Cyber-Physical Systems

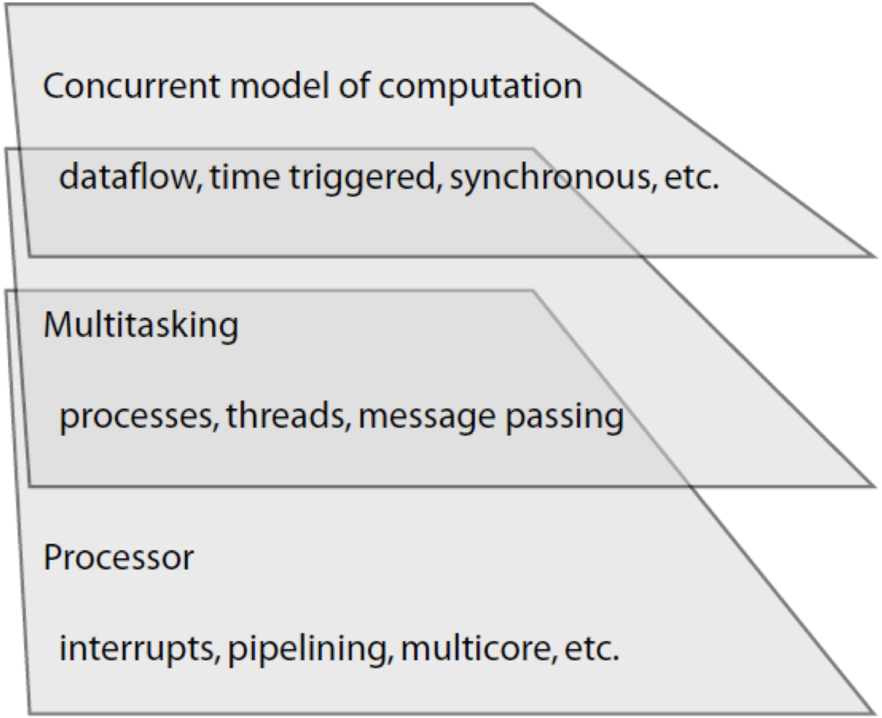
Multitasking

UNIVERSITY AT ALBANY
State University of New York

IECE 553/453– Fall 2019

Prof. Dola Saha

Layers of Abstraction for Concurrency



Concurrent model of computation

dataflow, time triggered, synchronous, etc.

Multitasking

processes, threads, message passing

Processor

interrupts, pipelining, multicore, etc.

Definition and Uses

- Threads are sequential procedures that share memory.
- Uses of concurrency:
  - Reacting to external events (interrupts)
  - Exception handling (software interrupts)
  - Creating the illusion of simultaneously running different programs (multitasking)
  - Exploiting parallelism in the hardware (e.g. multicore machines).
  - Dealing with real-time constraints.

OS Management of Application Execution

- Resources are made available to multiple applications

- The processor is switched among multiple applications so all will appear to be progressing

- The processor and I/O devices can be used efficiently

Process – several definitions

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

Process Elements

- Two essential elements of a process are:

Program code

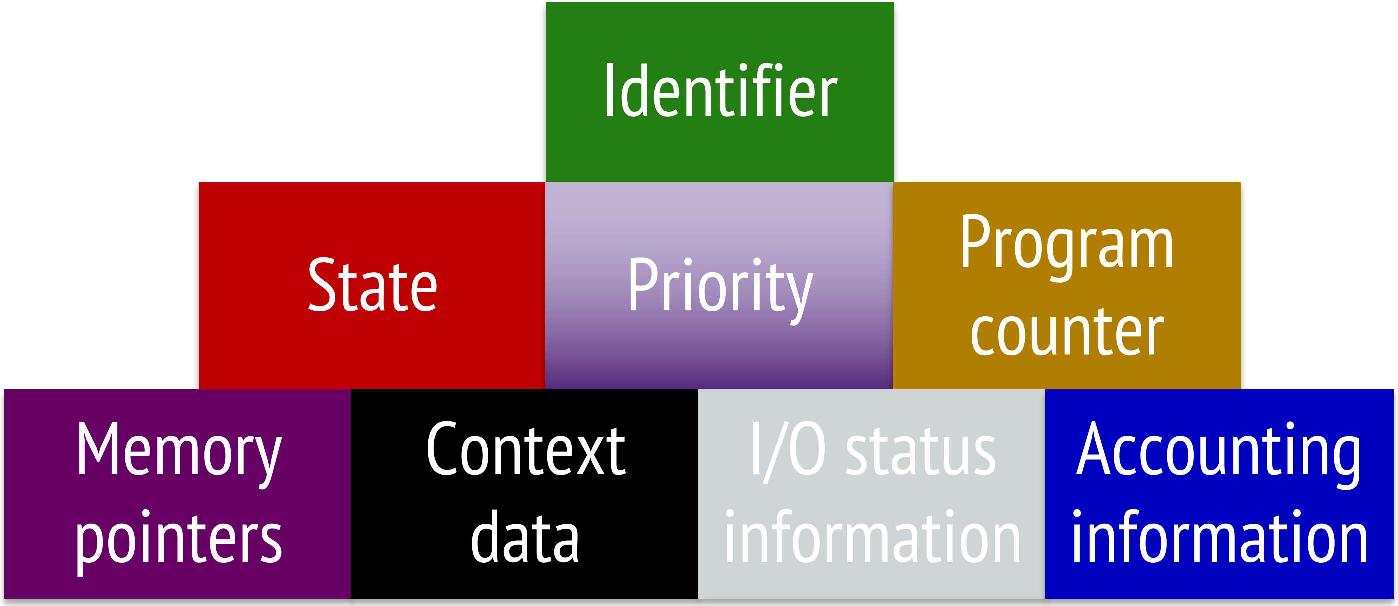
- which may be shared with other processes that are executing the same program

A set of data associated with that code

- when the processor begins to execute the program code, we refer to this executing entity as a process

Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements



Identifier

State

Priority

Program counter

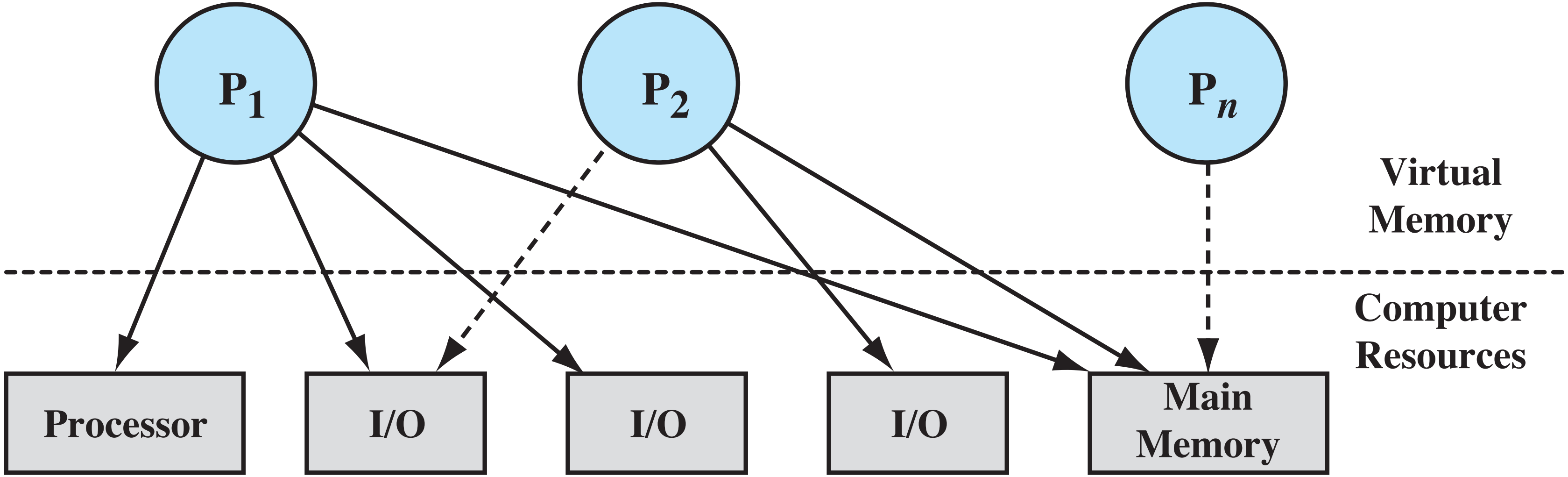
Memory pointers

Context data

I/O status information

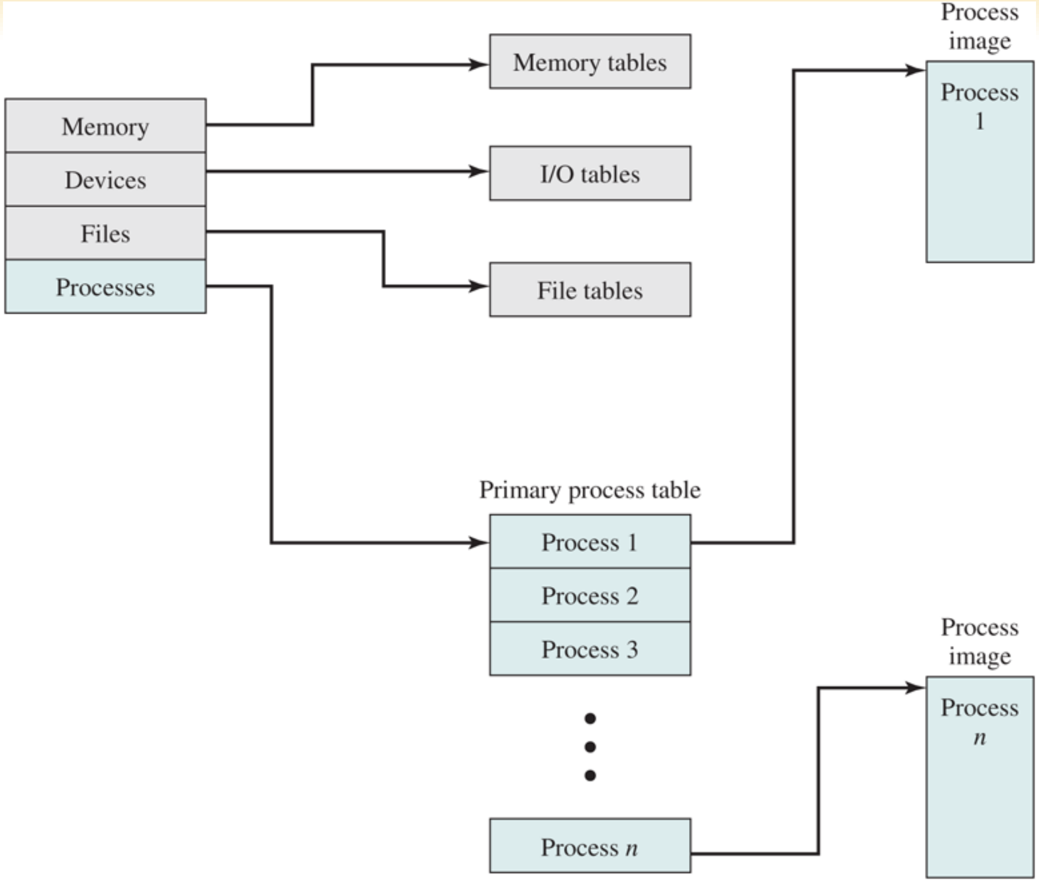
Accounting information

Processes & Resources



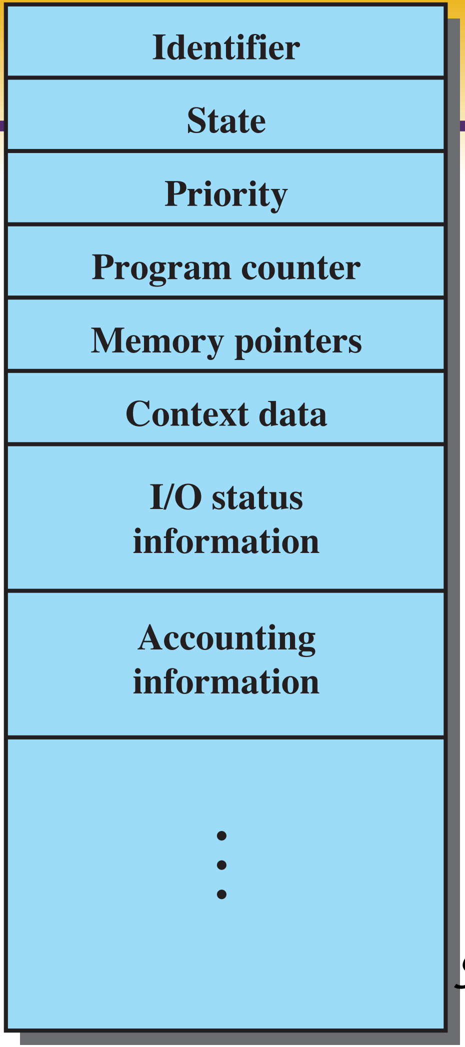
OS Control Structures

- **Memory tables**
  - used to keep track of both main (real) and secondary (virtual) memory.
  - Some is reserved for use by the OS; the remainder is available to processes.
  - Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism
- **I/O tables**
  - used to manage I/O devices and channels of the computer system.
- **File Tables**



Process Control Block

- ➢ Contains the process elements
- ➢ It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- ➢ Created and managed by the operating system
- ➢ Key tool that allows support for multiple processes



Identifier

State

Priority

Program counter

Memory pointers

Context data

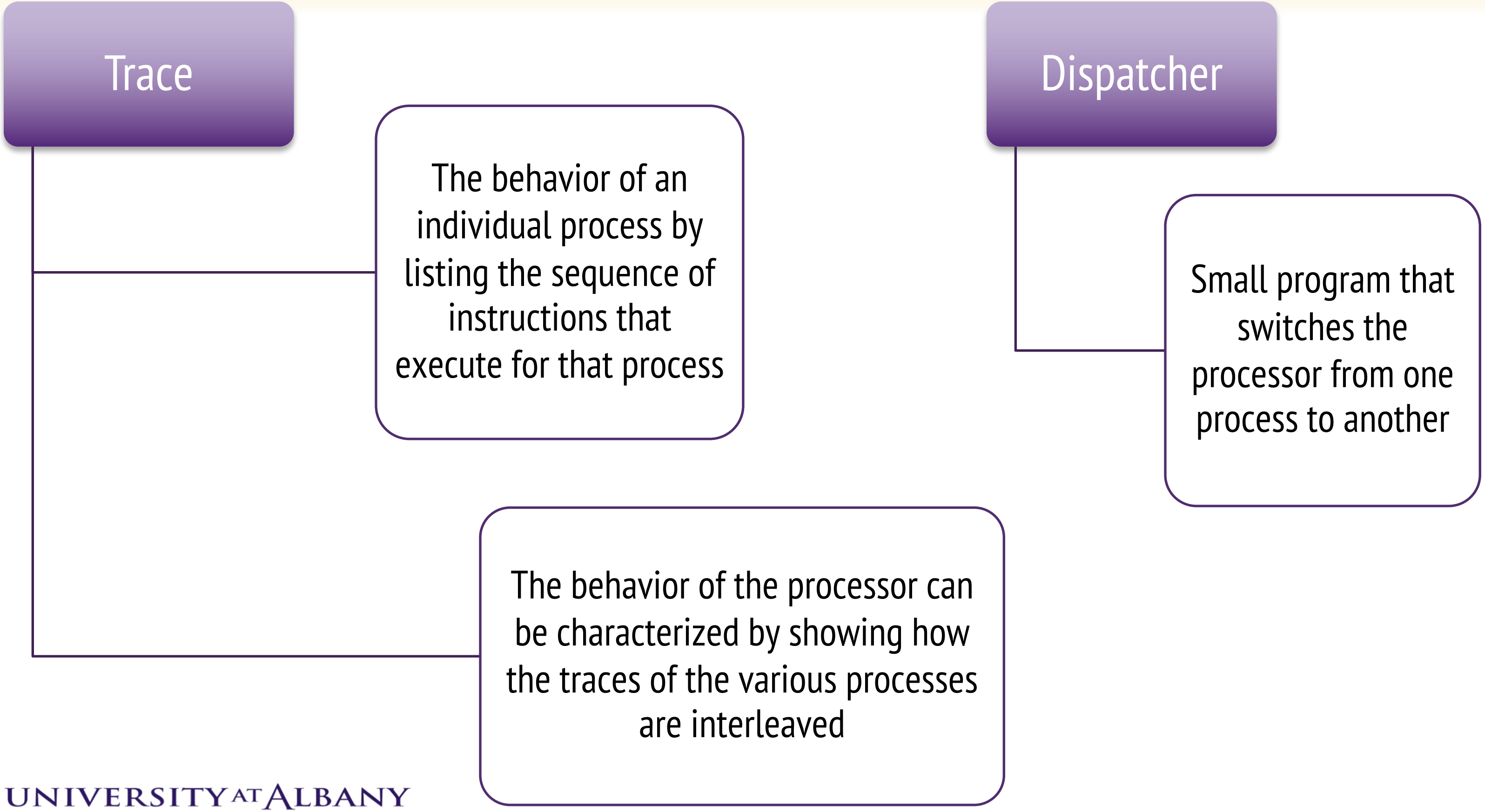
I/O status information

Accounting information

⋮

Stallings

Process States



Trace

The behavior of an individual process by listing the sequence of instructions that execute for that process

