operating systems

UNIT - 2

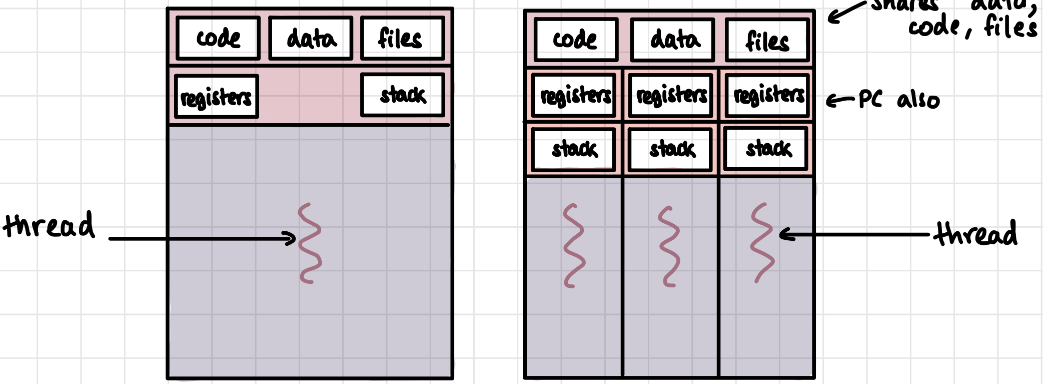
THREADS & CONCURRENCY

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

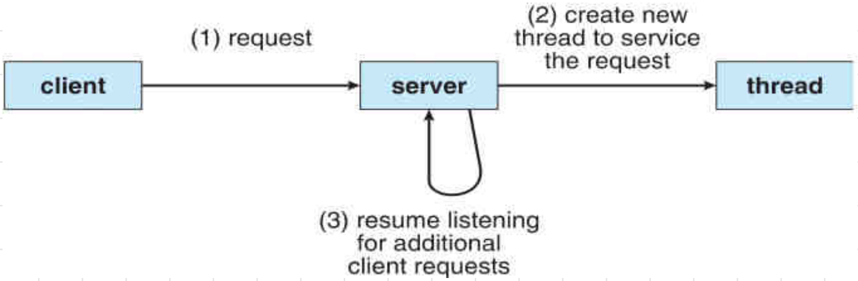
thread

- fundamental unit of CPU utilisation
- thread ID, PC, reg, stack
- run within app
- shares thread ID, data, file descriptors, signals to other threads of same process
- eg: browser
  - multiple tabs
  - load content, display animations, play video etc
- eg: word processor
  - input
  - spell checking
  - grammar checking



Multithreaded Server Architecture

- process creation heavy weighted, thread creation light weighted
- threads - remote procedure call (RPC) systems
- kernels usually multithreaded



(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

listen for requests
and assign thread

BENEFITS

- Responsiveness - continued execution if one thread blocked
  - UI
  - browser : image loading, user input

- Resource Sharing - share process resources, files, data etc
  - easier than shared memory or message passing (processes)
  - many threads within same address space

- Economy - cheaper than process creation, lower overhead than context switching
  - own registers and stack
  - eg: Solaris, process creation 30x slower and context switching 5x slower

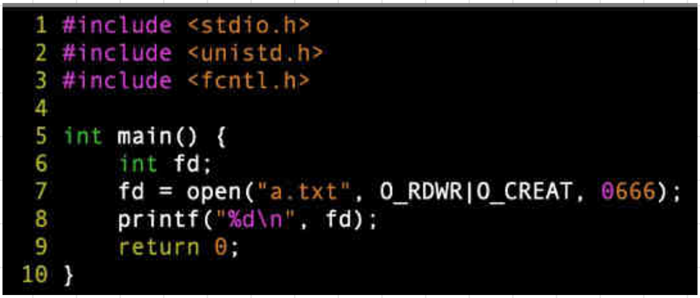
- Scalability – multiprocessor architecture
  - can run on multiple cores parallely

PROCESS vs THREAD

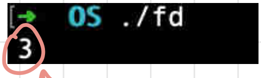
Process	Thread
• default: no shared memory	• default: shared memory
• most file descriptors not shared	• will share file descriptors
• do not share filesystem context	• share filesystem context
• do not share signal handling	• share signal handling

file descriptor

Read / Open File



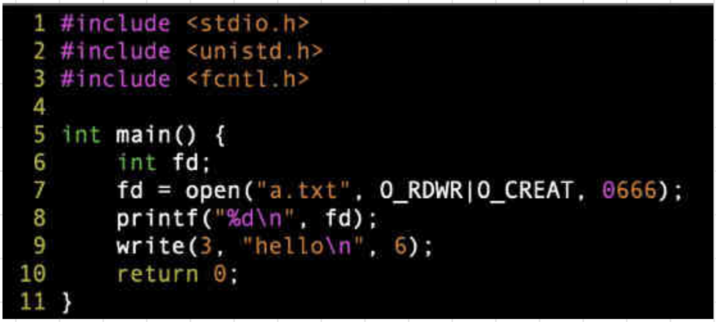
```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    fd = open("a.txt", O_RDWR|O_CREAT, 0666);
    printf("%d\n", fd);
    return 0;
}
```



