澳 門 理 工 學 院
Instituto Politécnico de Macau
Macao Polytechnic Institute

School of Applied Sciences (B.Sc. in Computing)

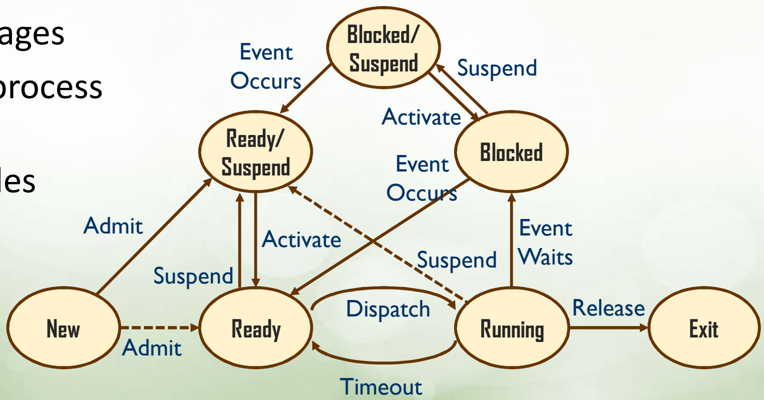
Notes #4: Processes and Threads

COMP 213
(21121/21221)
Operating Systems

2019-2020 1st Semester

Review

- PCBs and process images
- 5-state, and 7-state process models
- Different control tables



Eddie Law

Topics

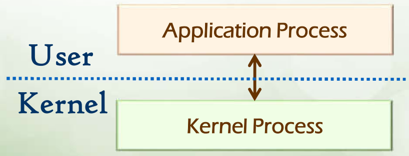
- Parent-child processes
- Process vs. Thread
- Multithreading
- Process image in multithreaded process
  - Thread Control Block (TCB)
- Thread switching
- Benefits of threads
- Symmetric multiprocessing (SMP) and multi-core
- Chapters 4.2, 4.1, + this note

Processes

- Have been studying about "processes"
- 7-state process models for multiprogramming
  - Resource ownership – a virtual address space to hold process image
  - Scheduling/execution – interleaving execution path with other processes
- But no discussion on user-level process and kernel-level process relationship yet

Linking User-level and Kernel-level Processes

- It's natural to have a kernel-level process working with a user-level process
- One-to-one mapping between user-level and kernel-level process?
  - E.g., MS-DOS

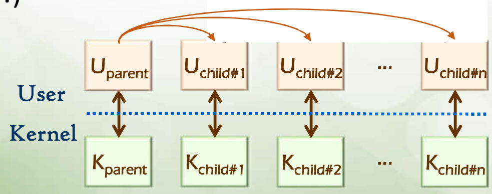


Process: Concurrency

- Process
  - An "abstract" model
  - Exchanges of control should be done automatically and implicitly on multiprogramming paradigm
- Multiprogramming is natural on multicore systems
- But can a process be run in parallel on multiprocessor? Even on a uniprocessor system??

Process and Thread: Concurrency

- How can we run **ONE** process in parallel?
  - Parent-child processes?
  - Multi-threading?
    (discuss soon!)



Start with Parent-Child Process Design

- For example, in Linux, BSD…
- `fork()`: for creation of a child process with an image copy of the parent process
  - `fork()` returns 0 for child process, returns child PID for parent

```
pid = fork()
if (pid == 0) cout << “child process”;
else cout << “parent process”;
```

- `wait()`: for synchronization of processes
  - A parent waits for the termination of child process
  - Parent process is awakened; parent can determine which of its child processes terminates

How Expensive to Create Process?

- Must construct new PCB
  - Inexpensive
- Must set up new tables for address space
  - More expensive
- Copy I/O state (file handles, etc)
  - Medium expense
- Copy data from parent process? (e.g., `fork()`)
  - Semantics of `fork()` are that the child process ***gets a complete copy of the parent memory and I/O state***
  - Originally *very* expensive
  - Much less expensive with "copy on write"

Eddie Law

Parallelism: An Example

- Consider calculating roots of a quadratic equation

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

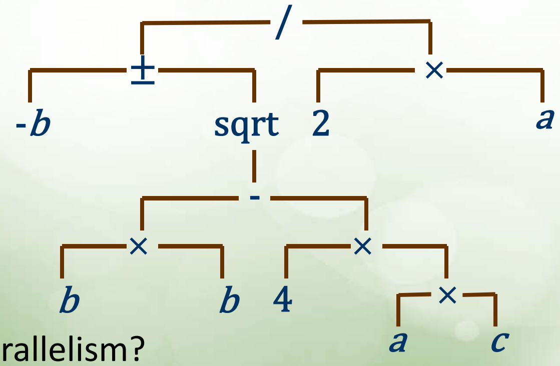
- To program it in **sequential statements**:

$$t_1 = b \times b$$
$$t_2 = 4 \times a$$
$$t_3 = t_2 \times c$$
$$t_4 = t_1 - t_3$$
$$t_5 = \mathrm{sqrt}(t_4)$$
$$t_6 = -b + t_5$$
$$t_7 = -b - t_5$$
$$t_8 = 2 \times a$$
$$t_9 = t_6 \div t_8 (\mathrm{ans1})$$
$$t_{10} = t_7 \div t_8 (\mathrm{ans2})$$

Eddie Law 10

If Breaking Down the Equation

- Decompose the equation from observation (outside in)



- Do you see the parallelism?