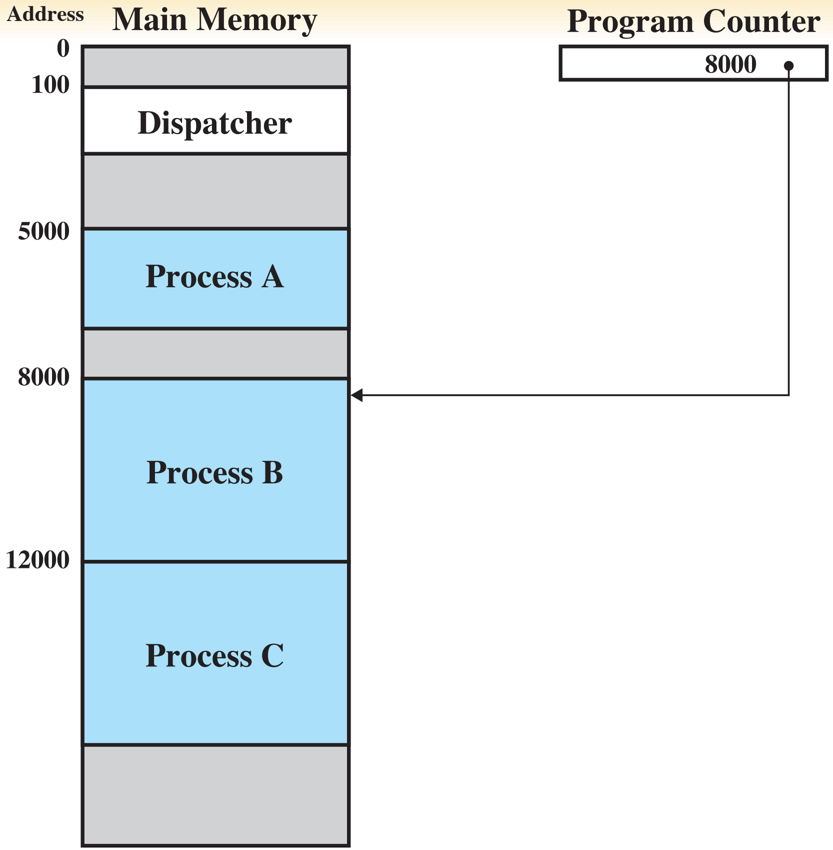
The behavior of the processor can be characterized by showing how the traces of the various processes are interleaved

Dispatcher

Small program that switches the processor from one process to another

Process Execution

- Memory layout of three processes
- Dispatcher program: switches processor from one process to another



Traces of Processes

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		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

Combined Traces

- Processor’s Point of View
- Blue shaded area is Dispatcher program

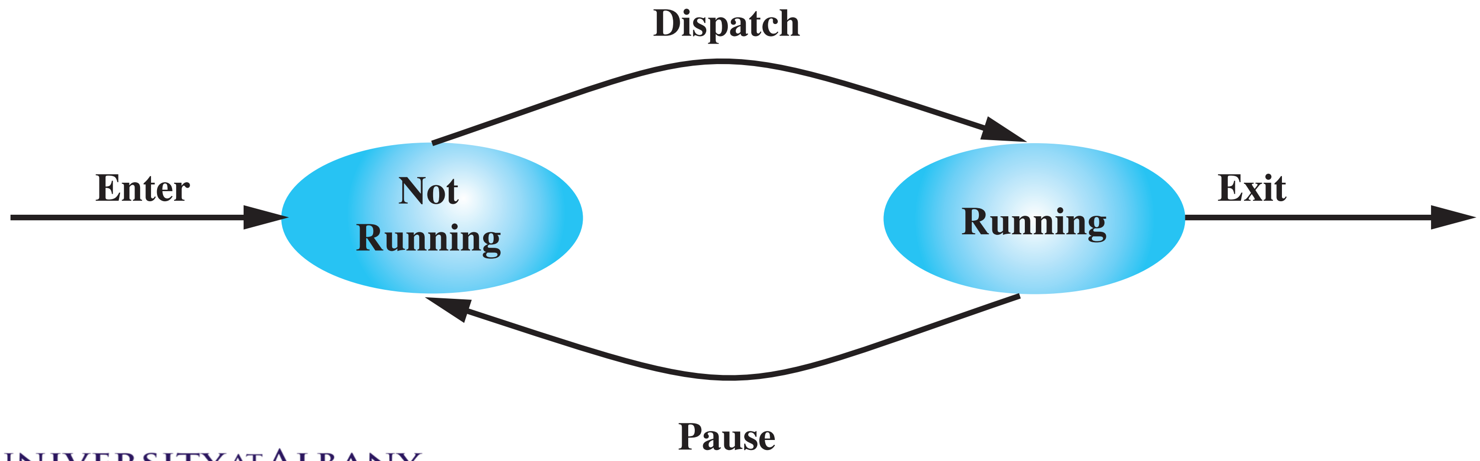
1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
--------------------Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------------I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
--------------------Timeout	
29	100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

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

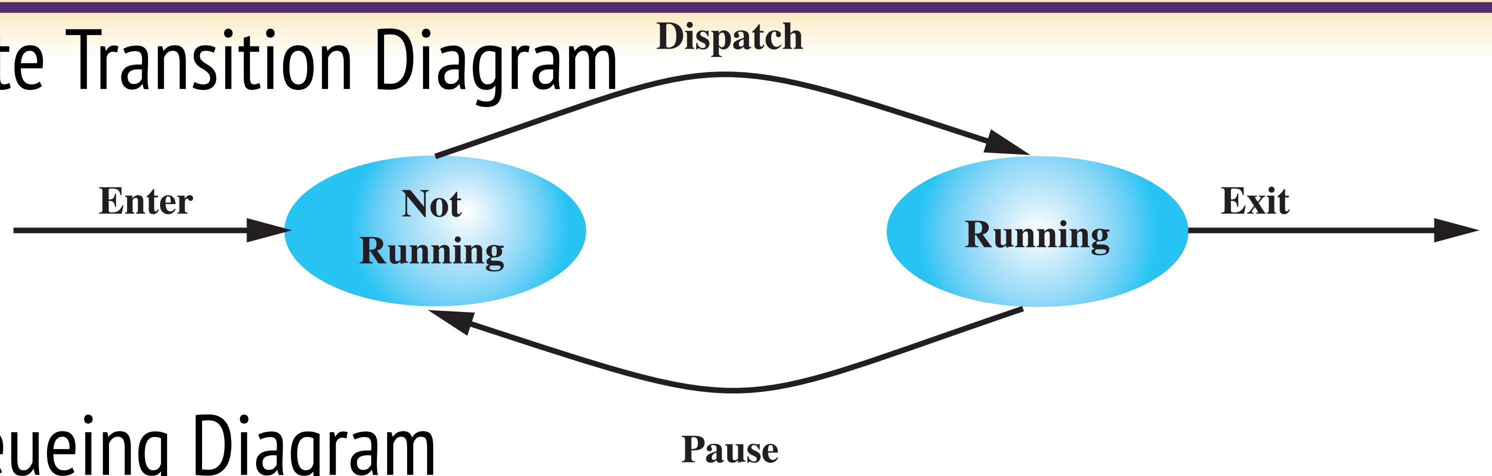
Two State Process Model

- Principal responsibility of OS is controlling the execution of processes
  - determining the interleaving pattern for execution and
  - allocating resources to processes.