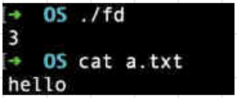
```
➜  OS ./fd
3
```

file descriptor integer

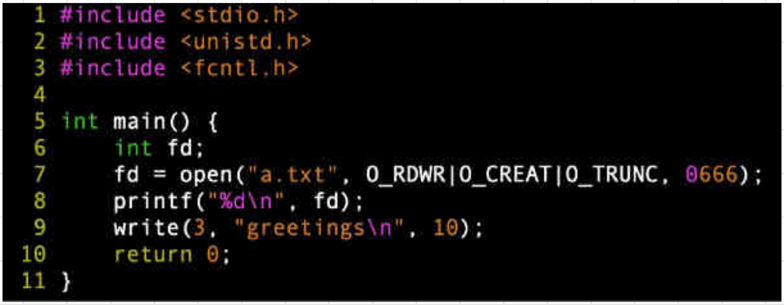
Write



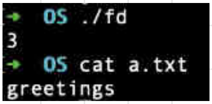
```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    fd = open("a.txt", O_RDWR|O_CREAT, 0666);
    printf("%d\n", fd);
    write(3, "hello\n", 6);
    return 0;
}
```



```
→  OS ./fd
3
→  OS cat a.txt
hello
```



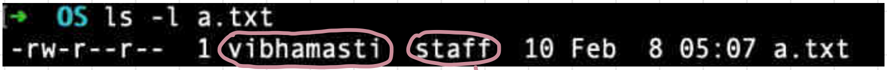
```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    fd = open("a.txt", O_RDWR|O_CREAT|O_TRUNC, 0666);
    printf("%d\n", fd);
    write(3, "greetings\n", 10);
    return 0;
}
```



```
→  OS ./fd
3
→  OS cat a.txt
greetings
```

Attributes Shared by Threads

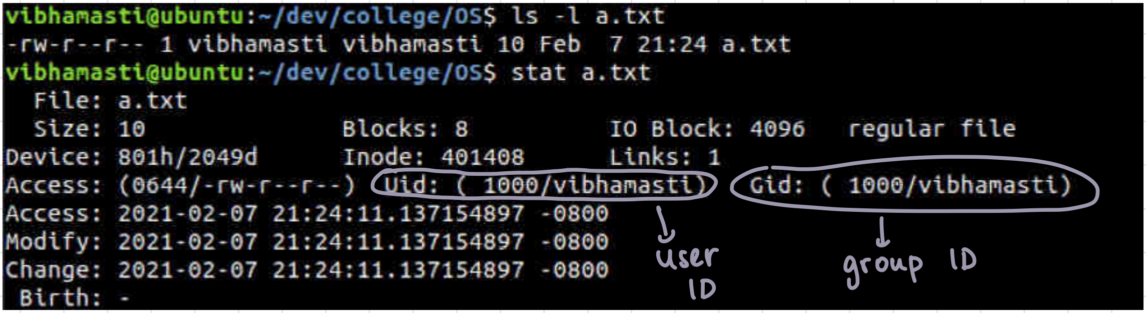
- process ID, parent process ID, process group ID, session ID



```
→  OS ls -l a.txt
-rw-r--r--  1 vibhamasti  staff  10 Feb  8 05:07 a.txt
```

user

group



```
vibhamasti@ubuntu:~/dev/college/OS$ ls -l a.txt
-rw-r--r-- 1 vibhamasti vibhamasti 10 Feb  7 21:24 a.txt
vibhamasti@ubuntu:~/dev/college/OS$ stat a.txt
  File: a.txt
  Size: 10              Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d      Inode: 401408      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/vibhamasti)   Gid: ( 1000/vibhamasti)
Access: 2021-02-07 21:24:11.137154897 -0800
Modify: 2021-02-07 21:24:11.137154897 -0800
Change: 2021-02-07 21:24:11.137154897 -0800
 Birth: -
```

user ID

group ID

- controlling terminal
- process credentials (user ID, group ID)
- record locks created using fcntl(); signal dispositions
- file system related information; umask, cwd, root
- resource limits, CPU time consumed (returned by times())
- resources consumed (getrusage()), nice value (setpriority(), nice())

Attributes Specific to Threads

- thread ID, signal mask
- errno variable, specific data
- floating point environment (fenv(3))
- stack

thread

- ID
- registers
- stack
- scheduling priority, policy
- signal mask
- errno variable
- thread-specific data

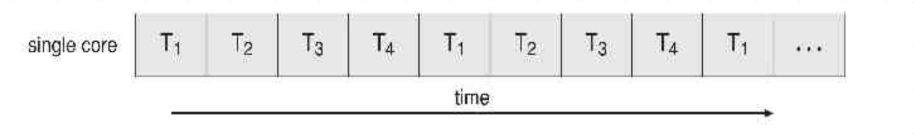
- no guarantee of execution order of newly created thread and calling thread

- newly created thread: access to process address space, inherits calling thread's fenv and signal mask