Parallelized the Computations

- From bottom up

$$t_1 = b \times b$$

$$t_2 = a \times c$$

$$t_3 = 4 \times t_2$$

$$t_4 = t_1 - t_3$$

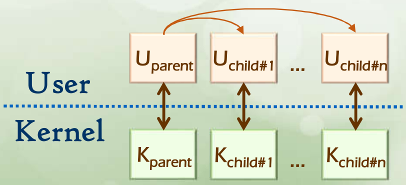
$$t_5 = \mathrm{sqrt}(t_4)$$

$$t_6 = -b - t_5 \qquad t_8 = 2 \times a \qquad t_7 = -b + t_5$$

$$\mathrm{ans}_1 = \frac{t_6}{t_8} \qquad \mathrm{ans}_2 = \frac{t_7}{t_8}$$

Fork and Join Process Notation

- (Beware: following not the Unix's `fork()` call)
- `fork( )` primitive
  - Creates a named parallel process that executes a designated code fragment
- `join( )` primitive
  - Waits for a parallel process created by a fork to terminate
- `quit( )`
  - Terminates a forked process



Eddie Law

Fork/Join: Example

process Main

fork A; t_1 = b*b; join A

$t_4 = t_1 - t_3$; t_5 = sqrt(t_4)

fork B; fork C; $t_6 = -b - t_5$; join B; join C

fork D; $t_{10} = t_6/t_8$; join D

end process Main

process A

t_2 = 4 * a; $t_3 = t_2$ * c

end process A

process B

t_8 = 2 * a

end process B

process C

$t_7 = -b + t_5$

end process C

process D

$t_9 = t_7 / t_8$

end process D

Eddie Law

Co-begin, Co-end

- One way to restrict programs to make code less obscure
  - Is to make the control statements of the program properly nested
- **Co-begin co-end** pairs
  - Enforce the proper nesting of parallel processes.
- Statements enclosed by a **co-begin co-end** pair are executed in parallel

Eddie Law

Co-begin, Co-end

- Unlike **fork** and **join**, non-nested dependencies between processes cannot be expressed
- The **co-begin**, **co-end** primitives also in form of **par-begin**, **par-end**

Eddie Law

Co-begin, Co-end: Example

begin Main

cobegin

t_1 = b*b;

begin

t_2 = 4*a;

$t_3 = t_2$*c

end

coend

$t_4 = t_1 - t_3$

t_5 = sqrt(t_4)

cobegin

t_6 = -b - t_5;

t_8 = 2*a;

t_7 = -b + t_5;

coend

cobegin

$t_{10} = t_6/t_8$;

$t_9 = t_7/t_8$

coend

end Main

Eddie Law

Process and Thread

- Faster performance through parallelization though multiple processes
- Any problems?
  - For each process, resources allocated by OS, including memory to hold the process image
  - All duplicated code base for duplicated processes, too much redundancy, waste resources
  - Further, follows an execution path that may be interleaved with other processes → bad case would be that some processes may have to wait
- A newer design → modern OS runs "thread" in kernel
  - Multithreading design
  - What is a thread?

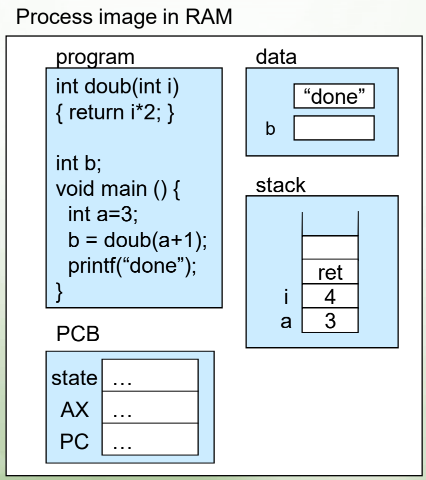
Eddie Law 18

Process Image

- This example: 1 process (1 thread)

Each process is allocated some resources, including the memory to hold the process image.

For two processes A and B:
- Process A **cannot** access the memory of process B.
- Process A and B **cannot** share data in memory.



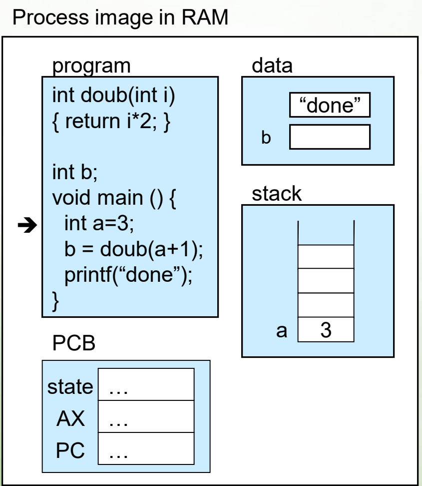
Process/Thread Execution (1)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

Also notice how the stack changes in the course of execution.



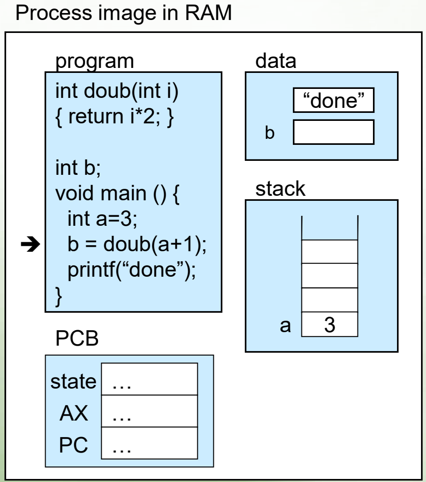
Process/Thread Execution (2)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

Also notice how the stack changes in the course of execution.