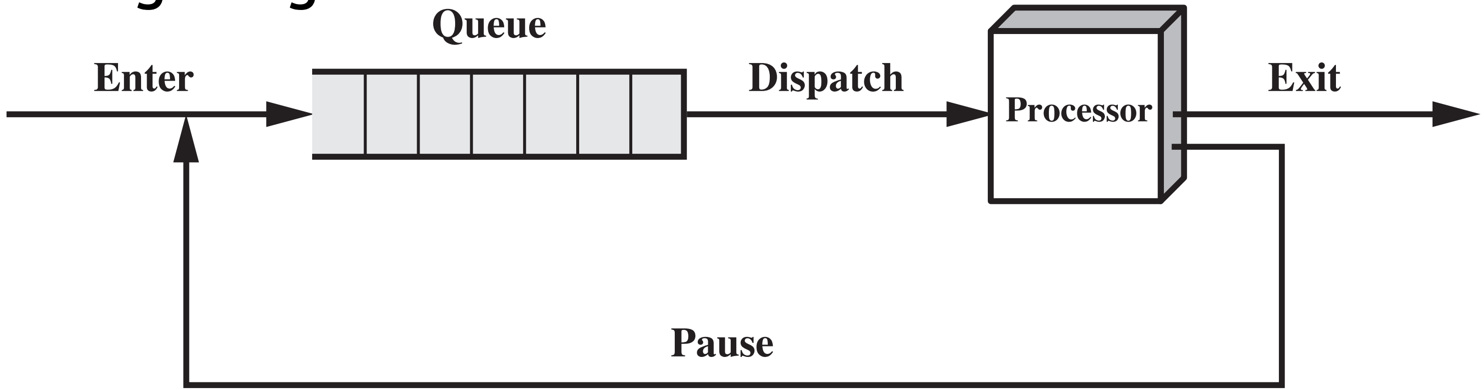


Two State Process Model

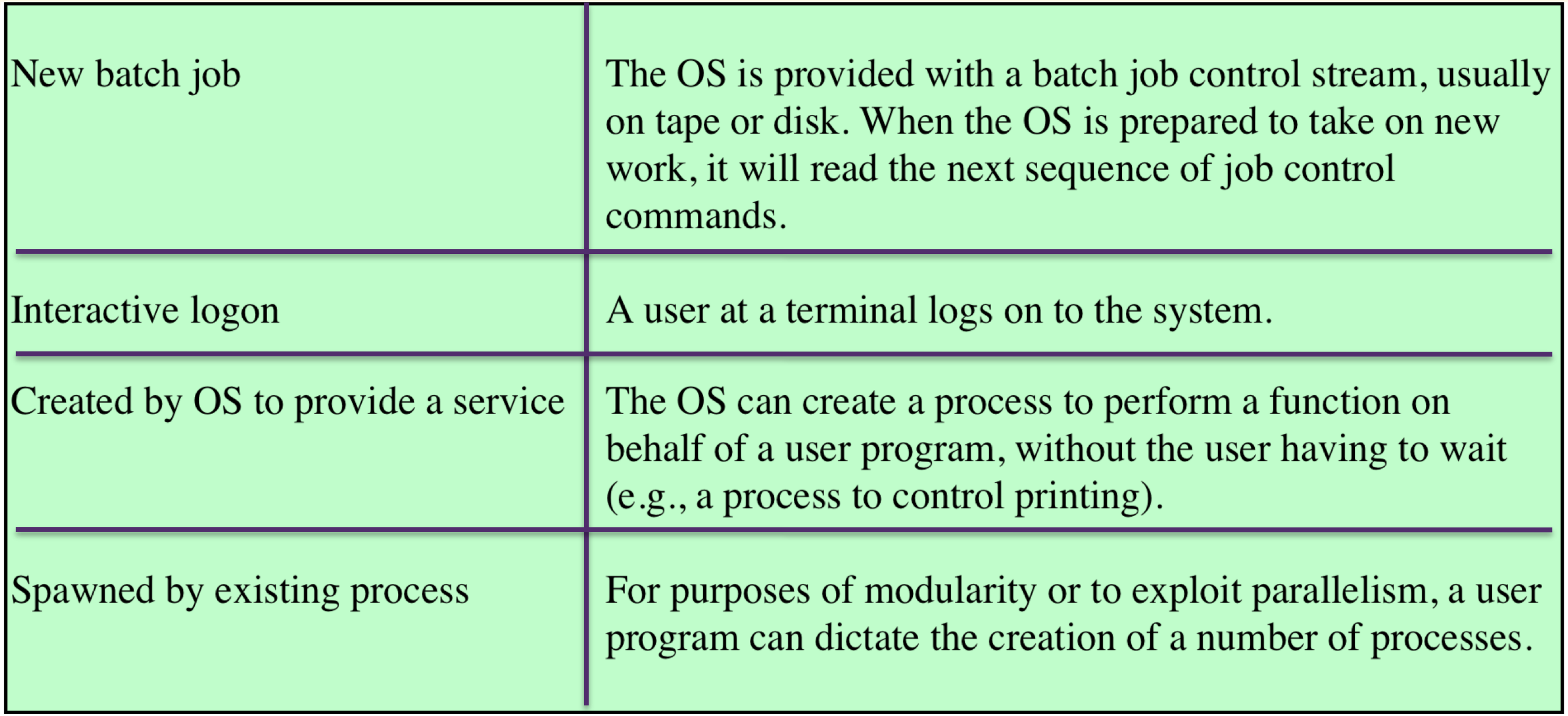
State Transition Diagram



Queueing Diagram



Reasons for Process Creation



New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Creation

Process spawning

- When the OS creates a process at the explicit request of another process

Parent process

- Is the original, creating, process

Child process

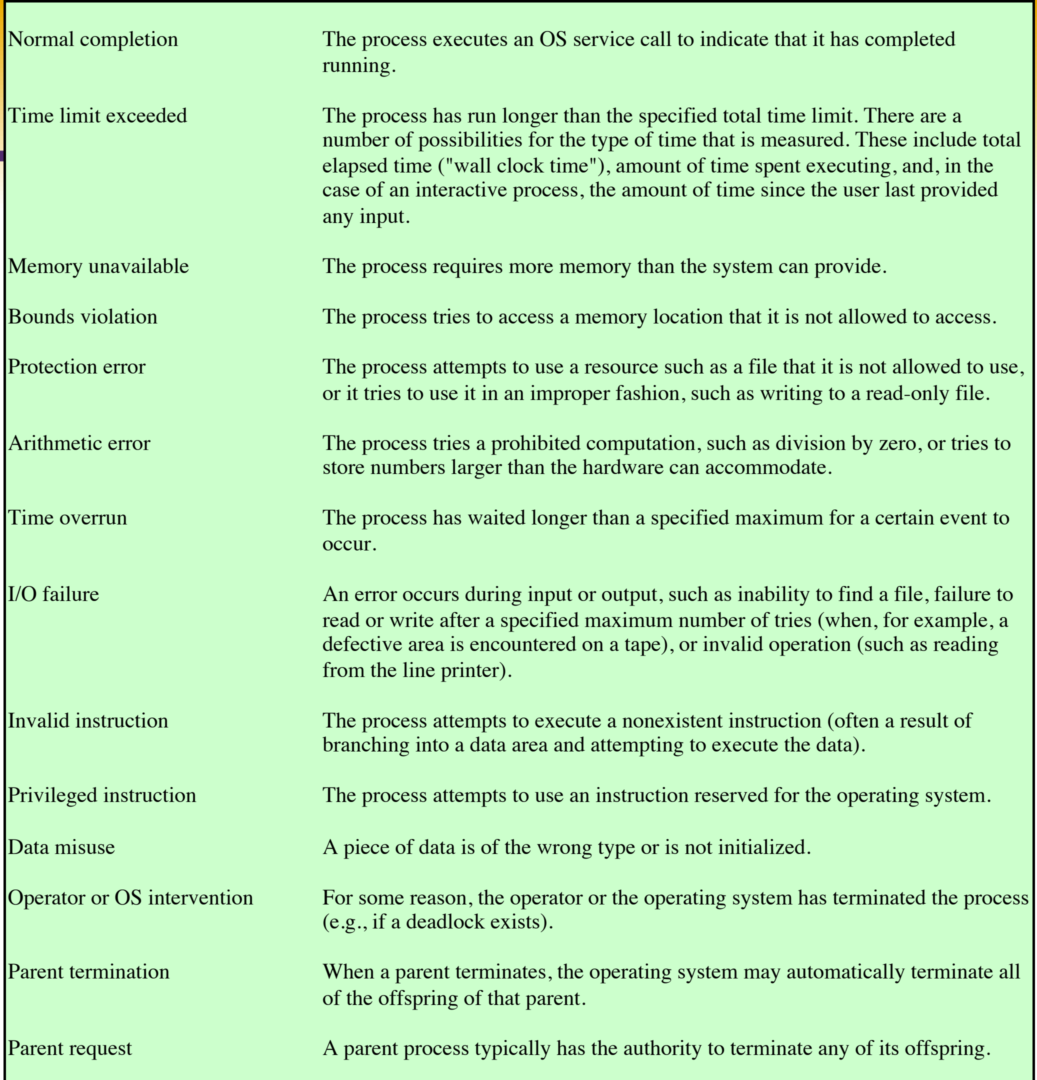
- Is the new process

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

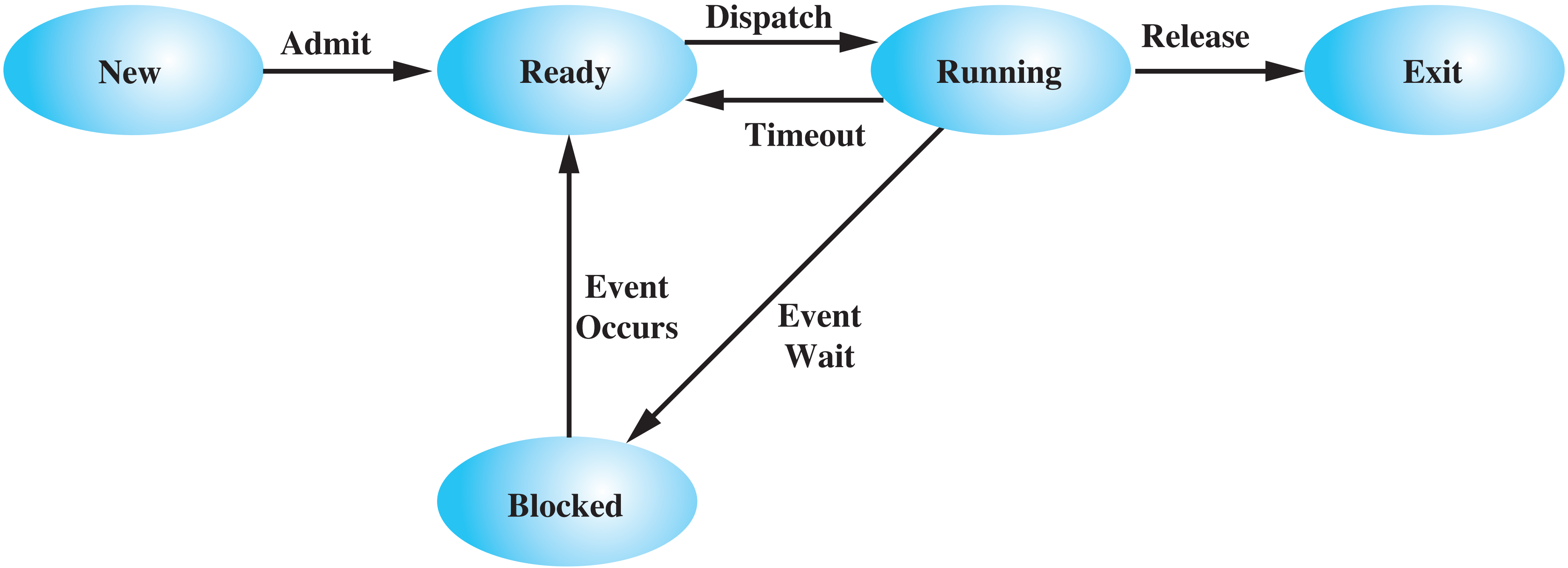
Reasons

- Possible reasons:
  - Normal Completion
  - Time limit exceeded
  - Memory unavailable
  - Parent termination
  - ….

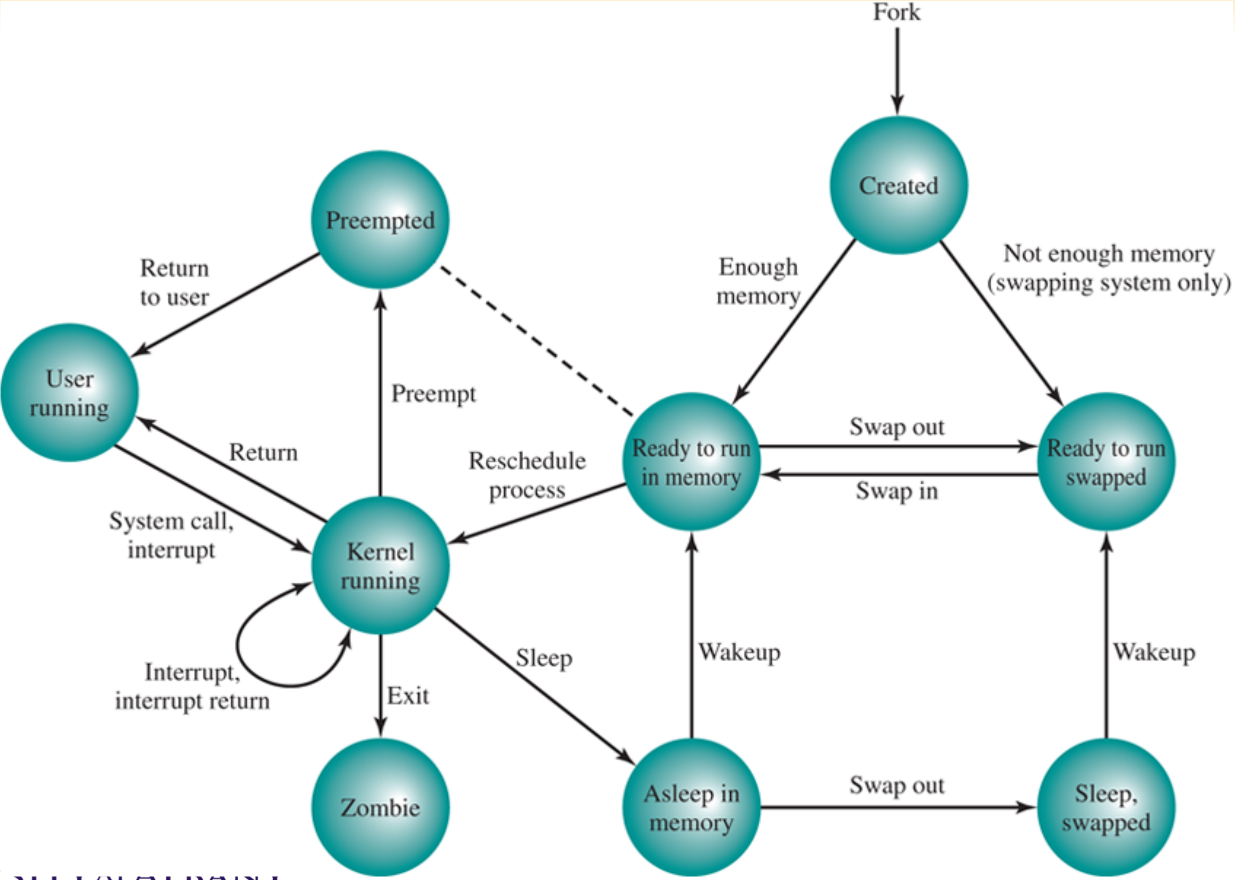


Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

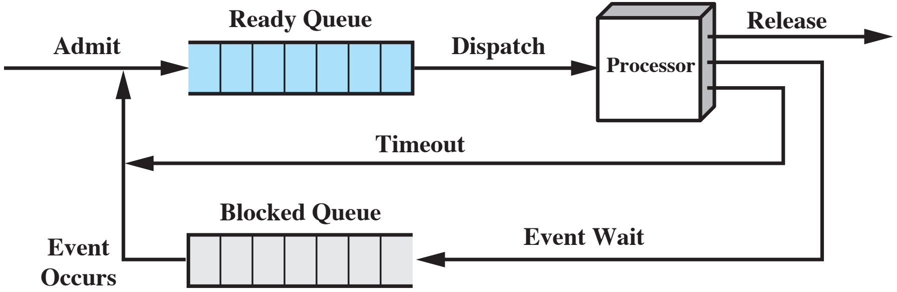
Five State Process Model

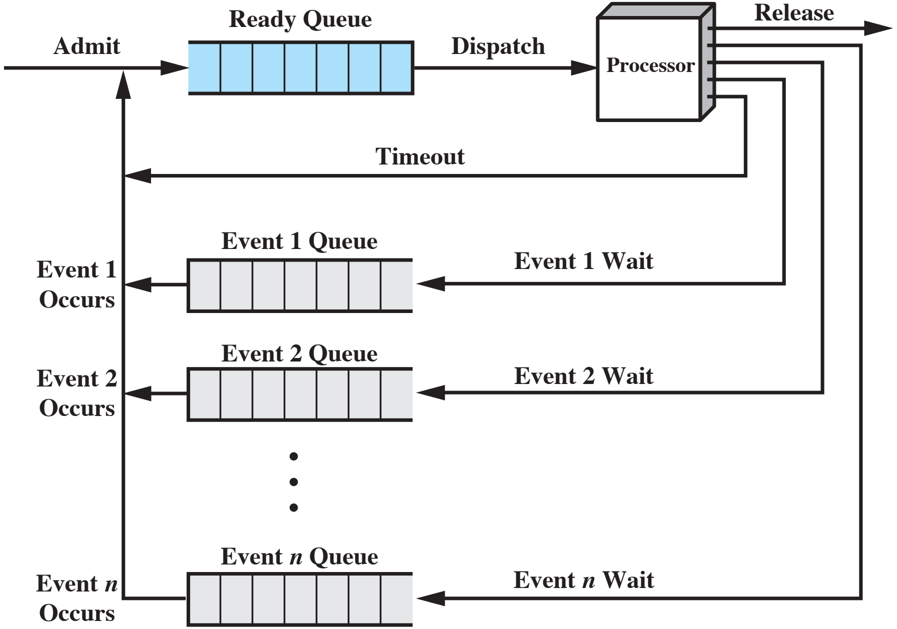


UNIX Process State Transition Diagram



Queuing Model





Single Blocked Queue

Multiple Blocked Queue

Processes Characteristics

- Resource Ownership
  - Process includes a virtual address space to hold the process image
  - The OS performs a protection function to prevent unwanted interference between processes with respect to resources
- Scheduling / Execution
  - Follows an execution path that may be interleaved with other processes
  - A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

Multiple Process Handling

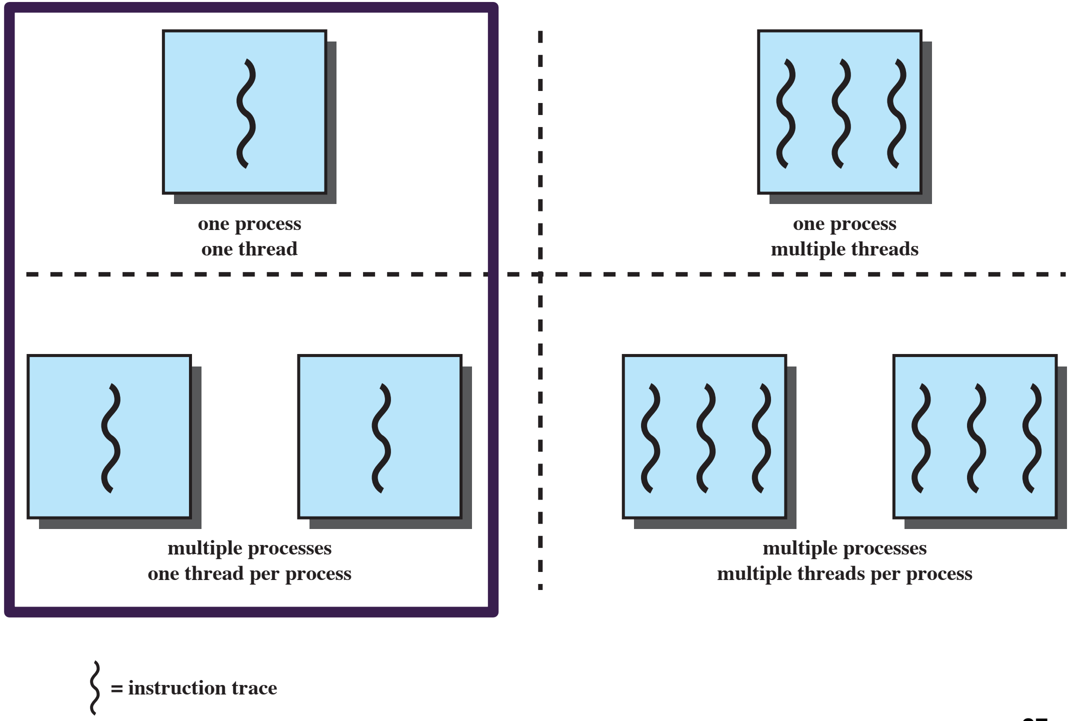
- Remote Procedure Call (RPC)
- MPI (Message Passing Interface)

Process & Thread

- The unit of dispatching is referred to as a *thread* or *lightweight process*
- The unit of resource ownership is referred to as a *process* or *task*
- *Multithreading* **-** The ability of an OS to support multiple, concurrent paths of execution within a single process

