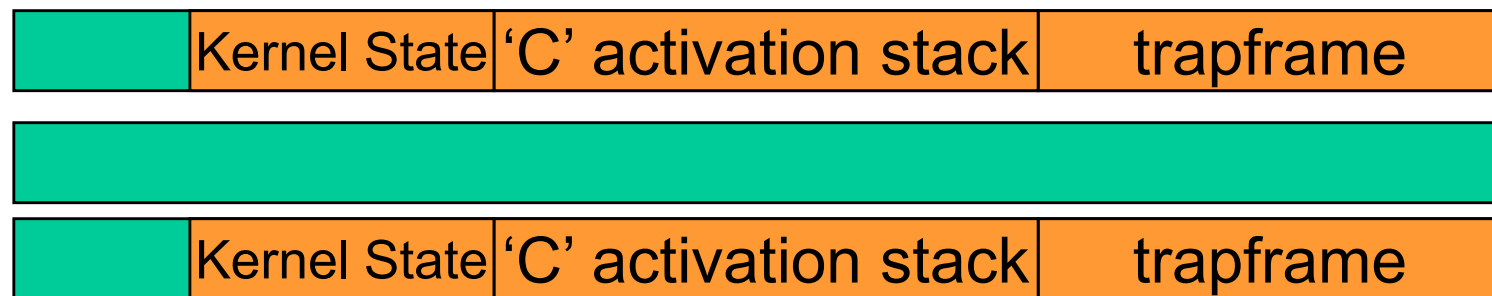
SP



# The Interesting Part of a Thread Switch

- What does the “push kernel state” part do???

SP



# OS/161 md\_switch

```
md_switch(struct pcb *old, struct pcb *nu)
{
    if (old==nu) {
        return;
    }
    /*
     * Note: we don't need to switch curspl, because splhigh()
     * should always be in effect when we get here and when we
     * leave here.
     */

    old->pcb_kstack = curkstack;
    old->pcb_ininterrupt = in_interrupt;

    curkstack = nu->pcb_kstack;
    in_interrupt = nu->pcb_ininterrupt;

    mips_switch(old, nu);
}
```



# OS/161 mips\_switch

```
mips_switch:
/*
 * a0 contains a pointer to the old thread's struct pcb.
 * a1 contains a pointer to the new thread's struct pcb.
 *
 * The only thing we touch in the pcb is the first word, which
 * we save the stack pointer in. The other registers get saved
 * on the stack, namely:
 *
 *      s0-s8
 *      gp, ra
 *
 * The order must match arch/mips/include/switchframe.h.
 */

/* Allocate stack space for saving 11 registers. 11*4 = 44 */
addi sp, sp, -44
```





# OS/161 mips\_switch

```
/* Save the registers */
```

```
sw ra, 40(sp)
```

```
sw gp, 36(sp)
```

```
sw s8, 32(sp)
```

```
sw s7, 28(sp)
```

```
sw s6, 24(sp)
```

```
sw s5, 20(sp)
```

```
sw s4, 16(sp)
```

```
sw s3, 12(sp)
```

```
sw s2, 8(sp)
```

```
sw s1, 4(sp)
```

```
sw s0, 0(sp)
```

```
/* Store the old stack pointer in the old pcb */
```

```
sw sp, 0(a0)
```

Save the registers  
that the 'C'  
procedure calling  
convention  
expects  
preserved



# OS/161 mips\_switch

```
/* Get the new stack pointer from the new pcb */
lw    sp, 0(a1)
nop                    /* delay slot for load */

/* Now, restore the registers */
lw    s0, 0(sp)
lw    s1, 4(sp)
lw    s2, 8(sp)
lw    s3, 12(sp)
lw    s4, 16(sp)
lw    s5, 20(sp)
lw    s6, 24(sp)
lw    s7, 28(sp)
lw    s8, 32(sp)
lw    gp, 36(sp)
lw    ra, 40(sp)
nop                    /* delay slot for load */

/* and return. */
j    ra
addi  sp, sp, 44      /* in delay slot */
.end  mips_switch
```



# Revisiting Thread Switch

Thread a

mips\_switch(a,b)  
{  
  
}

mips\_switch(a,b)  
{

Thread b

→ }

← mips\_switch(b,a)  
{

→ }