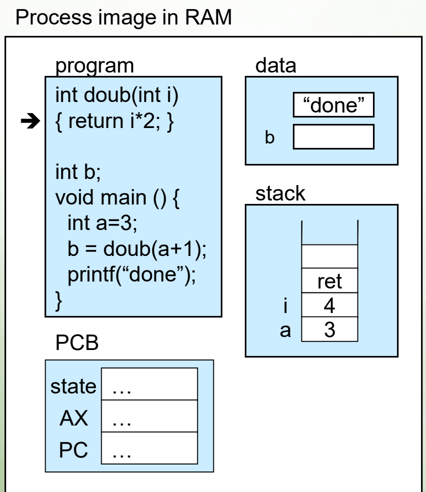
Process/Thread Execution (3)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

Also notice how the stack changes in the course of execution.



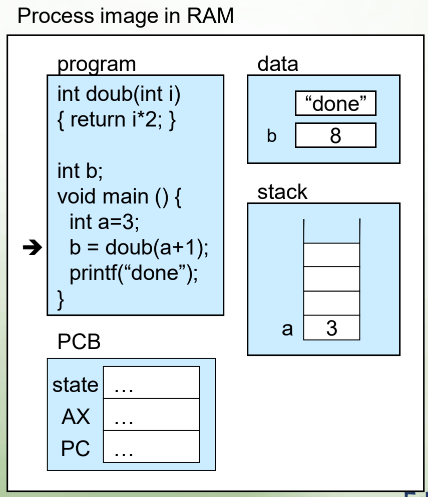
Process/Thread Execution (4)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

Also notice how the stack changes in the course of execution.



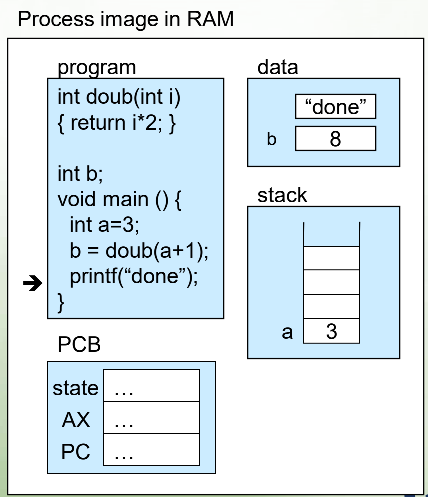
Process/Thread Execution (5)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

Also notice how the stack changes in the course of execution.



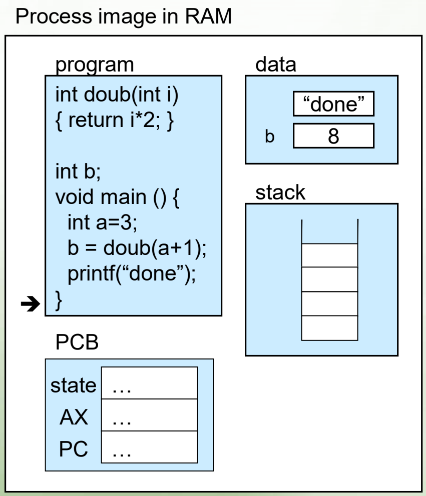
Process/Thread Execution (6)

Each process has **one** thread of execution. At any time, only one instruction is being executed.

The arrow indicates the current instruction being executed.

Its relation with the register PC and the PC value in the PCB.

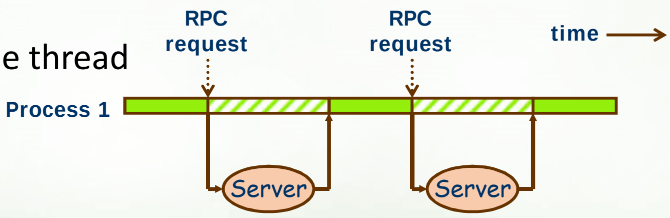
Also notice how the stack changes in the course of execution.



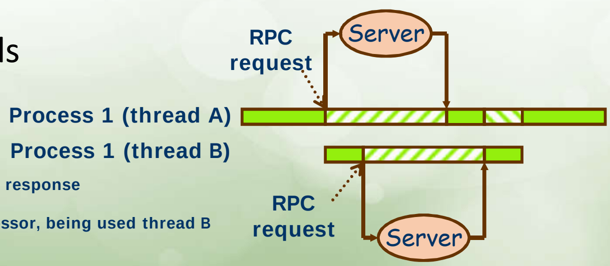
Eddie Law 25

Example: Remote Procedure Call (RPC)