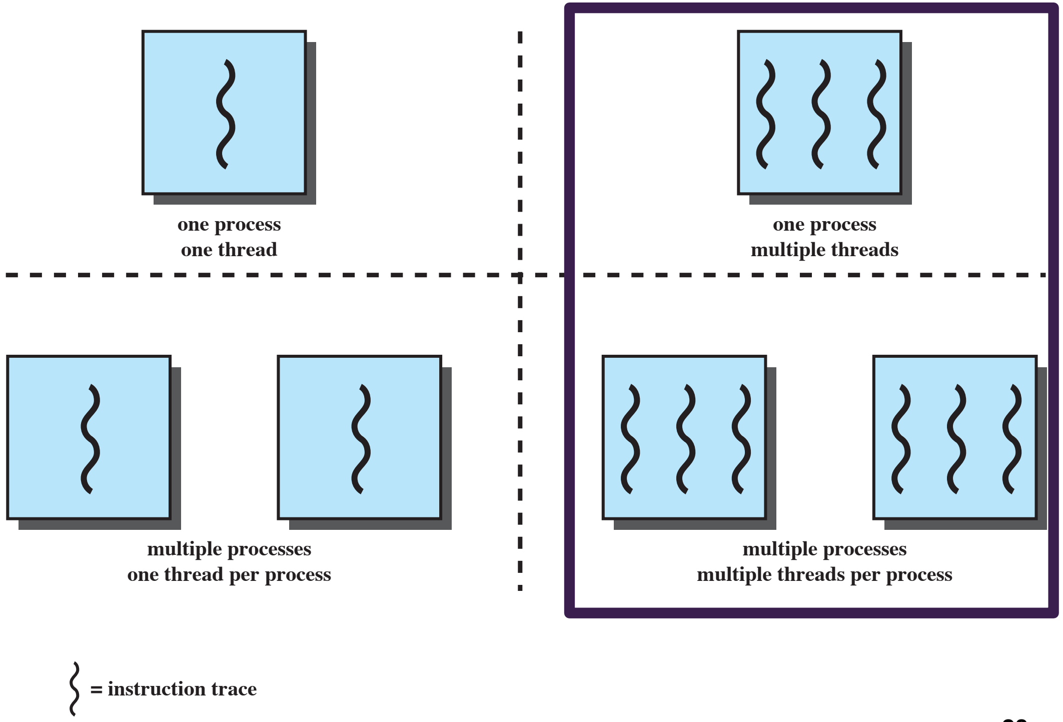
Single Threaded Approach

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- Example: MS-DOS

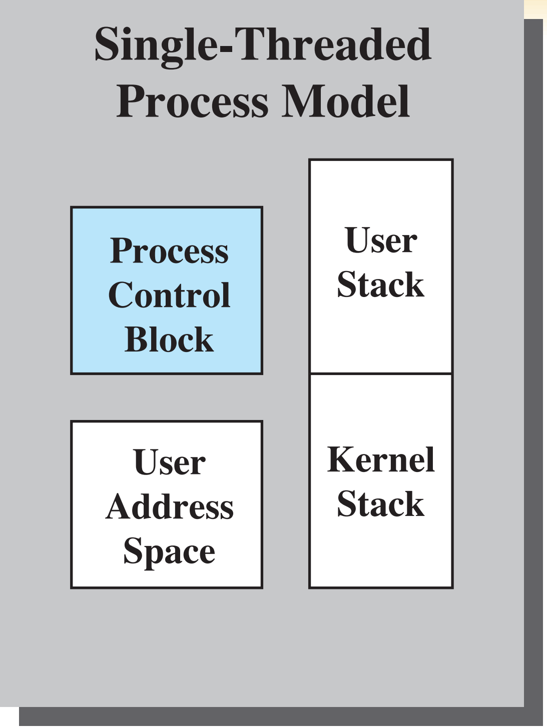


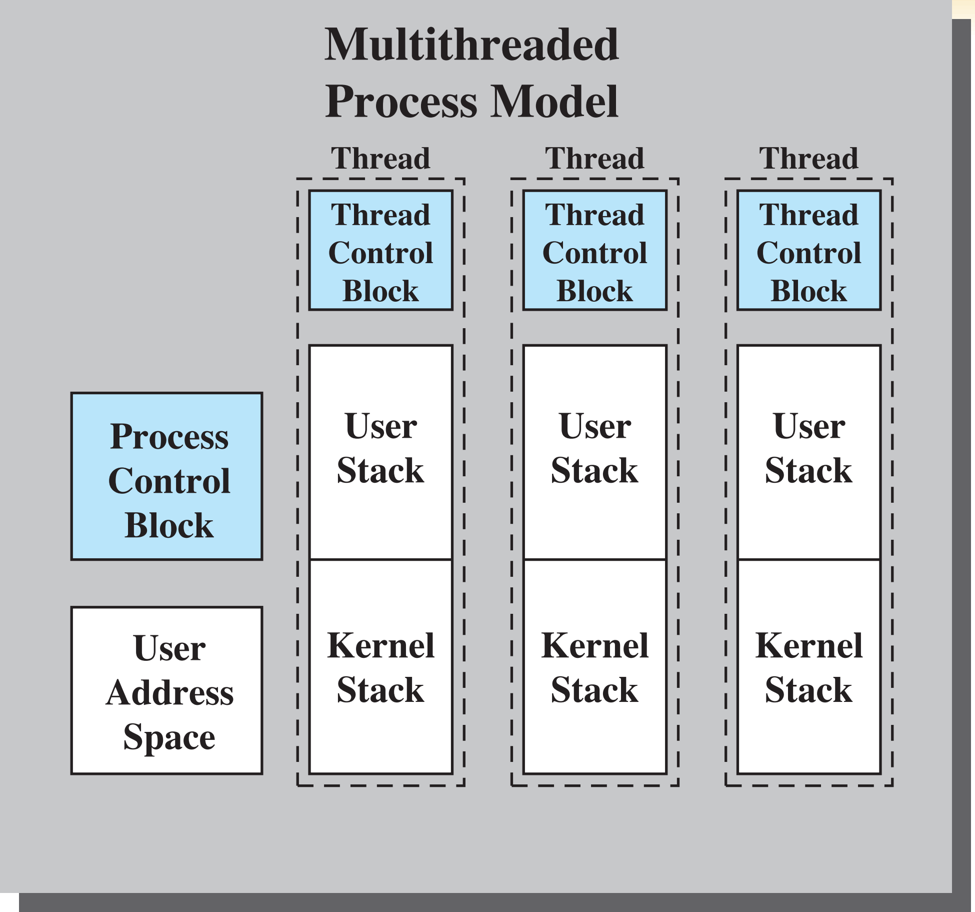
Multiple Threaded Approach

- Multiple Threads per process
- One process multiple threads (Java Runtime)
- Multiple processes each with multiple threads (UNIX, Windows, Solaris)

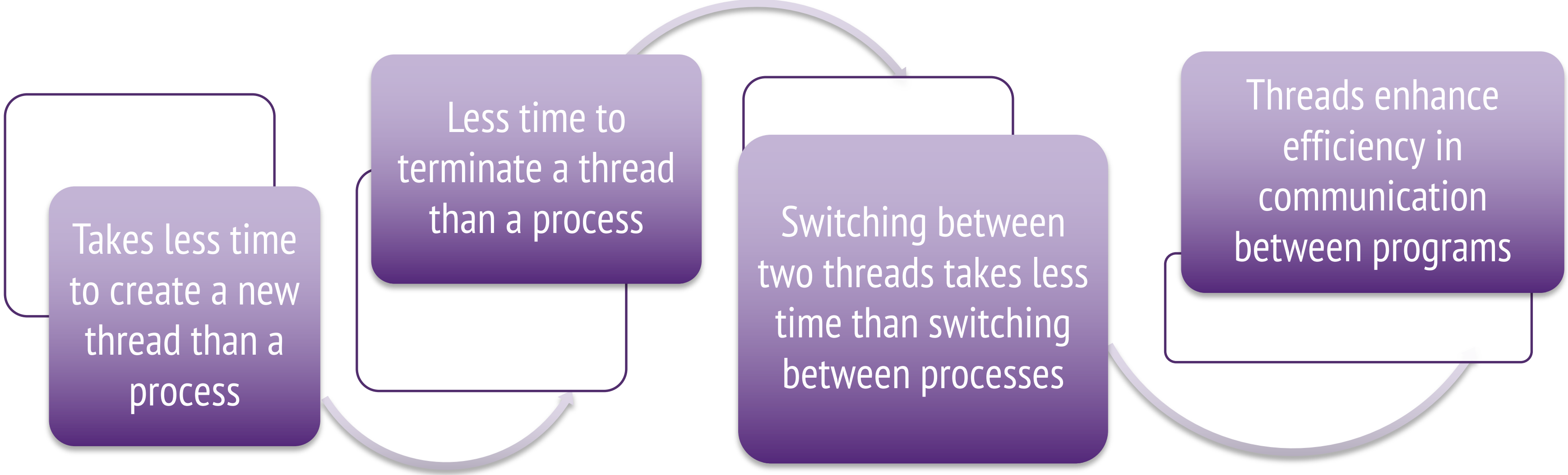


Process Model





Benefits of Thread



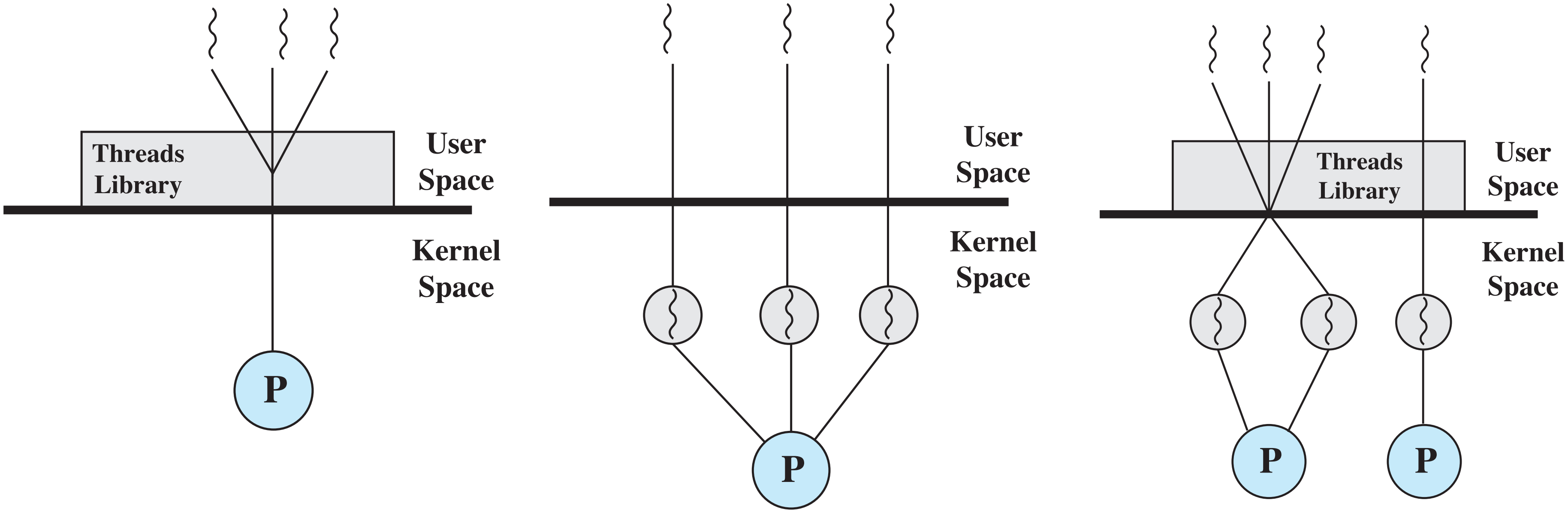
Takes less time to create a new thread than a process

Less time to terminate a thread than a process

Switching between two threads takes less time than switching between processes

Threads enhance efficiency in communication between programs

User and Kernel level Threads



(a) Pure user-level

(b) Pure kernel-level

(c) Combined

Thread Creation

- Prototype:
  - int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *arg);

    - *tid*: an unsigned long integer that indicates a threads id
    - *tattr*: attributes of the thread – usually NULL
    - *start_routine*: the name of the function the thread starts executing
    - *arg*: the argument to be passed to the start routine – only one
  - after this function gets executed, a new thread has been created and is executing the function indicated by *start_routine*

Waiting for a Thread

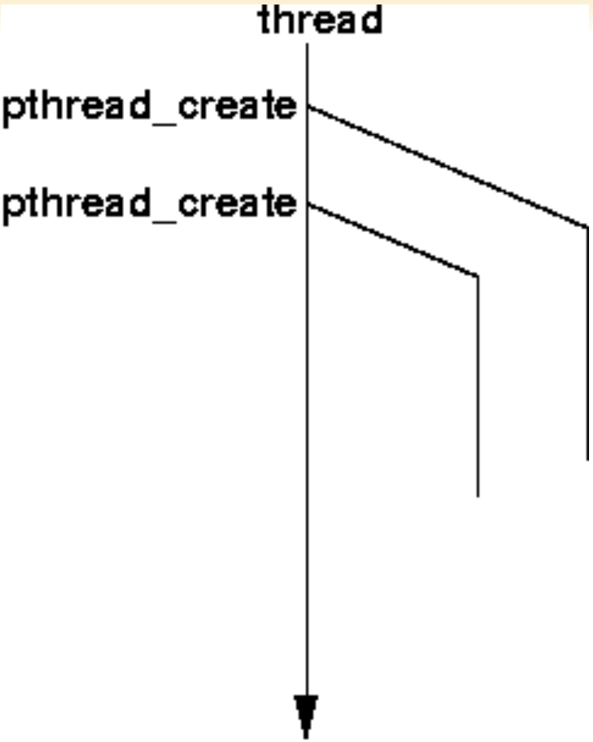
- Prototype:
  - int pthread_join(thread_t tid, void **status);
    - *tid*: identification of the thread to wait for
    - *status:* the exit status of the terminating thread – can be NULL
  - the thread that calls this function blocks its own execution until the thread indicated by *tid* terminates its execution
    - finishes the function it started with or
    - issues a *pthread_exit()* command – more on this in a minute

Exiting a Thread

- pthreads exist in user space and are seen by the kernel as a single process
  - if one issues and *exit()* system call, all the threads are terminated by the OS
  - if the *main()* function exits, all of the other threads are terminated
- To have a thread exit, use *pthread_exit()*
- Prototype:
  - void pthread_exit(void *status);
    - *status:* the exit status of the thread – passed to the *status* variable in the *pthread_join()* function of a thread waiting for this one

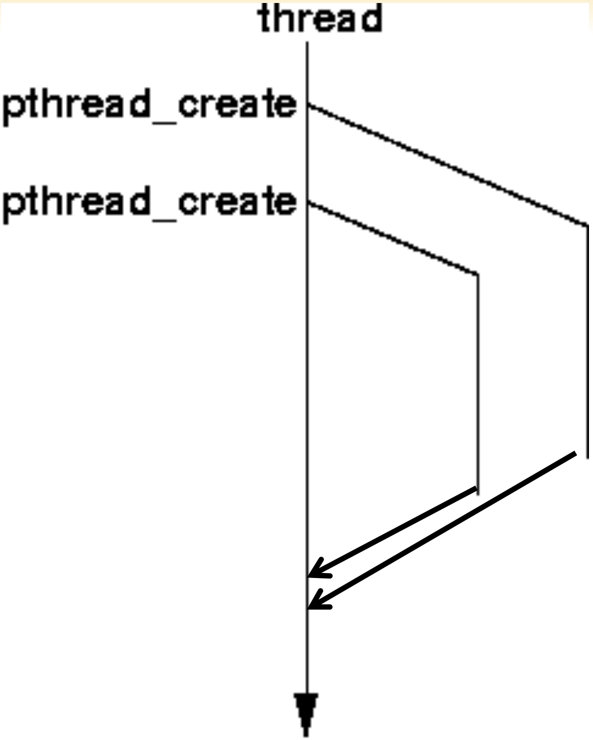
Create Thread

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
   printf(“Hello from thread %d\n", id);
}

void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
}
```

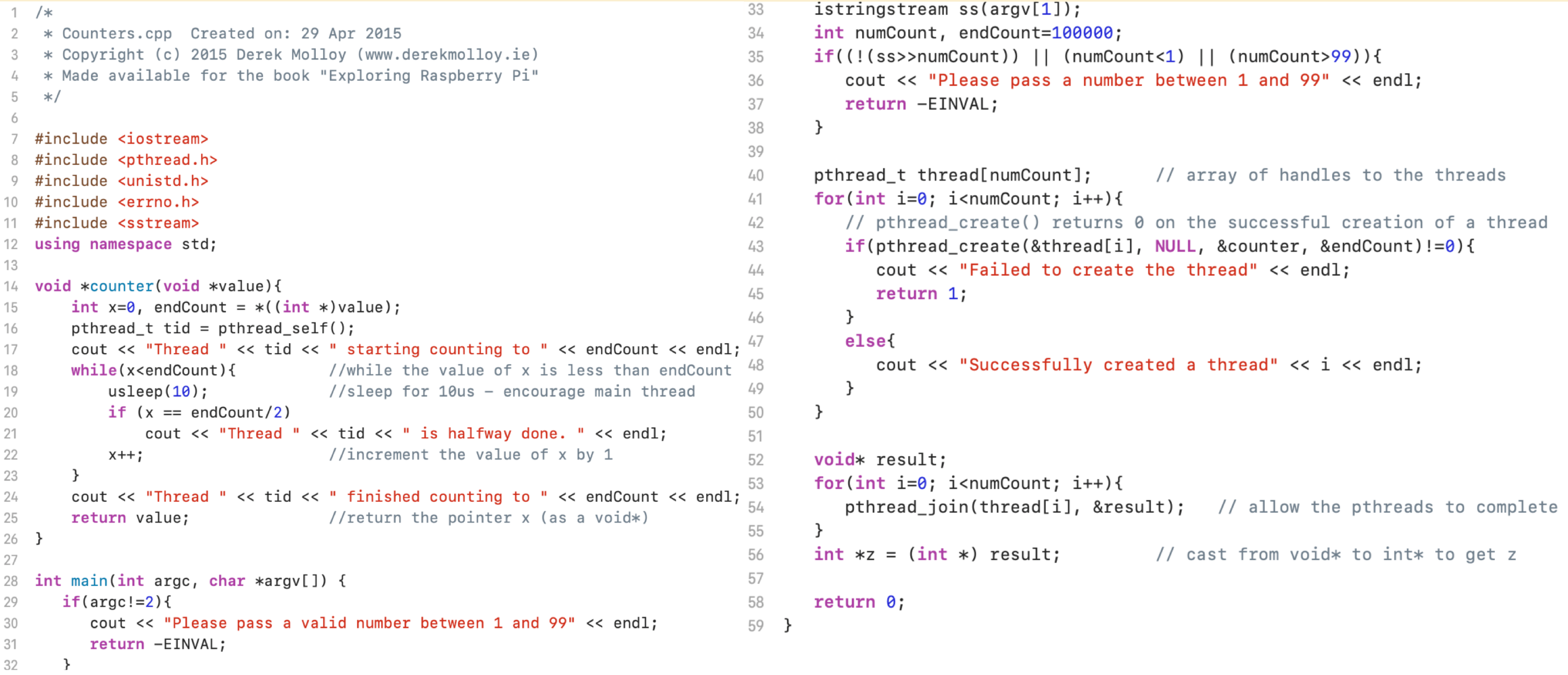


Create Thread and Join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
   printf(“Hello from thread %d\n", id);
}

void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
   pthread_join(thread0, NULL);
   pthread_join(thread1, NULL);
}
```