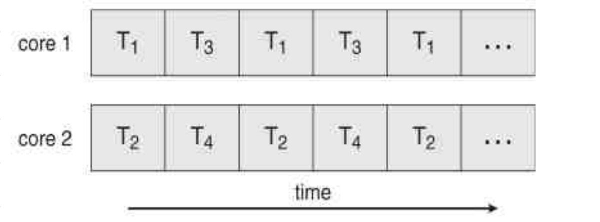
- pending signals cleared

- pthread functions return error code when they fail (do not set errno like other POSIX functions)

single CORE



multi CORE



MULTICORE PROGRAMMING

- challenges: dividing activities, balance, data splitting, data dependency, testing and debugging

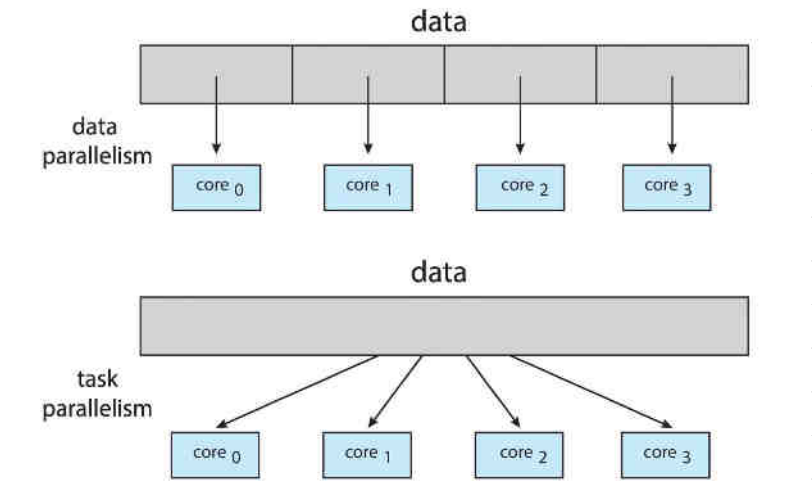
- parallelism: system can perform more than one task simultaneously

- concurrency: more than one task making progress ; scheduler provides concurrency in single core processors

Parallelism

1) Data parallelism: distributes subsets of same data across multiple cores, same operation on each
eg: sum of n numbers, each core finds subsum

2) Task parallelism: distributing threads across cores, each thread performing unique operation



AMDAHL'S LAW

- performance gains from adding additional cores to an app with both serial and parallel components

- S: portion of app that needs to be done in serial
N: processing cores

$$\boxed{\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}}$$

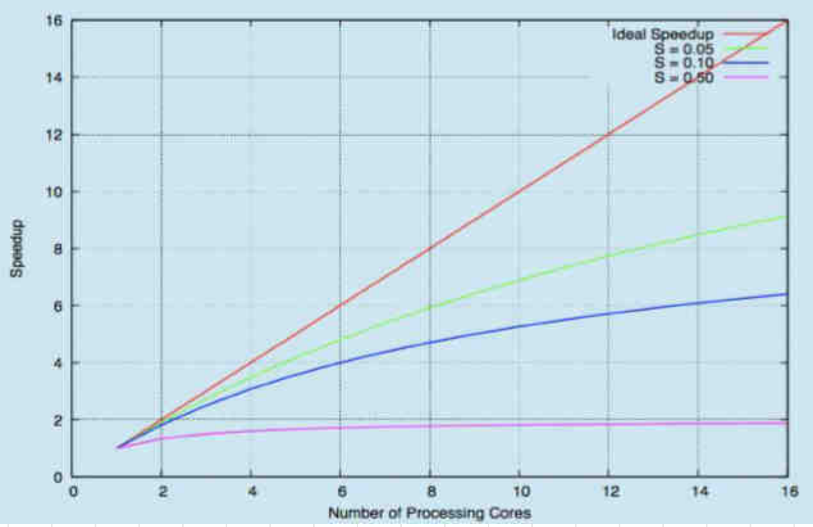
- if app is 75% parallel, 25% serial, moving from 1 to 2 cores

$$N=1 \quad \text{speedup} \leq \frac{1}{0.25+0.75} = 1$$

$$N=2: \quad \text{speedup} \leq \frac{1}{0.25+\frac{0.75}{2}} = 1.6$$

$\therefore$ speedup = 1.6 times

- as $n \rightarrow \infty$, speedup $\rightarrow \frac{1}{S}$

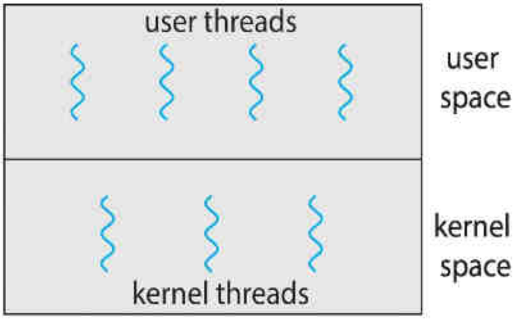


User threads & kernel threads

User threads: management done by user-level threads library
- POSIX Pthreads
- windows threads
- Java threads