- One process, single thread



- One process, two threads
  - Earlier completion



Blocked, waiting for RPC's response

Blocked, waiting for processor, being used thread B

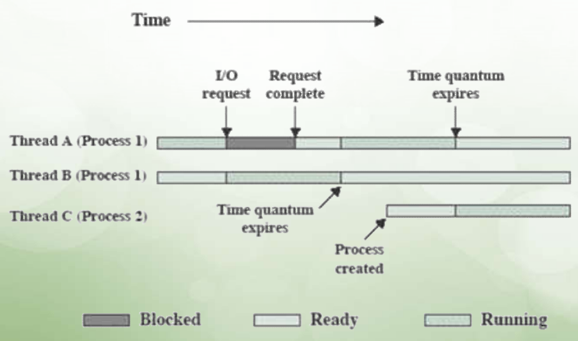
Running

Eddie Law

Multithreading

- Multiprogramming, multiple threads of execution

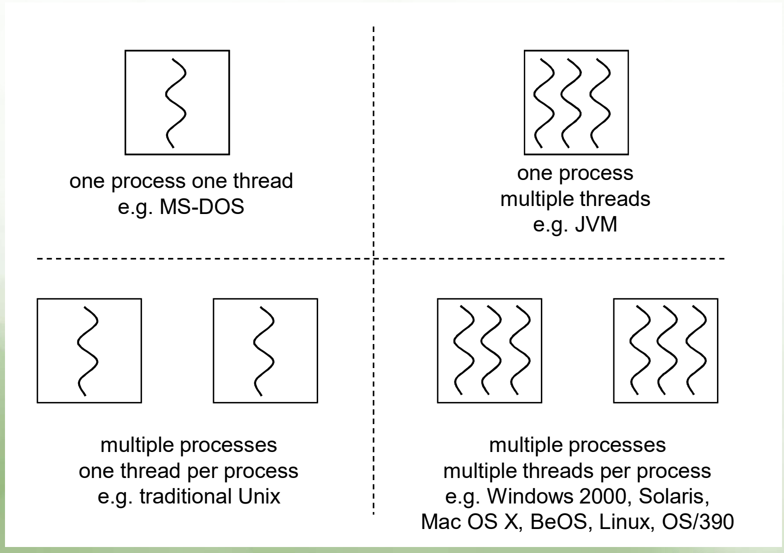
Multithreading vs Multiprogramming



Multithreading example on uniprocessor

Eddie Law 27

OS Support for Threads and Processes



Eddie Law 28

Process vs. Thread

- A process has one or more threads
- Process – Unit of resource ownership
  - Some resources allocated by OS, including memory, open files
- Thread – Unit of dispatching
  - An execution path
  - Execution may be interleaved with other threads / processes

Process and Threads

- In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection
- The following are associated with processes:
  - A virtual address space that holds the process image
  - Protected access to processors, other processes (for inter-process communication), files, and I/O resources (devices and channels)

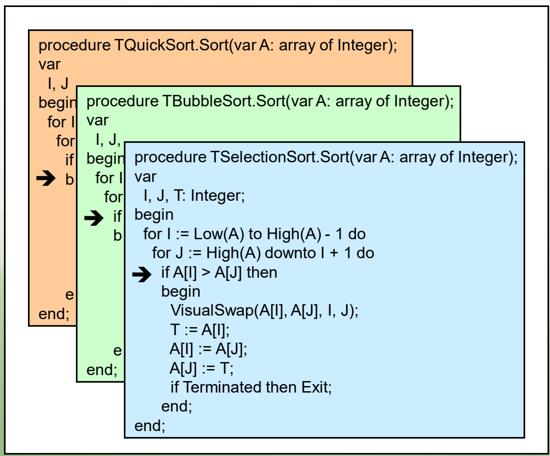
Process and Threads (cont'd)

- Within a process, there may be one or more threads, each with the following:
  - A thread execution state (Running, Ready, etc.)
  - A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process, shared with all other threads in that process

Eddie Law 31

Multithreading, Example

Process ThdDemo



When we run the *program* thddemo.exe, we create a *process* which consists of three *threads*: one thread for each sort procedure.

Actually, there is one more thread, the primary thread, in the process thddemo.

Eddie Law 32

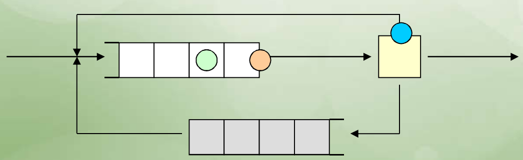
Thread as Unit of Dispatching

Process ThdDemo

```
procedure TQuickSort.Sort(var A: array of Integer);
var
  I, J
begin
  for I
    for
```

```
procedure TBubbleSort.Sort(var A: array of Integer);
var
  I, J,
begin
  for I
    for
```

```
procedure TSelectionSort.Sort(var A: array of Integer);
var
  I, J, T: Integer;
begin
  for I := Low(A) to High(A) - 1 do
    for J := High(A) downto I + 1 do
      … …
```



The OS schedules the three threads. Each thread is dispatched in turn, one at a time. The three threads of the process appear to be running 'at the same time'.

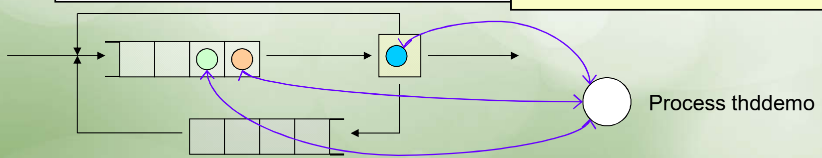
In Chap 3, the "process ball" that represents a process is in fact the single thread within the processes. And process switching is in fact thread switching.