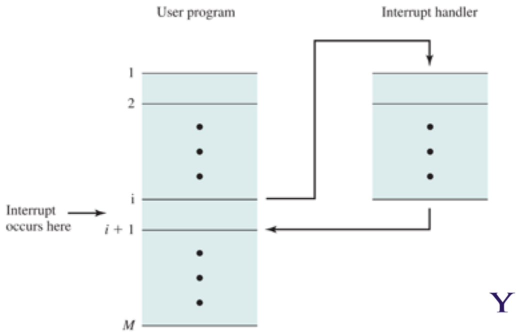
Counter Example

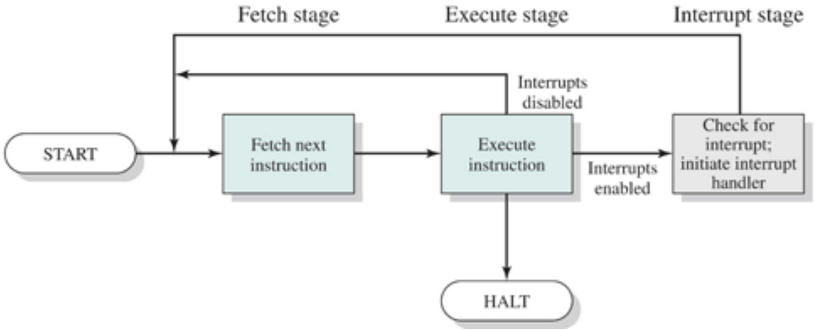


```cpp
/*
 * Counters.cpp  Created on: 29 Apr 2015
 * Copyright (c) 2015 Derek Molloy (www.derekmolloy.ie)
 * Made available for the book "Exploring Raspberry Pi"
 */

#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <sstream>
using namespace std;

void *counter(void *value){
    int x=0, endCount = *((int *)value);
    pthread_t tid = pthread_self();
    cout << "Thread " << tid << " starting counting to " << endCount << endl;
    while(x<endCount){          //while the value of x is less than endCount
        usleep(10);             //sleep for 10us - encourage main thread
        if (x == endCount/2)
            cout << "Thread " << tid << " is halfway done. " << endl;
        x++;                    //increment the value of x by 1
    }
    cout << "Thread " << tid << " finished counting to " << endCount << endl;
    return value;               //return the pointer x (as a void*)
}

int main(int argc, char *argv[]) {
   if(argc!=2){
      cout << "Please pass a valid number between 1 and 99" << endl;
      return -EINVAL;
   }
   istringstream ss(argv[1]);
   int numCount, endCount=100000;
   if((!(ss>>numCount)) || (numCount<1) || (numCount>99)){
      cout << "Please pass a number between 1 and 99" << endl;
      return -EINVAL;
   }

   pthread_t thread[numCount];      // array of handles to the threads
   for(int i=0; i<numCount; i++){
      // pthread_create() returns 0 on the successful creation of a thread
      if(pthread_create(&thread[i], NULL, &counter, &endCount)!=0){
         cout << "Failed to create the thread" << endl;
         return 1;
      }
      else{
         cout << "Successfully created a thread" << i << endl;
      }
   }

   void* result;
   for(int i=0; i<numCount; i++){
      pthread_join(thread[i], &result);   // allow the pthreads to complete
   }
   int *z = (int *) result;         // cast from void* to int* to get z

   return 0;
}
```

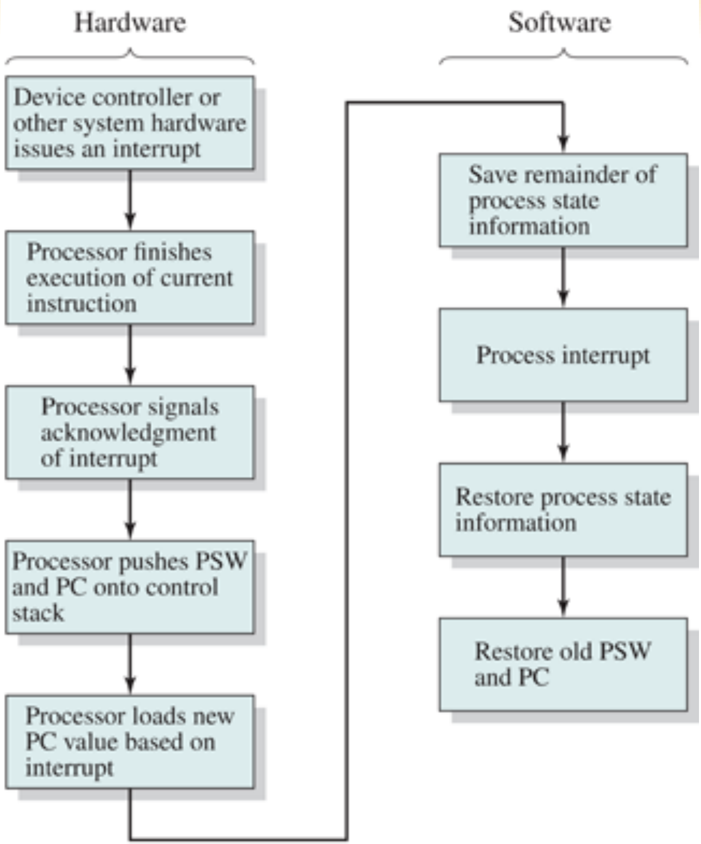
Interrupt

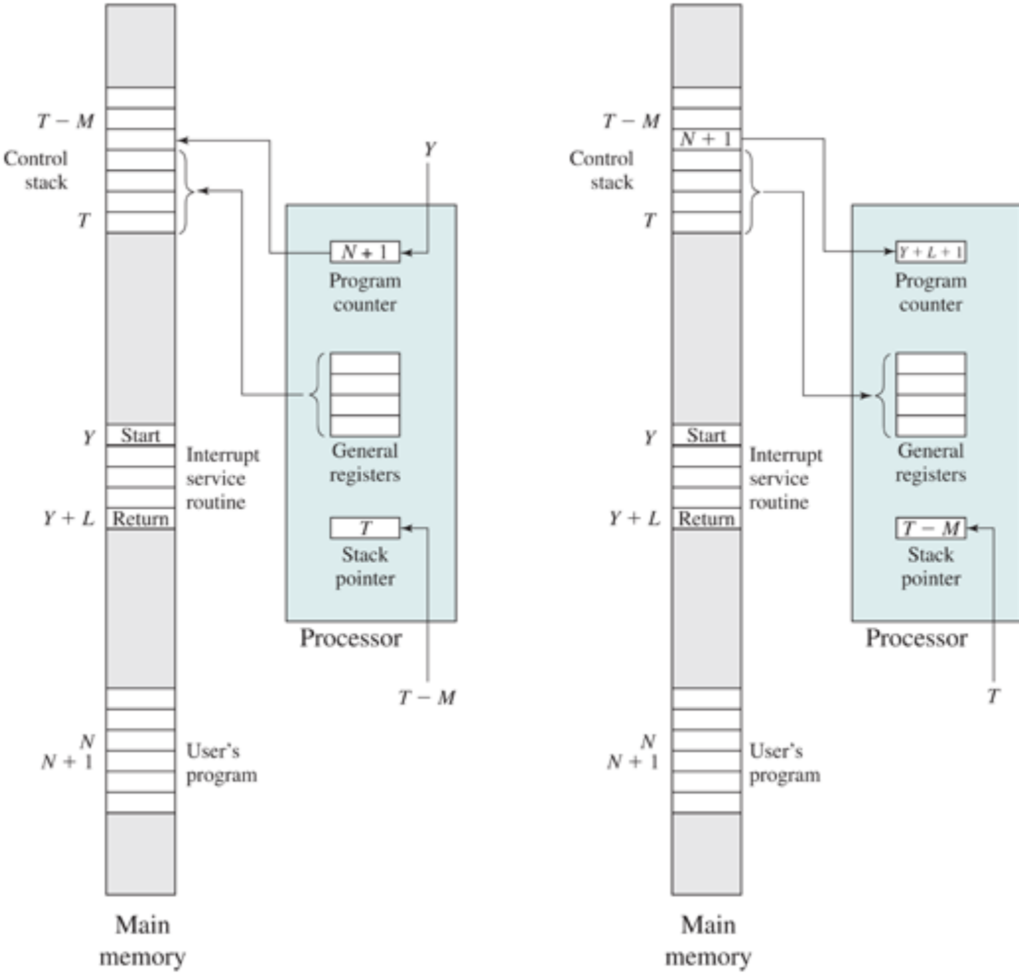
- Interrupt caused by software / hardware
- Emulates hardware interrupt in form of exception
- CPU may halt the execution of current instruction immediately
- Interrupt handler called based on interrupt priority





Interrupt Processing

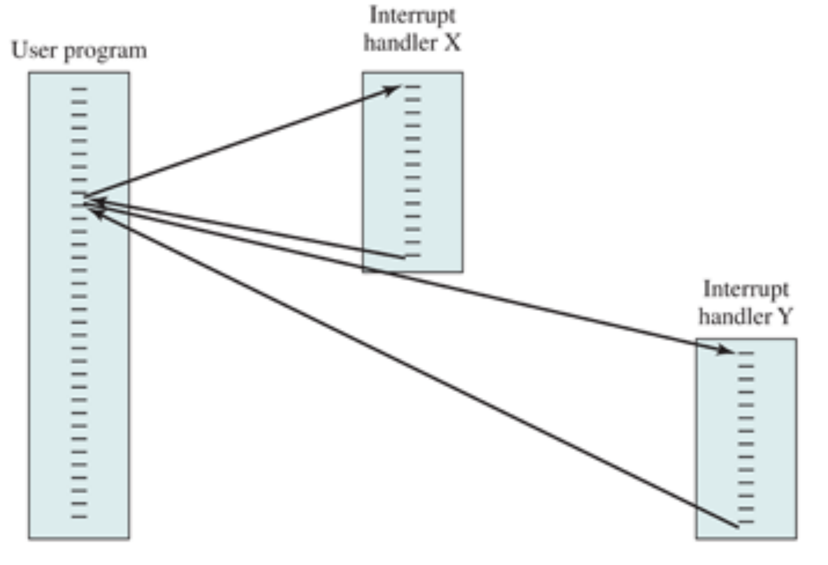

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC



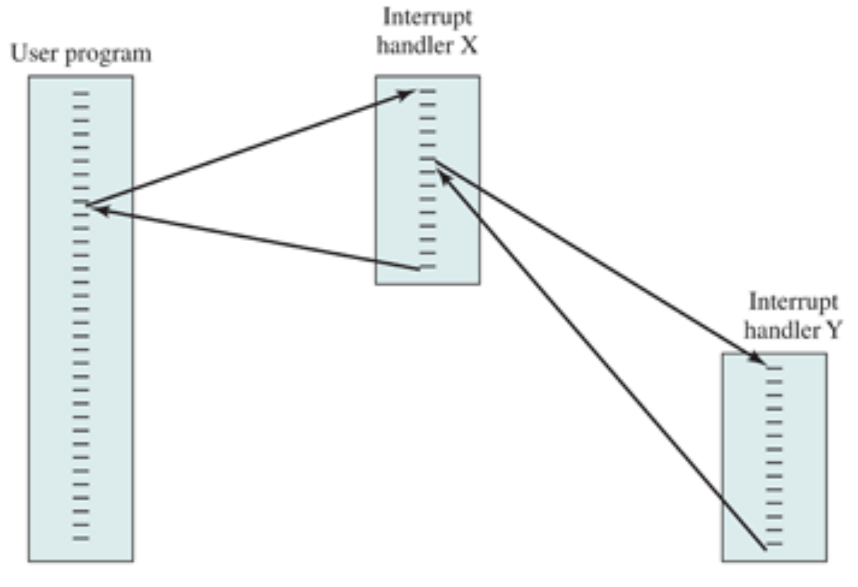
(a) Interrupt occurs after instruction at location N