Kernel threads: supported by kernel

- general purpose OSes



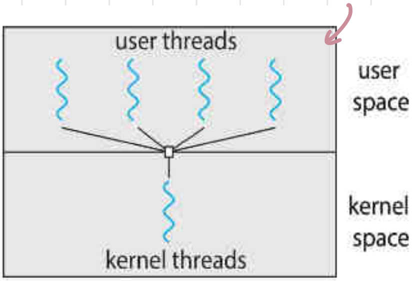
MULTITHREADING MODELS

- Many-to-one
- One-to-one
- Many-to-many

kernal threads for a user thread

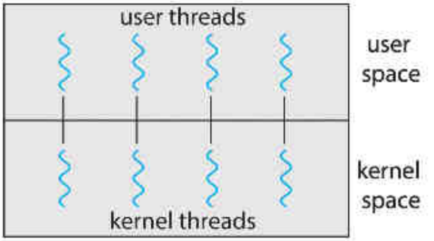
Many-to-One

- many user-level threads mapped to single kernel thread
- one thread blocking causes all to be blocked
- few systems use: Solaris Green Threads, GNU portable Threads



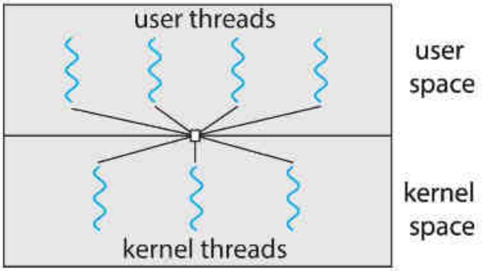
One-to-One

- single user thread mapped to single kernel thread
- more concurrency than many-to-one
- creating user thread creates kernel thread
- eg: Linux, Windows, Solaris 9 and later



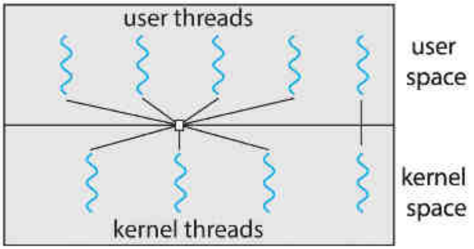
Many-to-Many

- many user level threads to be mapped to many kernel threads
- allows OS to create sufficient kernel threads
- eg: Solaris prior to version 9, Windows w ThreadFiber package



Two-Level Model

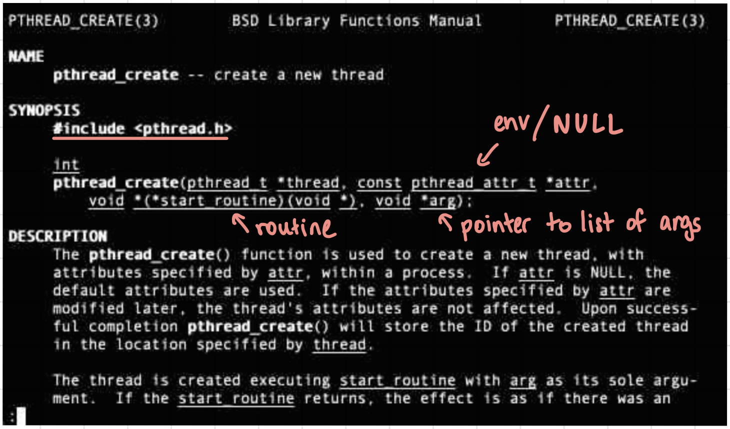
- Similar to M:M but allows UT to bind to KT
- eg: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



PTHREADS

- either user-level or kernel level
- POSIX standard API — specification
- In UNIX OSes — Solaris, Linux, MacOS

pthread-create()



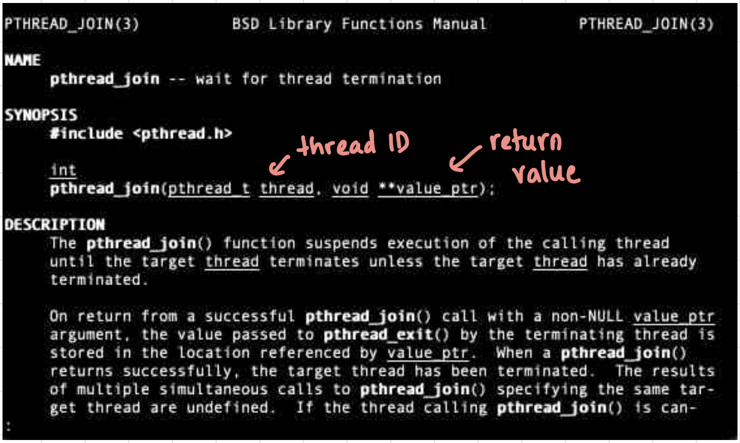
```
PTHREAD_CREATE(3)         BSD Library Functions Manual         PTHREAD_CREATE(3)

NAME
     pthread_create -- create a new thread

SYNOPSIS
     #include <pthread.h>

     int
     pthread_create(pthread_t *thread, const pthread_attr_t *attr,
         void *(*start_routine)(void *), void *arg);

DESCRIPTION
     The pthread_create() function is used to create a new thread, with
     attributes specified by attr, within a process.  If attr is NULL, the
     default attributes are used.  If the attributes specified by attr are
     modified later, the thread's attributes are not affected.  Upon success-
     ful completion pthread_create() will store the ID of the created thread
     in the location specified by thread.

     The thread is created executing start_routine with arg as its sole argu-
     ment.  If the start_routine returns, the effect is as if there was an
:
```

env/NULL

routine

pointer to list of args

pthread-join()