Process as Unit of Resource Ownership

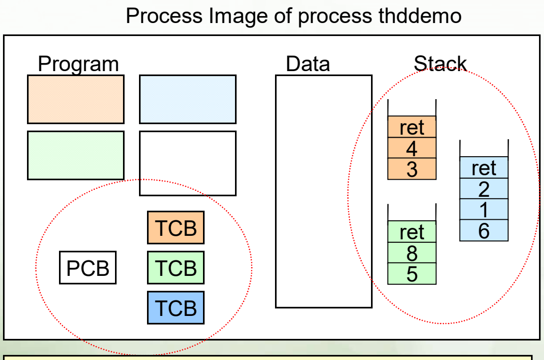
Process Image of process thddemo

Program	Data	Stack
procedure TQuickSort.Sort … …	Global variables, constant, etc	
procedure TBubbleSort.Sort … …		
procedure TSelectionSort.Sort … …		
begin { begin } … end.		

The three threads belong to a single process, which owns resources. They share memory, opened files and I/O resources. Only a copy of the program is stored. The three threads use the same copy of global variables.

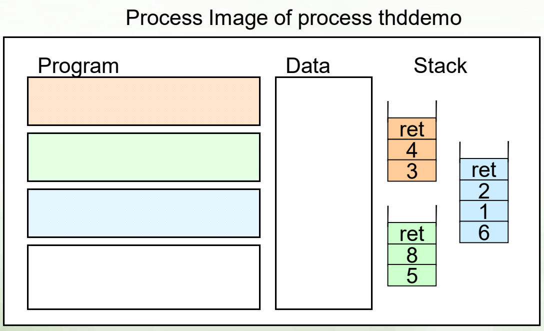


Stack and PCB in Multithreaded Process



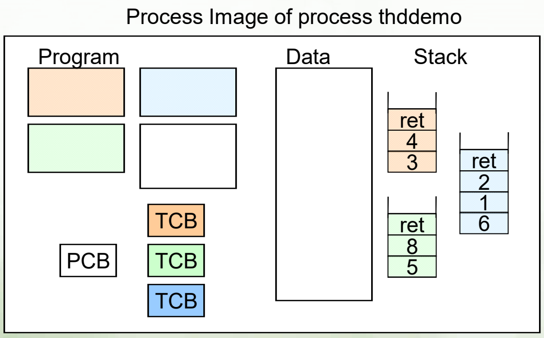
We see that a single copy of program and data are shared by the multiple threads of a multithreaded process. However, a single stack and a PCB is not sufficient.

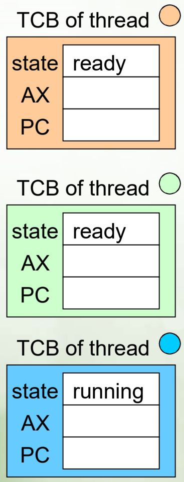
Each thread needs a user stack



Stack stores local variables in function call. Since each thread can run independently, their local variables appear and disappear in any order. We cannot use a single stack for multiple threads. Instead, each thread has its own stack.

PCB and TCB (Thread Control Block)





Each thread has its own state and thread context (the value of the CPU registers last time this thread is executed). Hence, a PCB alone is not sufficient. The state and context of each thread are stored in the Thread Control Block (TCB) of the thread. PCB stores information related to the whole process.

Splitting Information on PCB

- Process Identification:
  - Process ID, User ID [P], Thread ID [T]
- Processor State Information [T]
- Process Control Information
  - Scheduling [T] [P] and state information [T] [P]
  - Data structuring [T] [P]
  - Inter-process communication [P]
  - Process privileges [P]
  - Memory management [P]
  - Resource ownership and utilization [P]

Threads in Process: Resource Sharing

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

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	

Eddie Law

Creation and Termination

- Usually, a process is created with a single thread (called primary thread in Windows) This thread may create more threads.
- A process terminates when
  - in Windows: the primary thread terminates
  - in Unix: all threads of the process terminate

Eddie Law 40

Activities similar to Processes

- Threads have execution states and may synchronize with one another.
  - Similar to processes
- We look at these two aspects of thread functionality in turn.
  - States
  - Synchronisation (in Ch. 5)

Thread Execution States

- Key **states** for a thread
  - Running
  - Ready
  - Blocked

What about 'Suspended' state?

- **Operations** associated with a change in thread state
  - Spawn (another thread)
  - Block
    - Issue: will blocking a thread block other, or all, threads
  - Unblock
  - Finish (thread)
    - De-allocate register context and stacks

Thread Suspend State?

- Suspending a process involves suspending all threads of the process

since all threads share the same address space

Why?

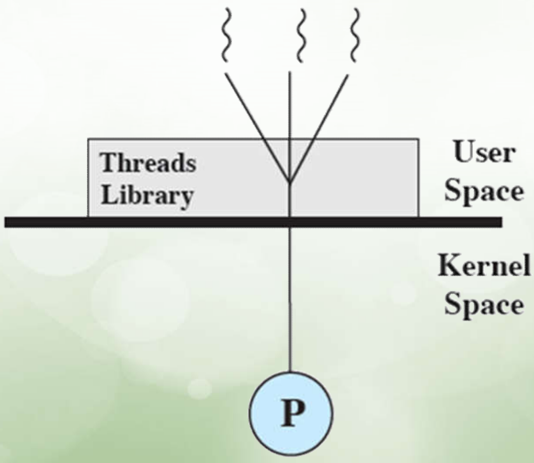
Thread Synchronization

More on this in Chapter 5

- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

User-Level Threads (ULTs)