(b) Return from interrupt

Interrupt Processing

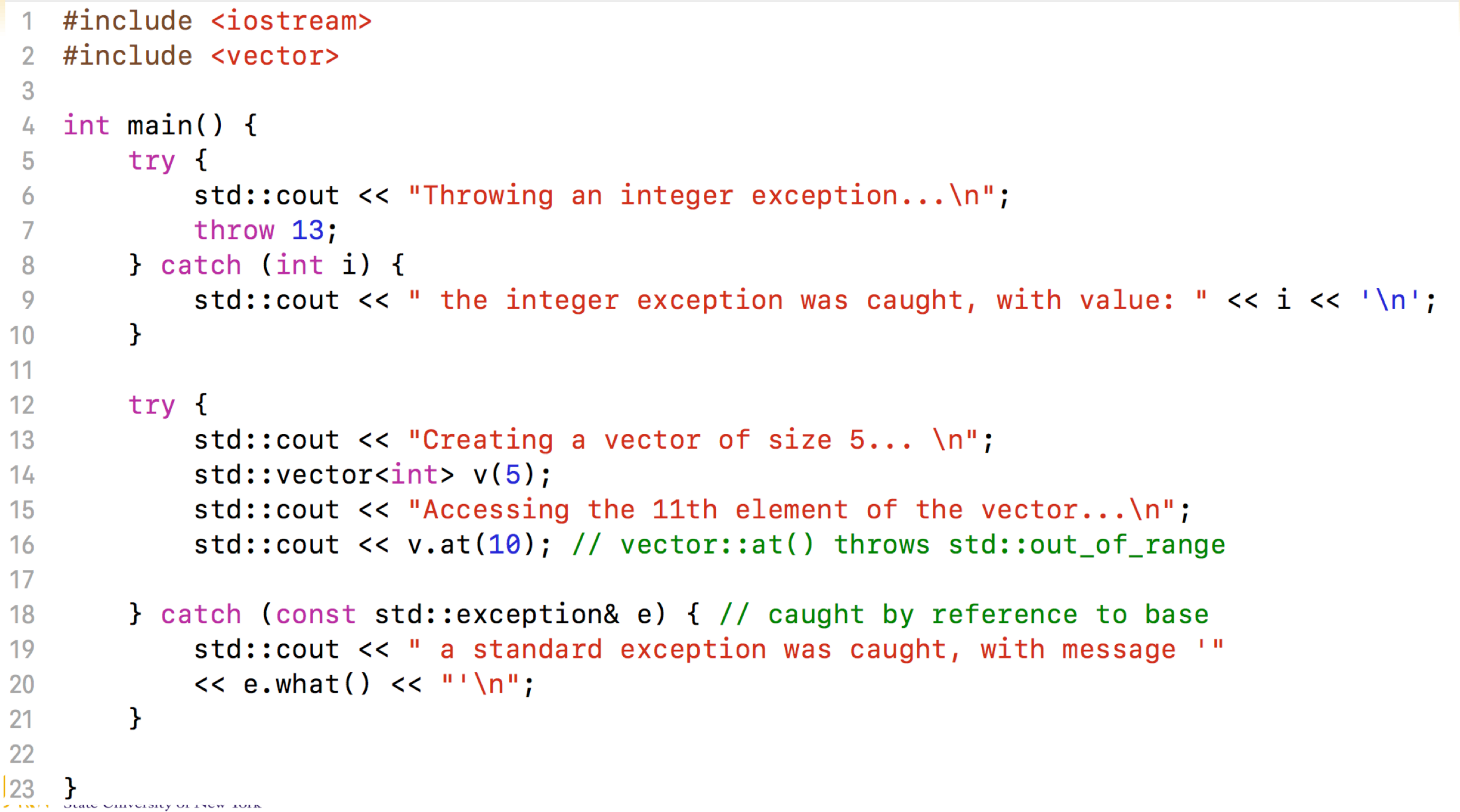


(a) Sequential interrupt processing



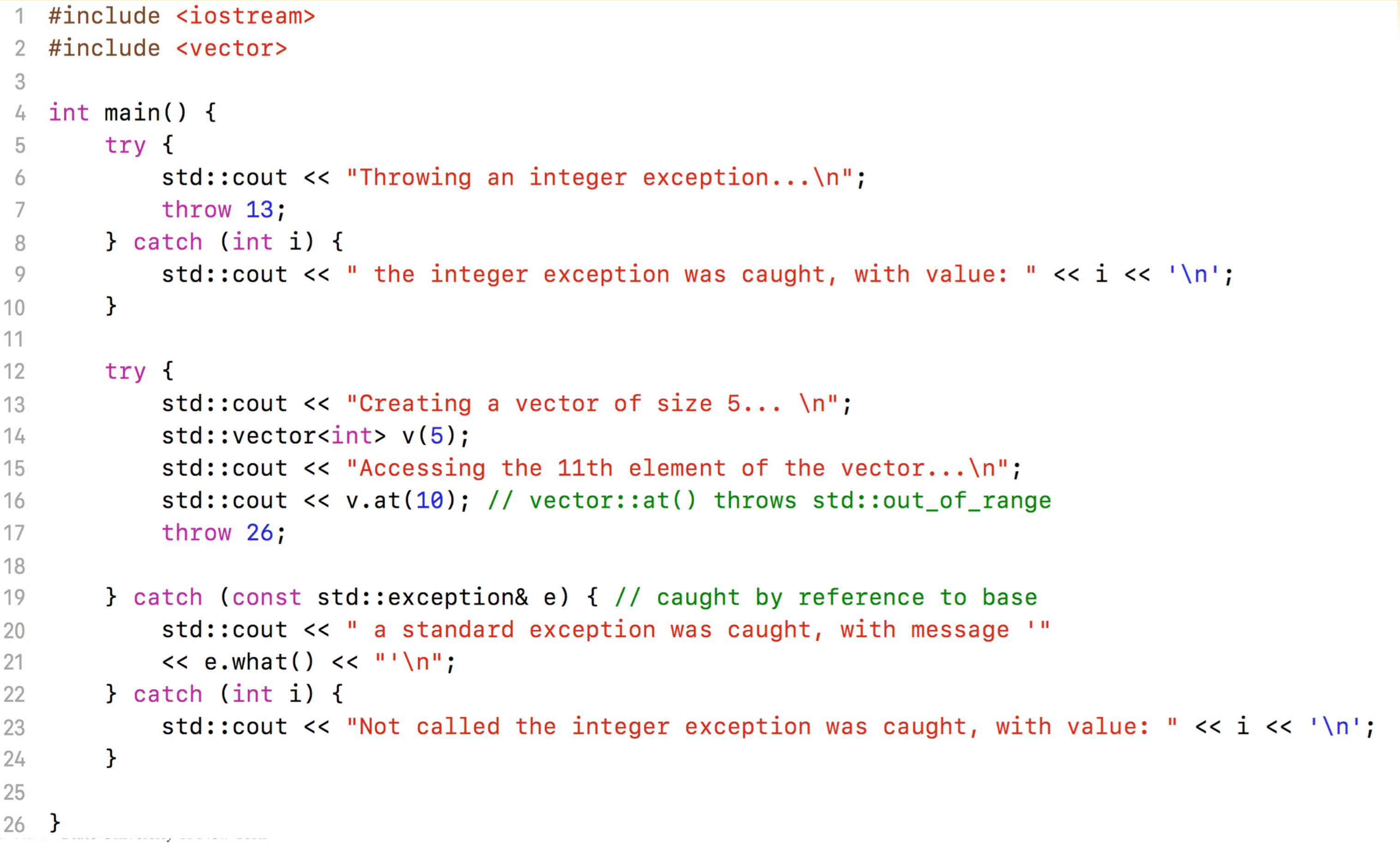
(b) Nested interrupt processing

Exception Handling



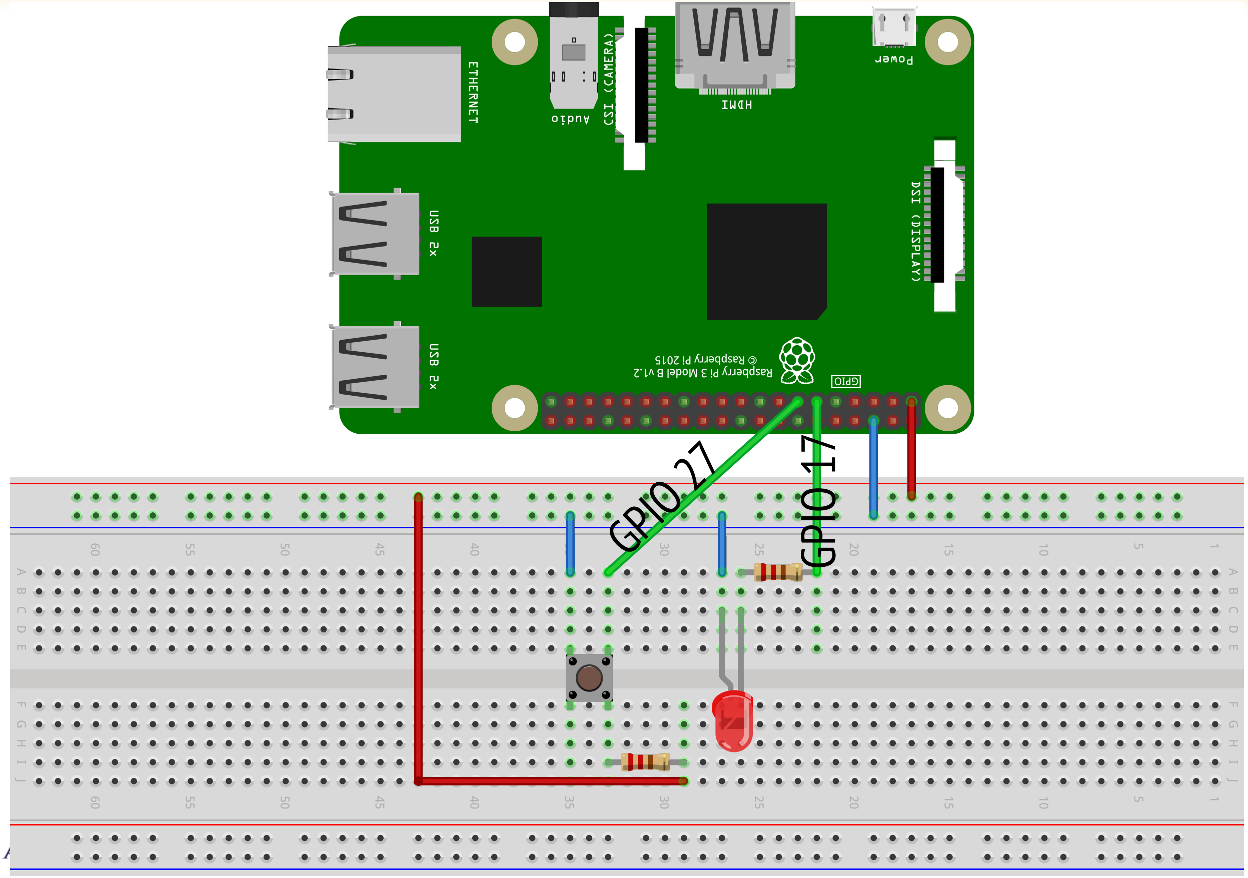
```
#include <iostream>
#include <vector>

int main() {
    try {
        std::cout << "Throwing an integer exception...\n";
        throw 13;
    } catch (int i) {
        std::cout << " the integer exception was caught, with value: " << i << '\n';
    }

    try {
        std::cout << "Creating a vector of size 5... \n";
        std::vector<int> v(5);
        std::cout << "Accessing the 11th element of the vector...\n";
        std::cout << v.at(10); // vector::at() throws std::out_of_range

    } catch (const std::exception& e) { // caught by reference to base
        std::cout << " a standard exception was caught, with message '"
        << e.what() << "'\n";
    }

}
```

Multiple Exceptions



```cpp
#include <iostream>
#include <vector>

int main() {
    try {
        std::cout << "Throwing an integer exception...\n";
        throw 13;
    } catch (int i) {
        std::cout << " the integer exception was caught, with value: " << i << '\n';
    }

    try {
        std::cout << "Creating a vector of size 5... \n";
        std::vector<int> v(5);
        std::cout << "Accessing the 11th element of the vector...\n";
        std::cout << v.at(10); // vector::at() throws std::out_of_range
        throw 26;

    } catch (const std::exception& e) { // caught by reference to base
        std::cout << " a standard exception was caught, with message '"
        << e.what() << "'\n";
    } catch (int i) {
        std::cout << "Not called the integer exception was caught, with value: " << i << '\n';
    }

}
```

Hardware Interrupt



Example Code

- exploringrpi/chp06/wiringPi/buttonLED.cpp

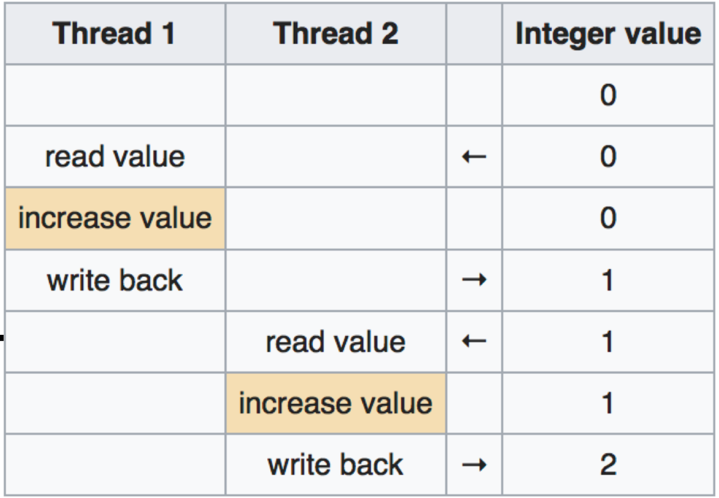
Dispatching Thread

- disable interrupts;
- determine which thread should execute (scheduling);
- if the same one, enable interrupts and return;
- save state (registers) into current thread data structure;
- save return address from the stack for current thread;
- copy new thread state into machine registers;
- replace program counter on the stack for the new thread;
- enable interrupts;
- return.

Race Condition

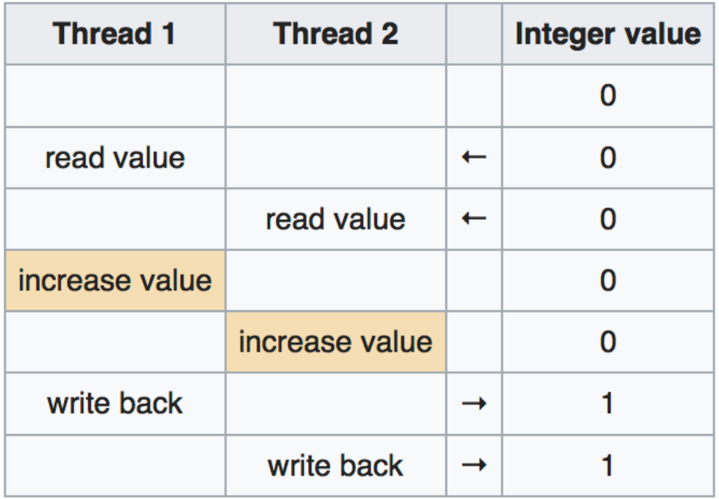
- When multiple threads read and write data items
- The final result depends on the order of execution
  - The “loser” of the race is the process that updates last and will determine the final value of the variable

Expected



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

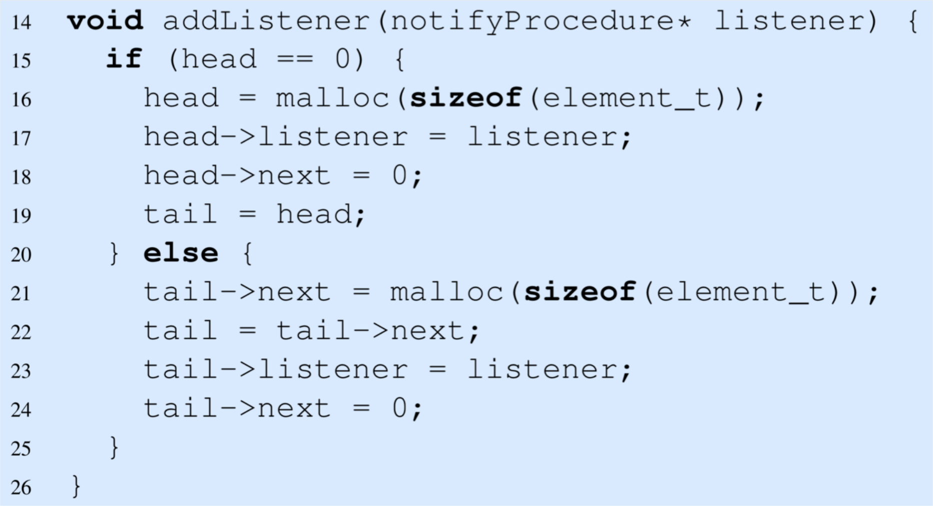
Race



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Tables from Wikipedia

Example Race Condition



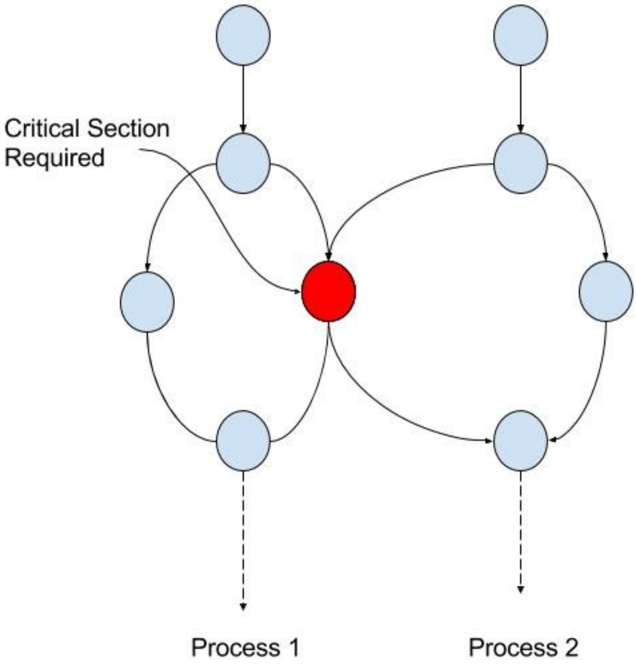
```
14  void addListener(notifyProcedure* listener) {
15    if (head == 0) {
16      head = malloc(sizeof(element_t));
17      head->listener = listener;
18      head->next = 0;
19      tail = head;
20    } else {
21      tail->next = malloc(sizeof(element_t));
22      tail = tail->next;
23      tail->listener = listener;
24      tail->next = 0;
25    }
26  }
```

- Thread 1 suspended just before line 23
- Thread 2 calls addListener
- Thread 2 changes value of tail
- Thread 1 resumes at Line 23