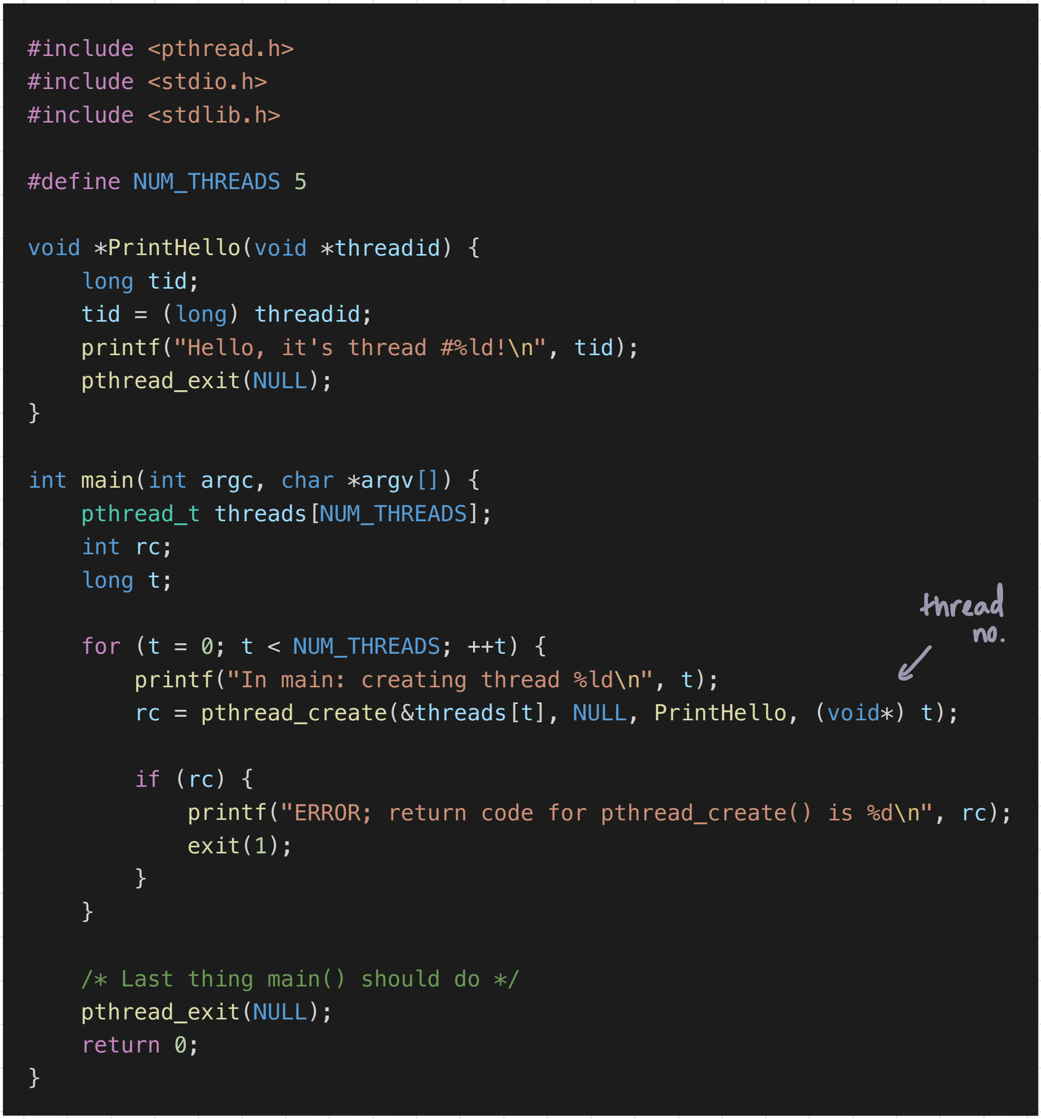
```
PTHREAD_JOIN(3)           BSD Library Functions Manual           PTHREAD_JOIN(3)

NAME
     pthread_join -- wait for thread termination

SYNOPSIS
     #include <pthread.h>

     int
     pthread_join(pthread_t thread, void **value_ptr);

DESCRIPTION
     The pthread_join() function suspends execution of the calling thread
     until the target thread terminates unless the target thread has already
     terminated.

     On return from a successful pthread_join() call with a non-NULL value_ptr
     argument, the value passed to pthread_exit() by the terminating thread is
     stored in the location referenced by value_ptr.  When a pthread_join()
     returns successfully, the target thread has been terminated.  The results
     of multiple simultaneous calls to pthread_join() specifying the same tar-
     get thread are undefined.  If the thread calling pthread_join() is can-
:
```

thread ID

return value

pthread-exit() , pthread-self(), pthread_attr_init()