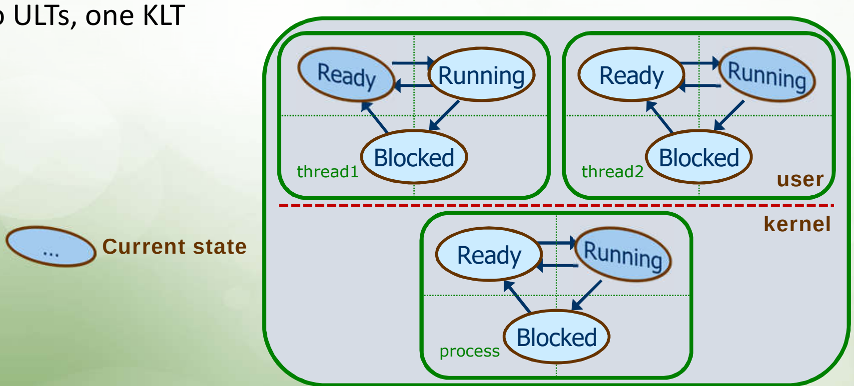
- One of the two types of threads
- All thread management is done by the application
- The kernel is not aware of the existence of thread



Eddie Law 45

Example on ULT

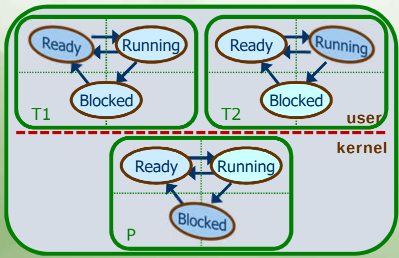
- Consider state 1 of a process
  - Two ULTs, one KLT



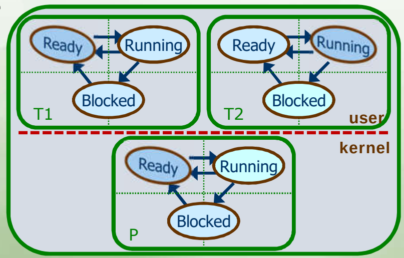
Eddie Law

E.G. (cont')

- From state 1
- Thread 2 (T2) makes a blocking I/O call → kernel invokes I/O → places Process (P) to blocked state
- The T library maintains T2 in running state



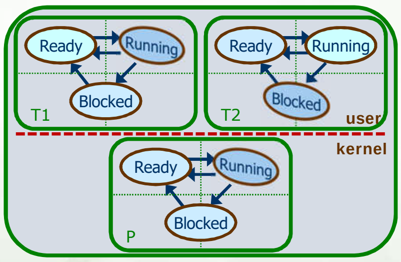
- From state 1
- Clock interrupt → moves to kernel → time slot for P is over → places P to ready state
- The T library maintains T2 in running state



Eddie Law

E.G. (cont')

- From state 1
- T2 is waiting data from T1 → T2 is blocked and T1 becomes ready
- Kernel is not invoked → state of P is unchanged



Eddie Law

Advantages of ULTs

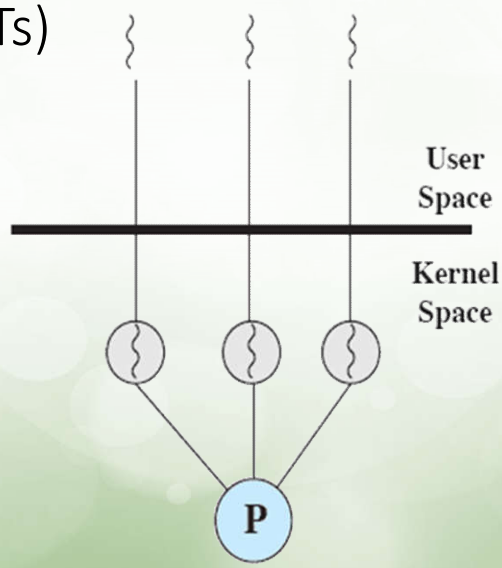
- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS

Disadvantages of ULTs

- In a typical OS, many system calls are blocking → all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

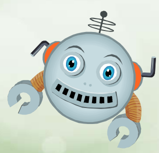
Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows is an example of this approach



Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded



Disadvantage of KLTs

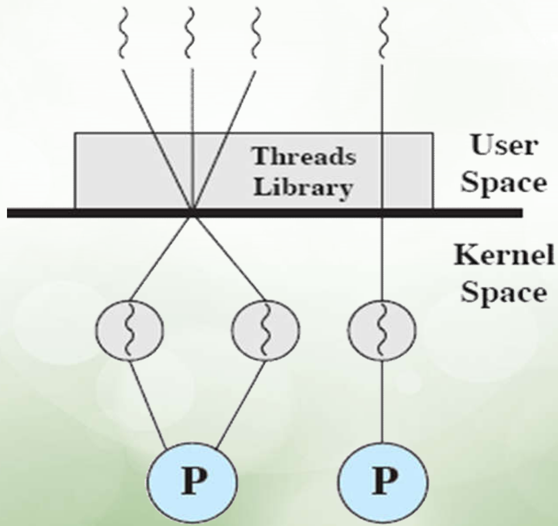
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Null Fork: the time to create, schedule, execute, and complete a process/thread that invokes the null procedure (i.e., the overhead of forking a process/thread)
Signal-Wait: the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together)

Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example



Relationship between Threads and Processes

Threads: Processes	Description
1:1	Each thread of execution is a unique process with its own address space and resources.
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.
M:N	It combines attributes of M:1 and 1:M cases.