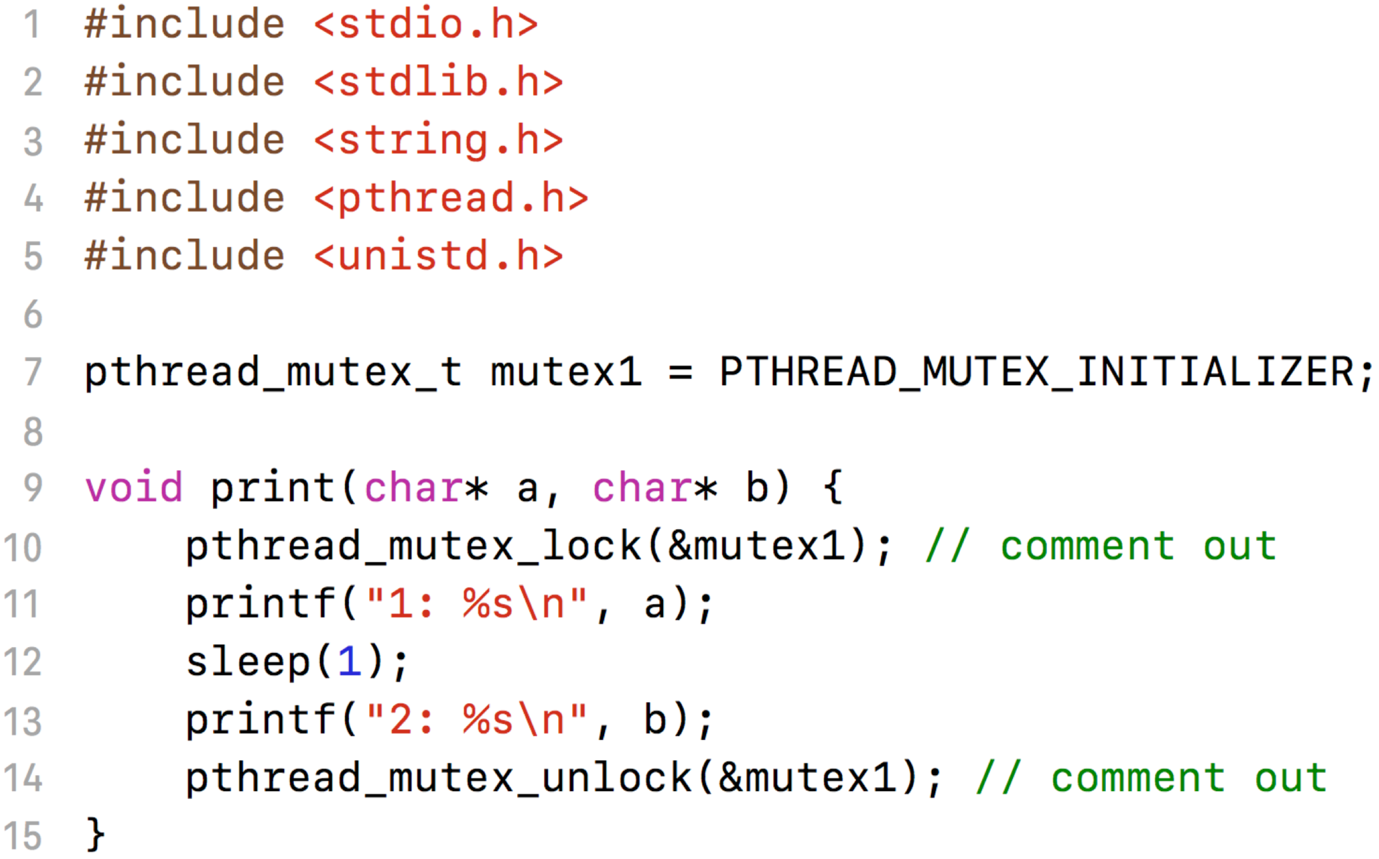
- Could result in a list with second to last element points to a random address.

Mutual Exclusion - Mutex

- Prevents Race Condition
- Enables resource sharing
- Critical section is performed by a single process or thread
- One thread blocks a critical section by using locking technique (mutex)
- Other threads have to wait to get their turn to enter into the section.

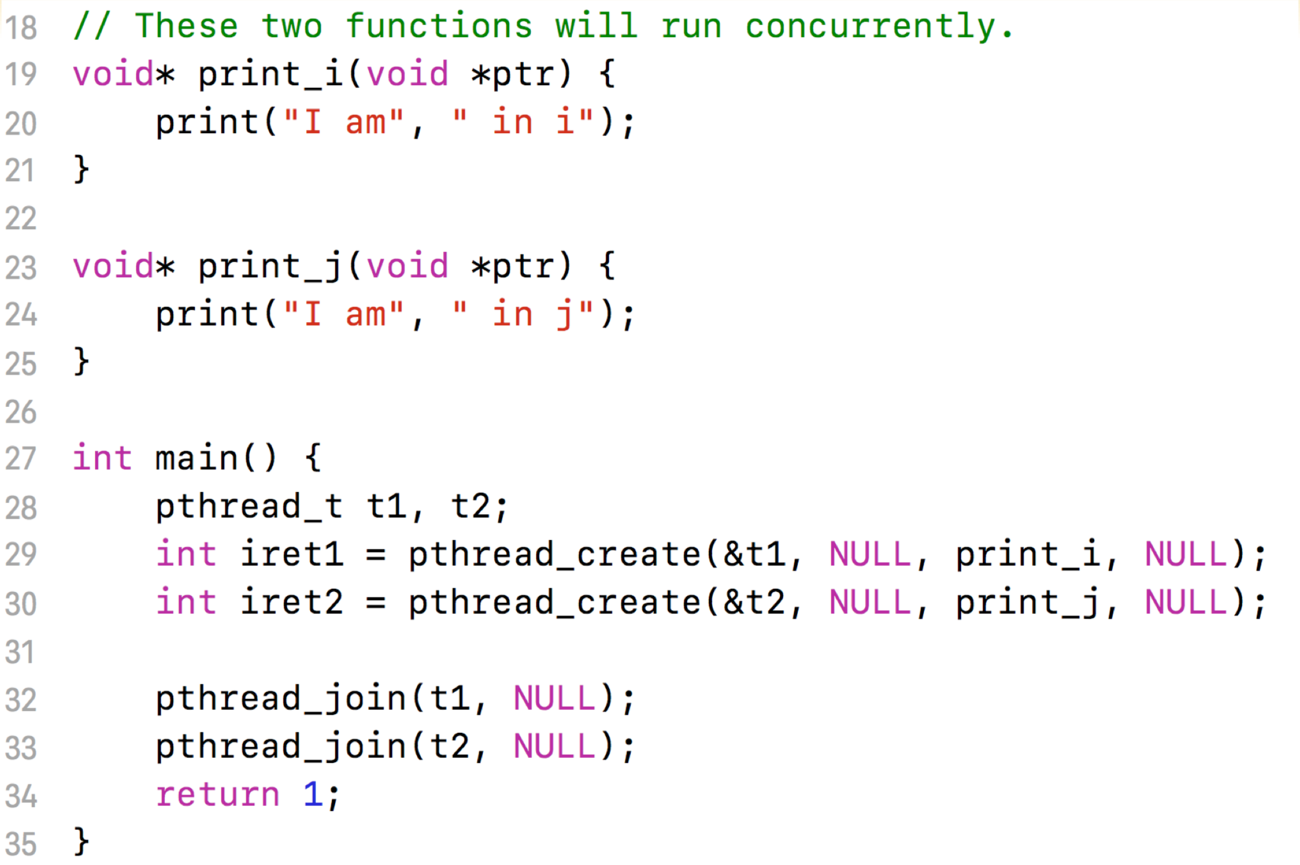


pthread_mutex()



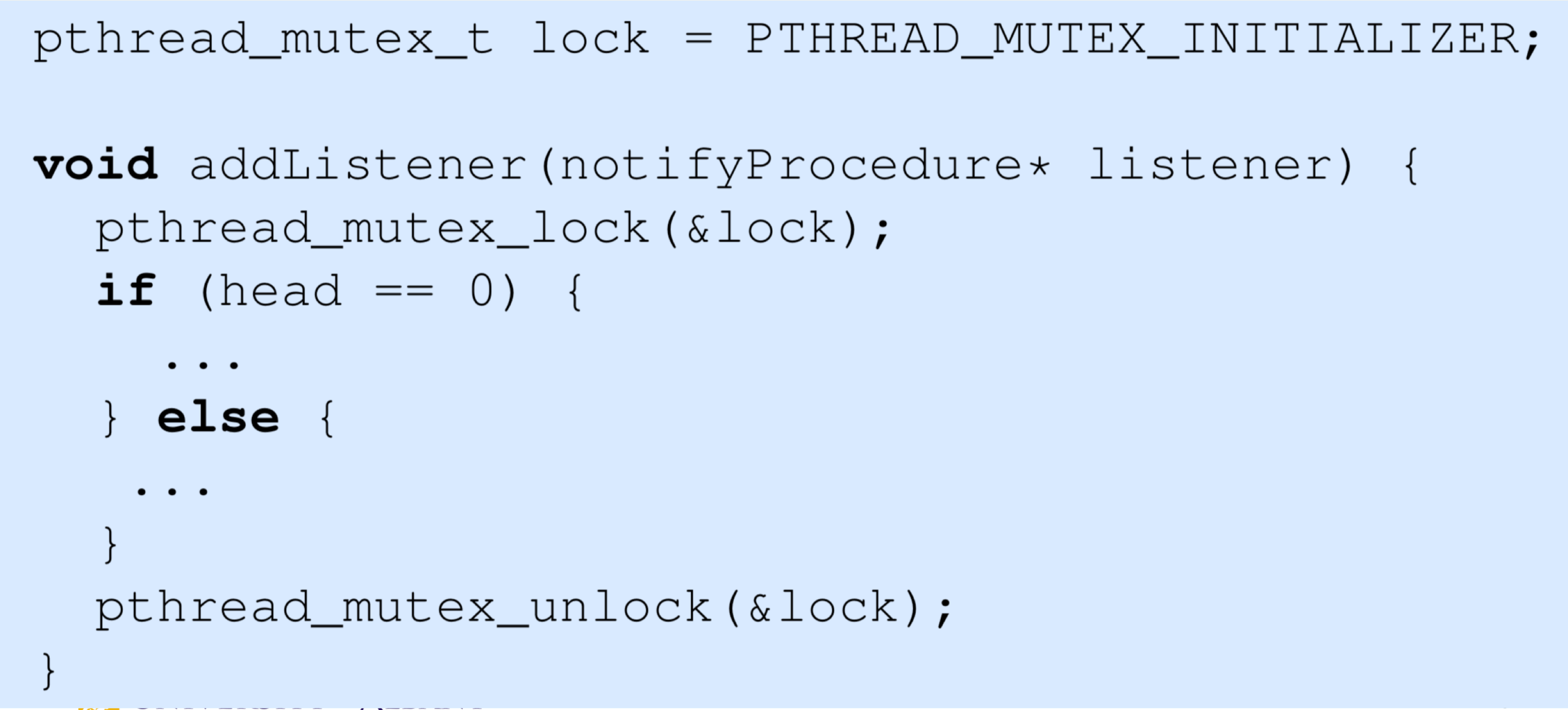
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void print(char* a, char* b) {
    pthread_mutex_lock(&mutex1); // comment out
    printf("1: %s\n", a);
    sleep(1);
    printf("2: %s\n", b);
    pthread_mutex_unlock(&mutex1); // comment out
}
```

pthread_mutex()



```c
// These two functions will run concurrently.
void* print_i(void *ptr) {
    print("I am", " in i");
}

void* print_j(void *ptr) {
    print("I am", " in j");
}

int main() {
    pthread_t t1, t2;
    int iret1 = pthread_create(&t1, NULL, print_i, NULL);
    int iret2 = pthread_create(&t2, NULL, print_j, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 1;
}
```

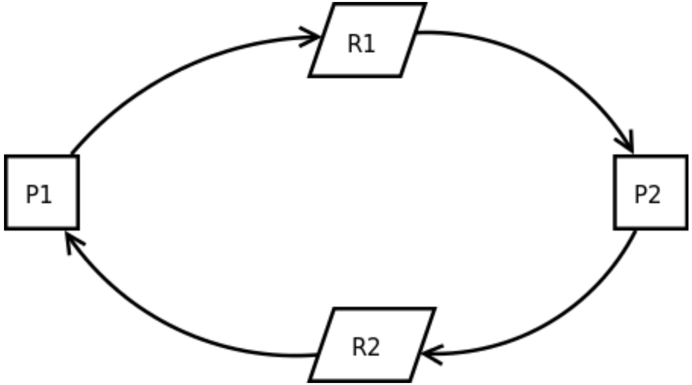
Mutex for Listener code



```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void addListener(notifyProcedure* listener) {
  pthread_mutex_lock(&lock);
  if (head == 0) {
    ...
  } else {
   ...
  }
  pthread_mutex_unlock(&lock);
}
```

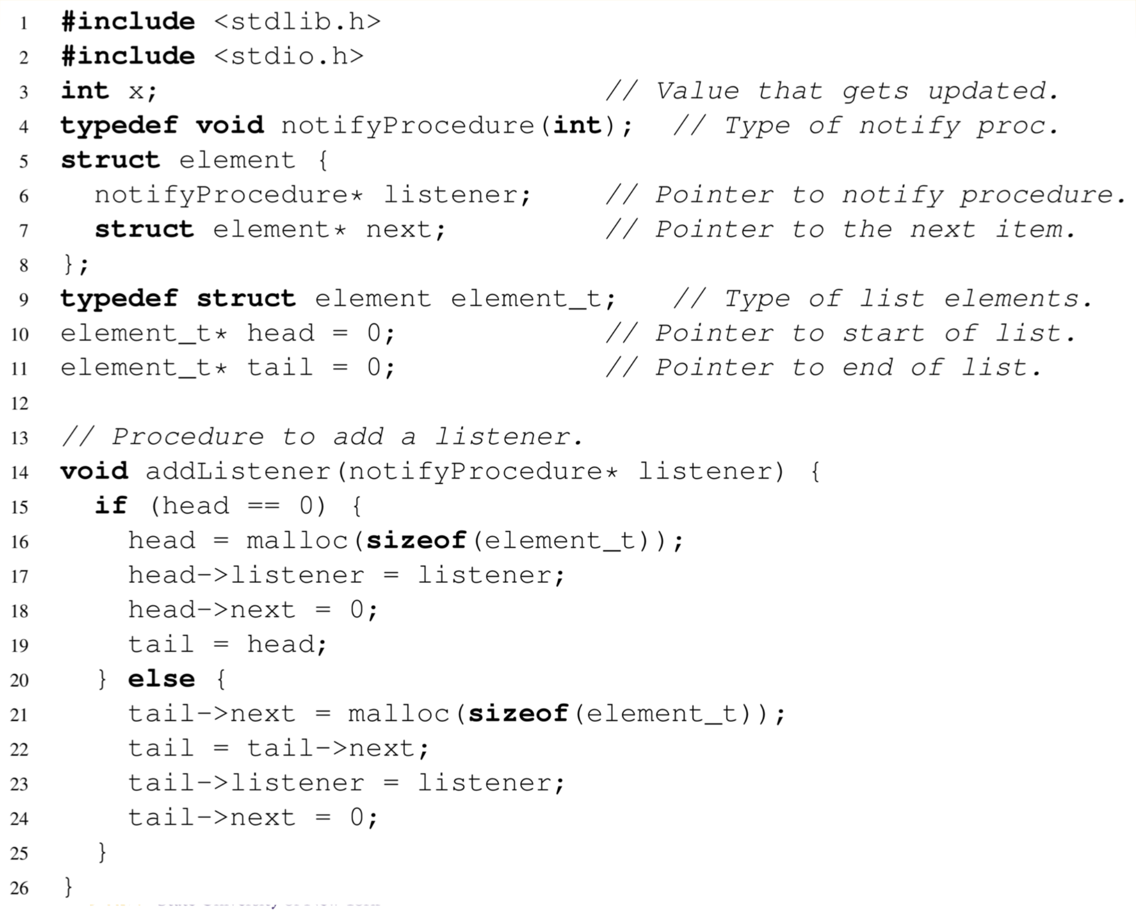
Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

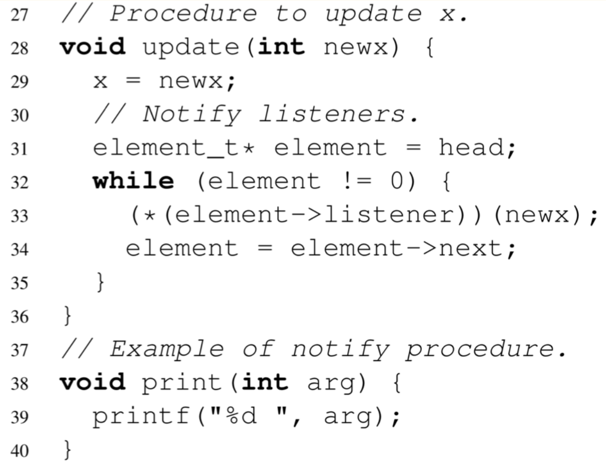


- Example: addListener() and update()

List Data Structure in C

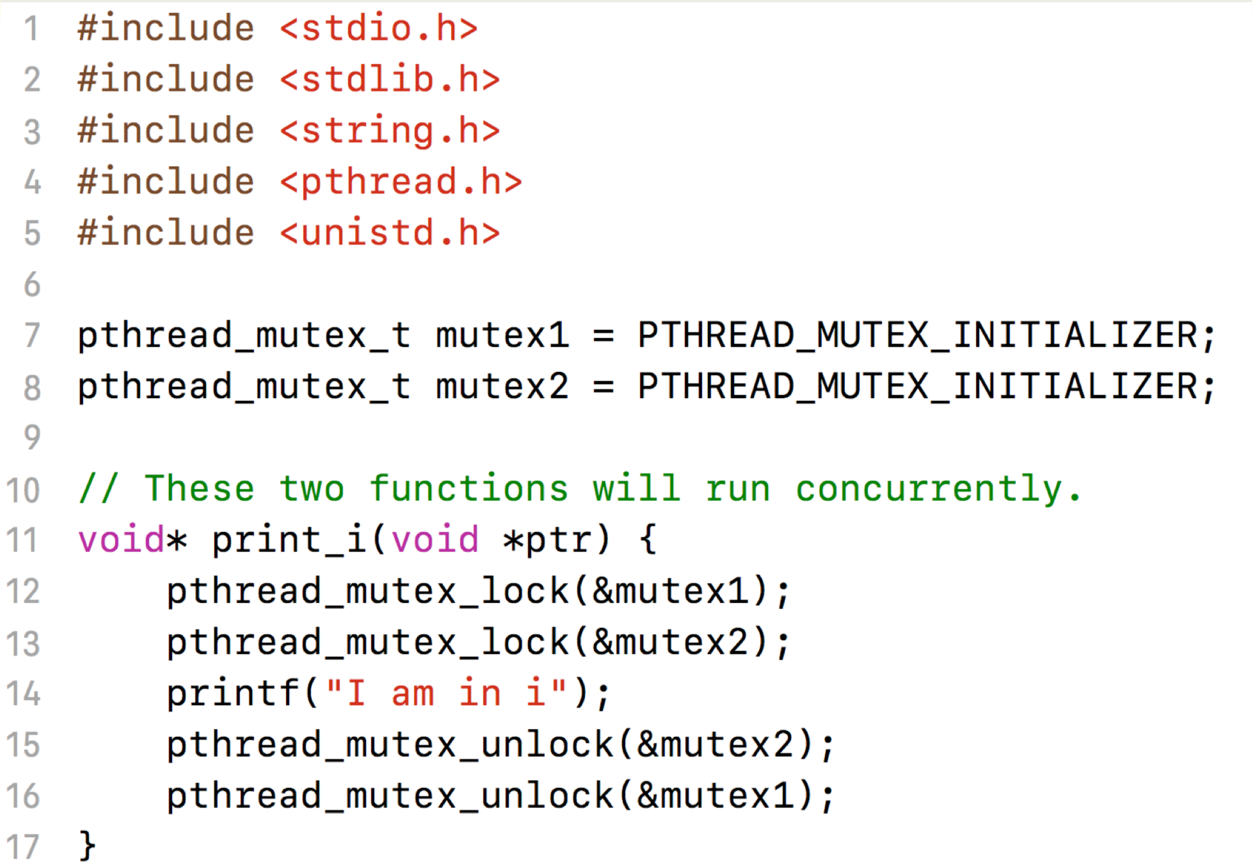