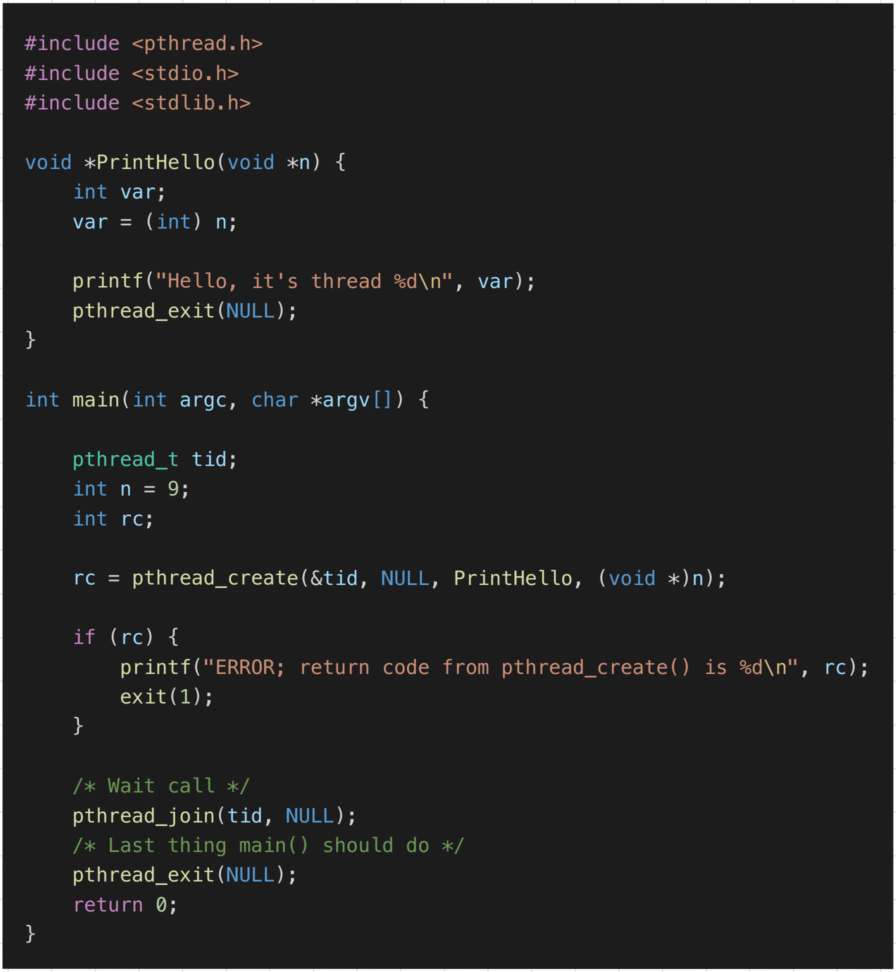
thread1.c https://www.geeksforgeeks.org/multithreading-c-2/



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

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    long tid;
    tid = (long) threadid;
    printf("Hello, it's thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;

    for (t = 0; t < NUM_THREADS; ++t) {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void*) t);

        if (rc) {
            printf("ERROR; return code for pthread_create() is %d\n", rc);
            exit(1);
        }
    }

    /* Last thing main() should do */
    pthread_exit(NULL);
    return 0;
}
```

thread no.

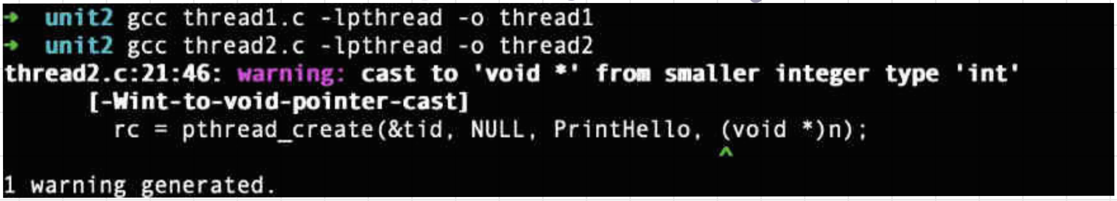
thread2.c



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

void *PrintHello(void *n) {
    int var;
    var = (int) n;

    printf("Hello, it's thread %d\n", var);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

    pthread_t tid;
    int n = 9;
    int rc;

    rc = pthread_create(&tid, NULL, PrintHello, (void *)n);

    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(1);
    }

    /* Wait call */
    pthread_join(tid, NULL);
    /* Last thing main() should do */
    pthread_exit(NULL);
    return 0;
}
```

COMPILING

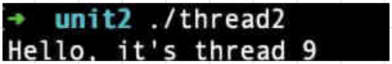
linking necessary



```
➜  unit2 gcc thread1.c -lpthread -o thread1
➜  unit2 gcc thread2.c -lpthread -o thread2
thread2.c:21:46: warning: cast to 'void *' from smaller integer type 'int'
      [-Wint-to-void-pointer-cast]
        rc = pthread_create(&tid, NULL, PrintHello, (void *)n);
                                                     ^
1 warning generated.
```