Process, in Summary

- Have a virtual address space which holds the process image, which contains:
  - program and data
  - one stack per thread
  - PCB and one TCB per thread
- Protected access to memory, files, and I/O resources
- A process has one or more threads

Thread, in Summary

- Has an execution state
  - Running, ready, blocked, …
- Has an execution stack
- Thread context is saved and restored in thread switching
  - Read the following slides for how to change the steps of process switching to the steps of thread switching
- Has access to the memory and resources of its process
- Has some per-thread static storage for local variables

Thread Switching, with Steps

1. Save processor state (incl. program counter and other registers) in the TCB: processor state information
2. Update the TCB with the new state and any accounting information
3. Move the TCB to appropriate queue – ready, blocked

Thread Switching, Steps (cont'd)

4. Select another thread for execution
5. Update the TCB of the thread selected (new state and accounting information)
6. Update memory-management data structures
7. Restore context of the selected thread
    - Restore the previous value of the program counter and other registers from the TCB

Eddie Law 59

Thread Switching, Steps (Simplified)

- Thread switching from thread *A* to thread *B*:

1. Save thread context of thread *A*: CPU $\Rightarrow$ TCB *A*
2. Update state in TCB *A*
3. Move TCB *A* to appropriate queue
4. Select another thread (thread *B*)
5. Update state in TCB *B*
6. Update memory-management data structures
7. Restore thread context of thread *B*: TCB *B* $\Rightarrow$ CPU

Eddie Law 60

Benefits of Threads (1)

- Less time to create / terminate a new thread than a process
- Less time to switch between two threads within the same process
- Communication among threads of the same process is easier

Benefits of Threads (2)

- Less time to create / terminate a new thread than a process
  - When the OS creates a process, it has to allocate memory for the process image, load the program and data, and initialize other resources
  - When the OS creates a thread within an existing process, it does not need to initialize the process image. It only needs to set up a stack and TCB for the new thread.

Benefits of Threads (3)

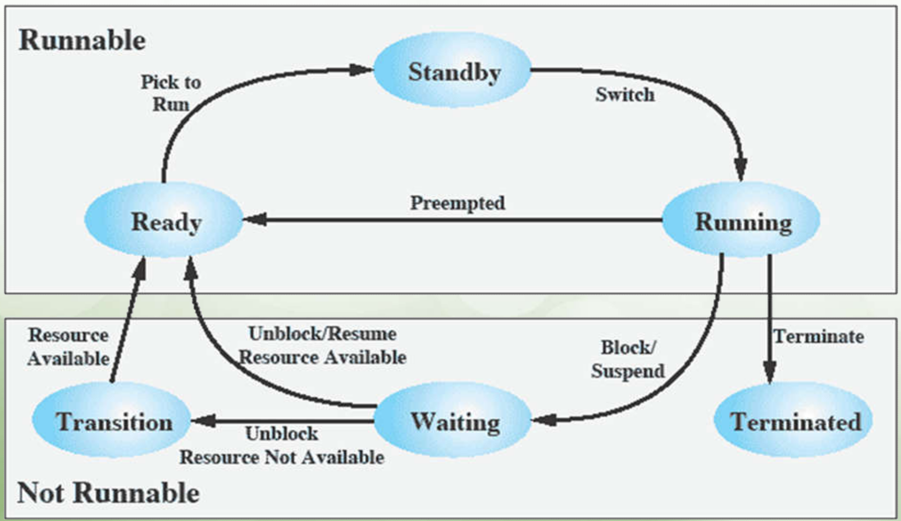
- Less time to switch between two threads within the same process
  - The “update memory management data structure” step is only necessary in case of thread switching between different process. (Why?)

Benefits of Threads (4)

- Communication among threads of the same process is easier
  - Threads within a process share memory and files
  - Different processes have to communicate with the help of the kernel

Kernel is the core of OS

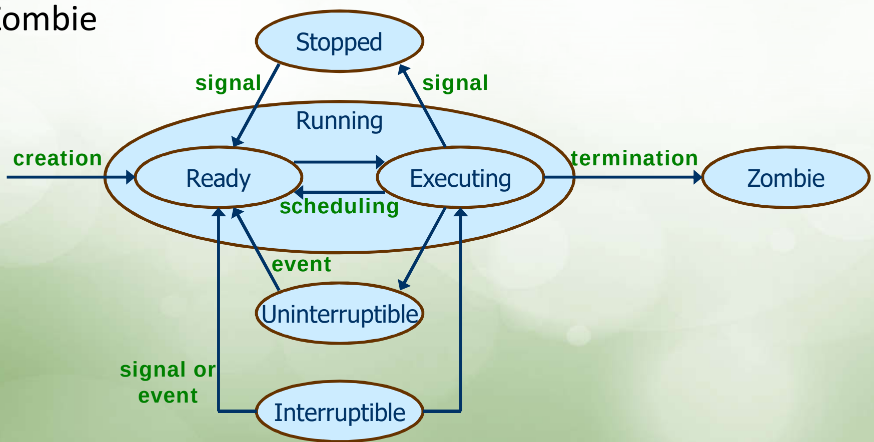
Example: Windows Thread States



Eddie Law 65

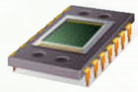
Linux States of a Process

- Process/thread model: Ready, Running, Interruptible, Uninterruptible, Stopped, Zombie