```c
#include <stdlib.h>
#include <stdio.h>
int x;                           // Value that gets updated.
typedef void notifyProcedure(int);  // Type of notify proc.
struct element {
  notifyProcedure* listener;     // Pointer to notify procedure.
  struct element* next;          // Pointer to the next item.
};
typedef struct element element_t;   // Type of list elements.
element_t* head = 0;             // Pointer to start of list.
element_t* tail = 0;             // Pointer to end of list.

// Procedure to add a listener.
void addListener(notifyProcedure* listener) {
  if (head == 0) {
    head = malloc(sizeof(element_t));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(element_t));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```



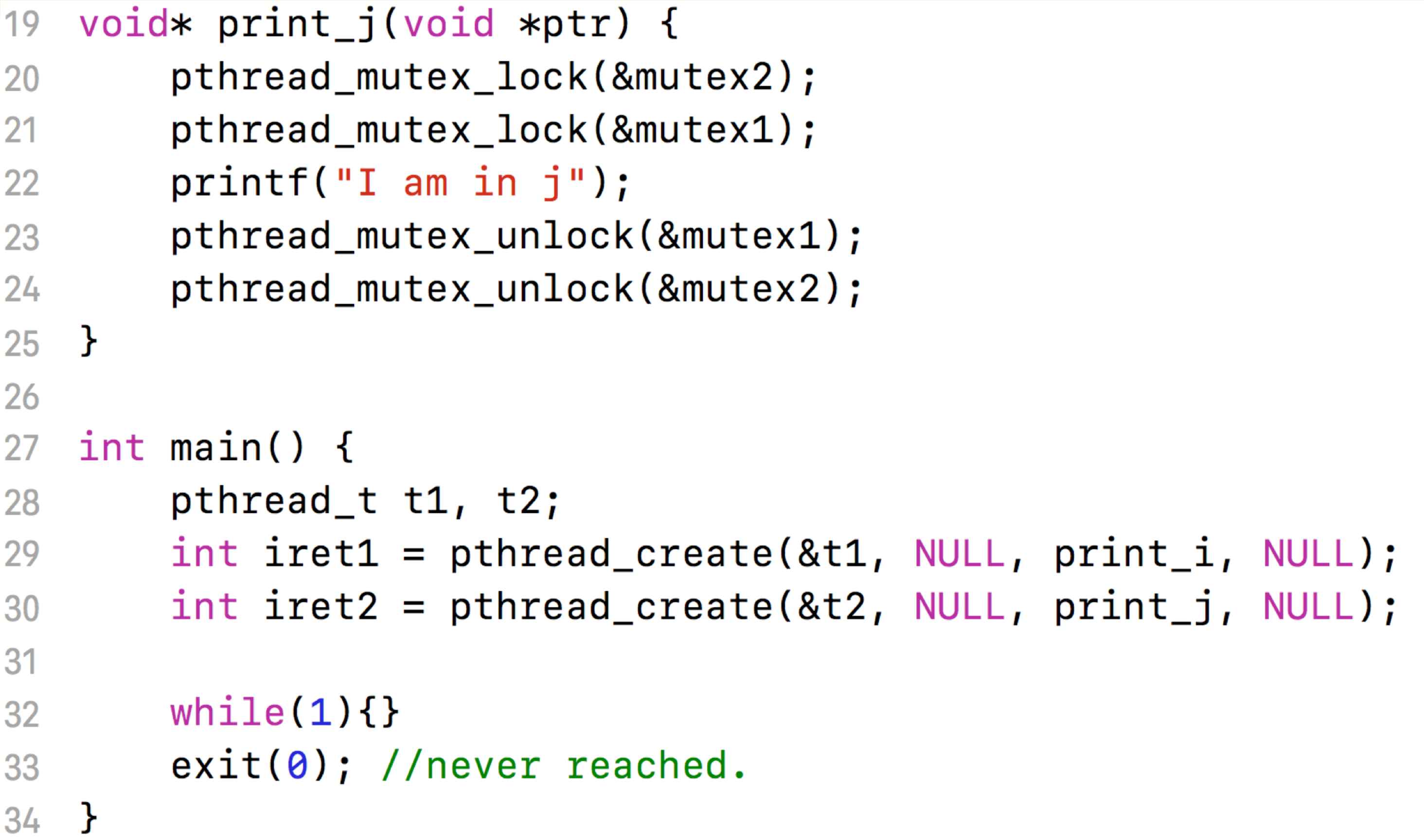
```c
// Procedure to update x.
void update(int newx) {
  x = newx;
  // Notify listeners.
  element_t* element = head;
  while (element != 0) {
    (*(element->listener))(newx);
    element = element->next;
  }
}
// Example of notify procedure.
void print(int arg) {
  printf("%d ", arg);
}
```

Deadlock Example



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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

// These two functions will run concurrently.
void* print_i(void *ptr) {
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    printf("I am in i");
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
}
```

Deadlock Example

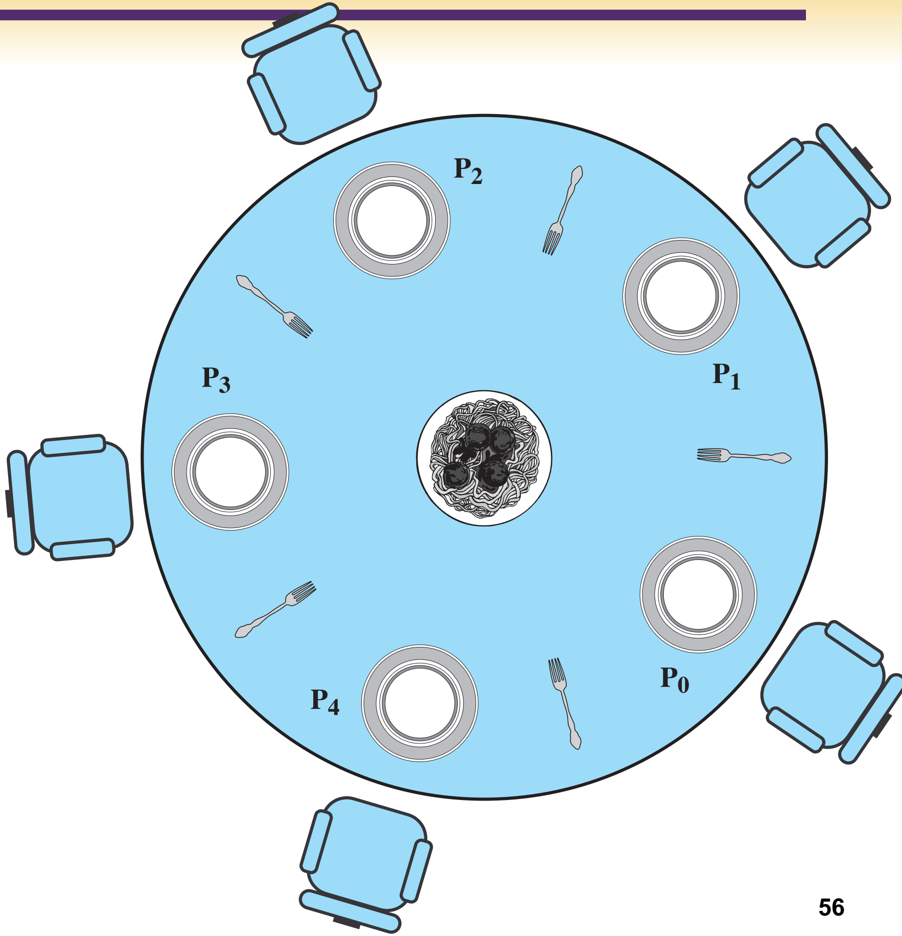


```c
void* print_j(void *ptr) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    printf("I am in j");
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
}

int main() {
    pthread_t t1, t2;
    int iret1 = pthread_create(&t1, NULL, print_i, NULL);
    int iret2 = pthread_create(&t2, NULL, print_j, NULL);

    while(1){}
    exit(0); //never reached.
}
```

Dining Philosopher's Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)

- No philosopher must starve to death (avoid deadlock and starvation)



Solution 1: Dining Philosopher’s Problem

- Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table.
- This solution leads to deadlock:
  - If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there.

Other Solutions: Dining Philosopher's Problem

- We could buy five additional forks (more resource)
- Teach the philosophers to eat spaghetti with just one fork (change algorithm).
- We could consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks. (Semaphores)

Semaphore

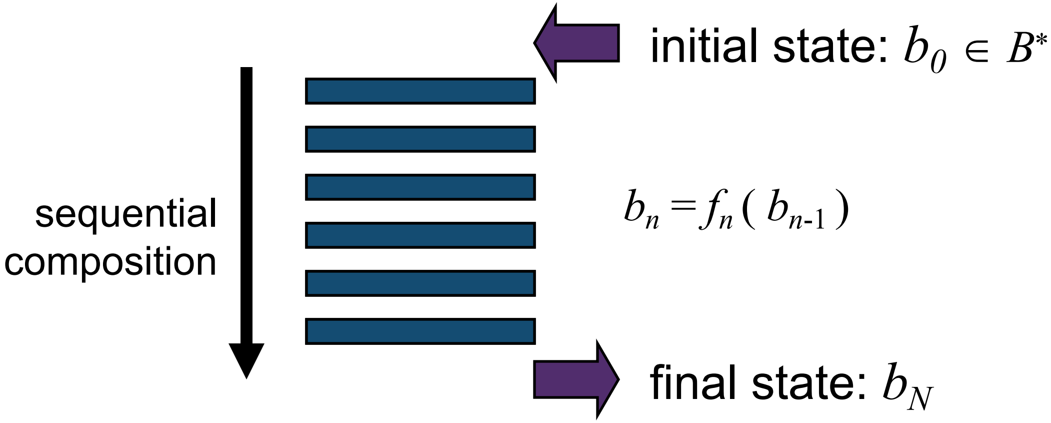
- Admits a pool of processes or threads access shared resources
- Signaling mechanism
- Binary Semaphore and Mutex operates differently, provides same result

After years of use without problems, a Berkeley Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.

Problems with the Foundations of Threads

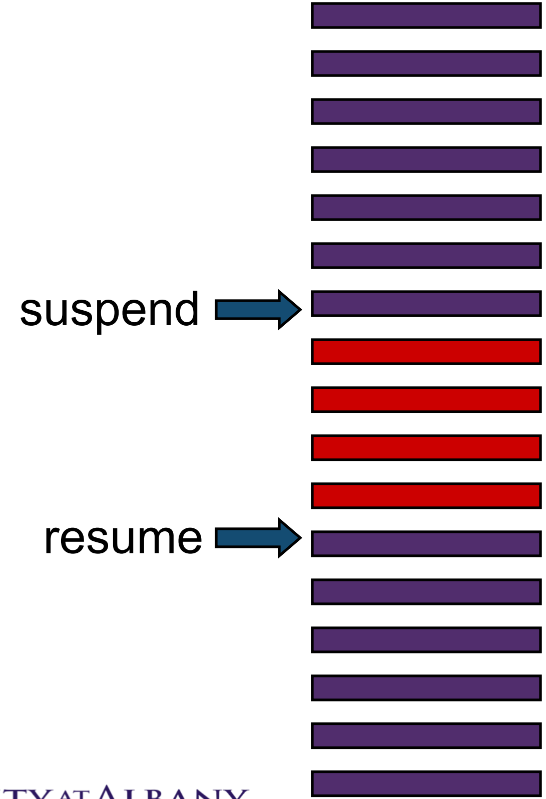
- A model of computation:
  - Bits: $B = \{0, 1\}$
  - Set of finite sequences of bits: B^*
  - Computation: $f : B^* \rightarrow B^*$
  - Composition of computations: $f \bullet f'$
  - Programs specify compositions of computations
- Threads augment this model to admit concurrency.
- But this model does not admit concurrency gracefully.

Basic Sequential Computation



Formally, composition of computations is function composition.

When There are Threads, Everything Changes



suspend

resume

A program no longer computes a function.

$$b_n = f_n(\, b_{n\text{-}1}\,)$$

another thread can change the state

$$b'_n = f_n(\, b'_{n\text{-}1}\,)$$

Apparently, programmers find this model appealing because nothing has changed in the *syntax*.

Succinct Problem Statement

- Threads are wildly nondeterministic.

- The programmer's job is to prune away the nondeterminism
  - by imposing constraints on execution order (e.g., mutexes)
  - limiting shared data accesses (e.g., OO design).

Processes & Message Passing

- Processes use their own memory space
- Implementation requires hardware support of Memory Management Unit (MMU)
- MMU provides address translation
- Message Passing :
  - One process creates data and keeps in a shared memory
  - Notifies another process

Processes: Problems

- Can be easier for shared variables
- Issues:
  - What is producer produces data faster than consumer?
  - How do we choose the size of buffer?
  - Message passing can deadlock as well
- Solution:
  - Higher level of concurrency models