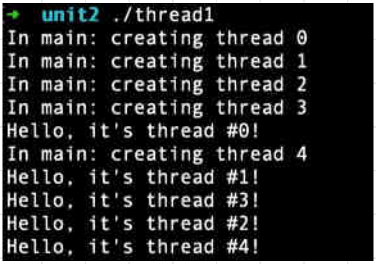
OUTPUT

thread2.c



```
➜  unit2 ./thread2
Hello, it's thread 9
```

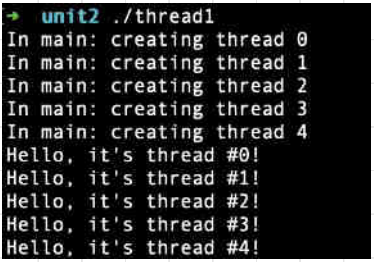
thread1.c



```
→  unit2 ./thread1
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello, it's thread #0!
In main: creating thread 4
Hello, it's thread #1!
Hello, it's thread #3!
Hello, it's thread #2!
Hello, it's thread #4!
```

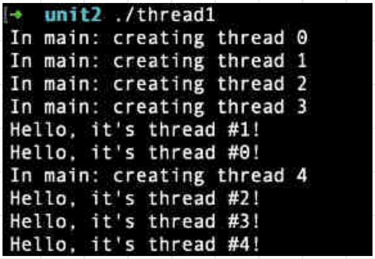
synchronisation - later

race condition



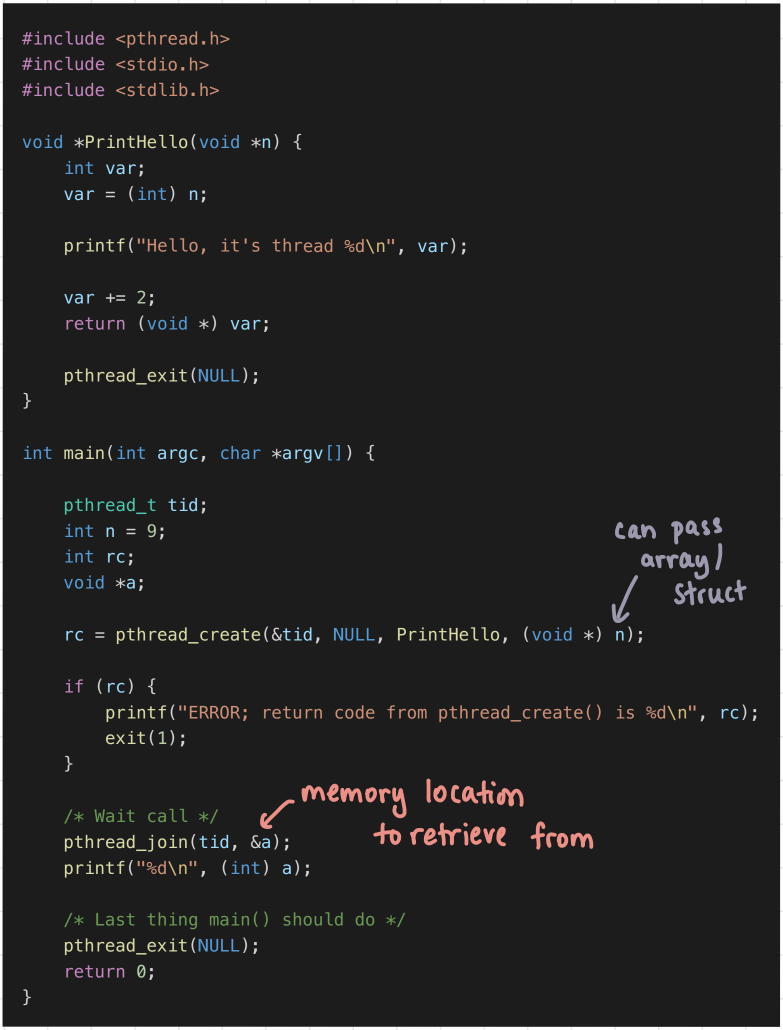
```
→  unit2 ./thread1
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello, it's thread #0!
Hello, it's thread #1!
Hello, it's thread #2!
Hello, it's thread #3!
Hello, it's thread #4!
```

competing for CPU resources



```
→  unit2 ./thread1
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello, it's thread #1!
Hello, it's thread #0!
In main: creating thread 4
Hello, it's thread #2!
Hello, it's thread #3!
Hello, it's thread #4!
```

thread3.c return from routine

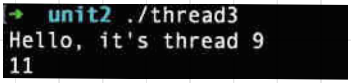


```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *PrintHello(void *n) {
    int var;
    var = (int) n;

    printf("Hello, it's thread %d\n", var);

    var += 2;
    return (void *) var;

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

    pthread_t tid;
    int n = 9;
    int rc;
    void *a;

    rc = pthread_create(&tid, NULL, PrintHello, (void *) n);

    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(1);
    }

    /* Wait call */
    pthread_join(tid, &a);
    printf("%d\n", (int) a);

    /* Last thing main() should do */
    pthread_exit(NULL);
    return 0;
}
```