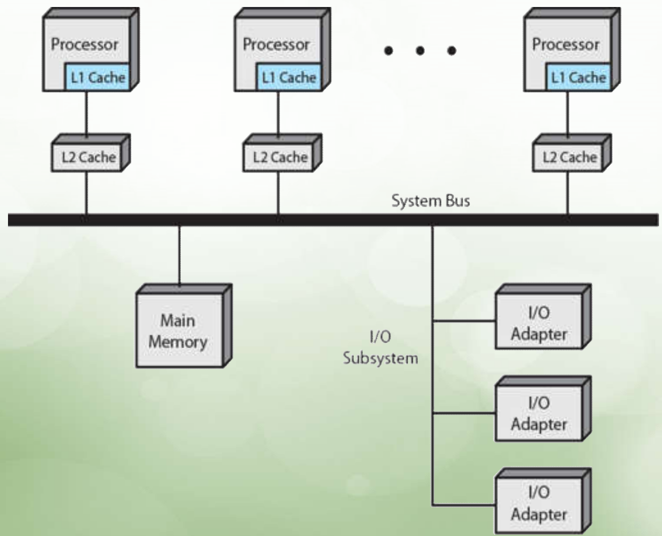
Eddie Law

Symmetric Multiprocessors (SMP) and Multi-core



- A stand-alone computer system with the following characteristics:
  - Two or more similar processors of comparable capability
  - Processors share the same main memory and are interconnected by a bus or other internal connection scheme
  - Processors share access to I/O devices
  - All processors can perform the same functions
  - The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels
- Multiprocessing - more than one processors in a machine

SMP Organization



SMP Advantages

Performance
- a system with multiple processors will yield greater performance if work can be done in parallel

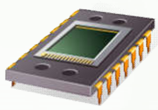
Scaling
- vendors can offer a range of products with different price and performance characteristics

Availability
- the failure of a single processor does not halt the machine

Incremental Growth
- an additional processor can be added to enhance performance

Eddie Law 69

Multicore Computer



- Also known as a multiprocessor chip
- Combines two or more processors (cores) on a single piece of silicon (die)
  - Each core consists of all of the components of an independent processor
- In addition, multi-core chips also include L1 cache and in some cases L2 cache

Eddie Law 70

Performance on Multicore

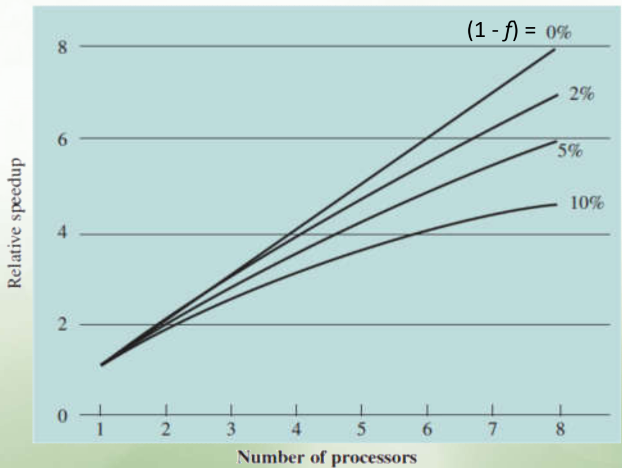
- Amdahl's Law

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \dfrac{f}{N}}$$

  - A program in which a fraction (1 - *f*) of the execution time involves code that is inherently serial, and a fraction *f* that involves code that is infinitely parallelizable with no scheduling overhead

Performance (1)

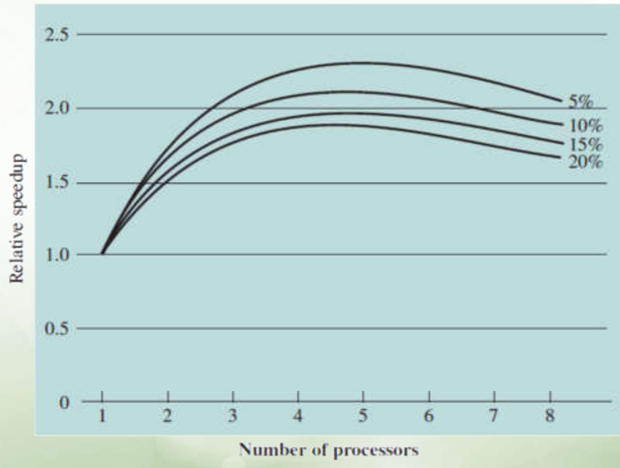
- If only 10% of the code is inherently serial (f = 0.9), running the program on a multicore system with 8 cores yields a performance gain of a factor of only 4.7



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Performance (2)

- Software typically incurs overhead as a result of communication and distribution of work to multiple processors and cache coherence overhead
- This results in a curve where performance peaks and then begins to degrade because of the increased burden of the overhead of using multiple processors



(b) Speedup with overheads

Eddie Law 73

Summary

- Process/related to resource ownership
- Thread/related to program execution
- Types of threads: User-level threads and kernel-level threads
- Symmetric Multiprocessors (SMP) and multicore

Eddie Law 74

Next Topic

- Concurrency: Mutual Exclusion and Synchronization
- Read Ch. 5