can pass array / struct

memory location to retrieve from

OUTPUT



```
→  unit2 ./thread3
Hello, it's thread 9
11
```

THREAD SCHEDULING

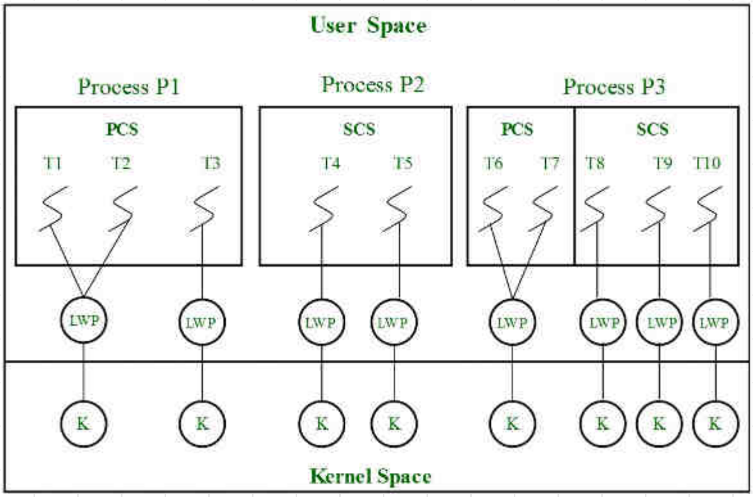
- distinction b/w user-level and kernel-level threads

- user-level threads on lightweight processes (LWP)
  - Process-Contention Scope (PCS): threads within app/process
  - System-Contention Scope (SCS): threads within system

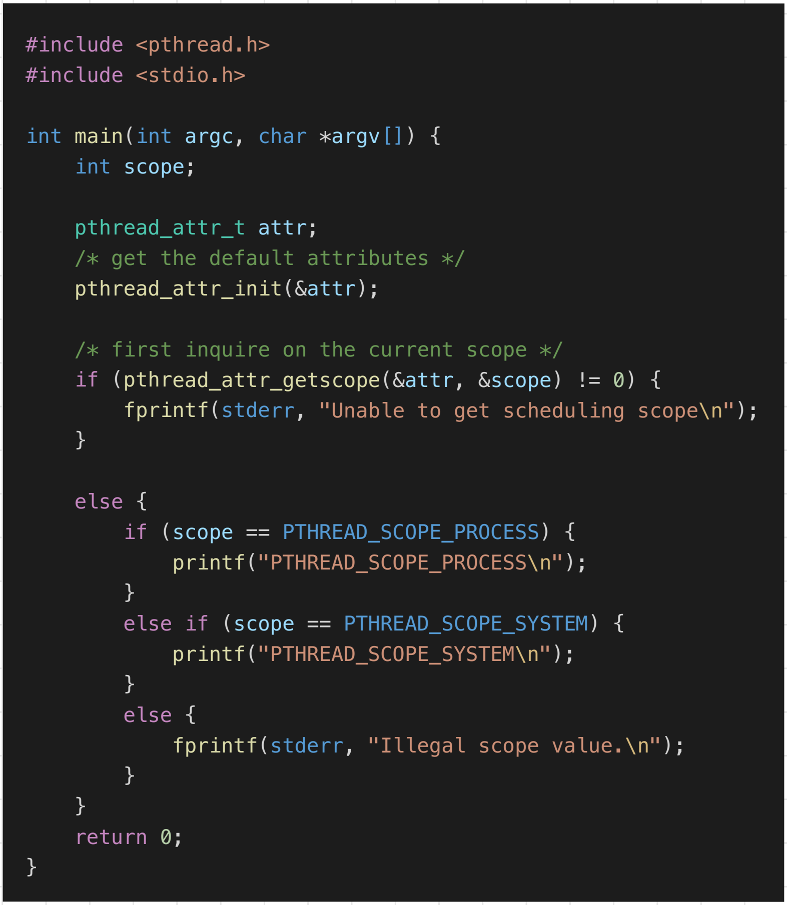
- API to specify scope of thread (PCS or SCS)
  - PTHREAD_SCOPE_PROCESS: PCS
  - PTHREAD_SCOPE_SYSTEM: SCS

- Linux & MacOS only allow PTHREAD_SCOPE_SYSTEM

- More: https://www.geeksforgeeks.org/thread-scheduling/

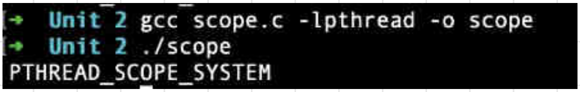


Scope.c



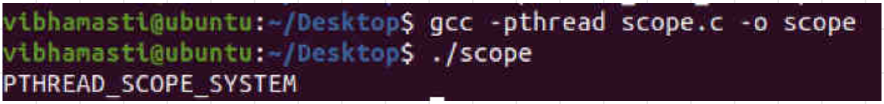
```c
#include <pthread.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int scope;

    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get scheduling scope\n");
    }

    else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD_SCOPE_PROCESS\n");
        }
        else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD_SCOPE_SYSTEM\n");
        }
        else {
            fprintf(stderr, "Illegal scope value.\n");
        }
    }
    return 0;
}
```

MacOs



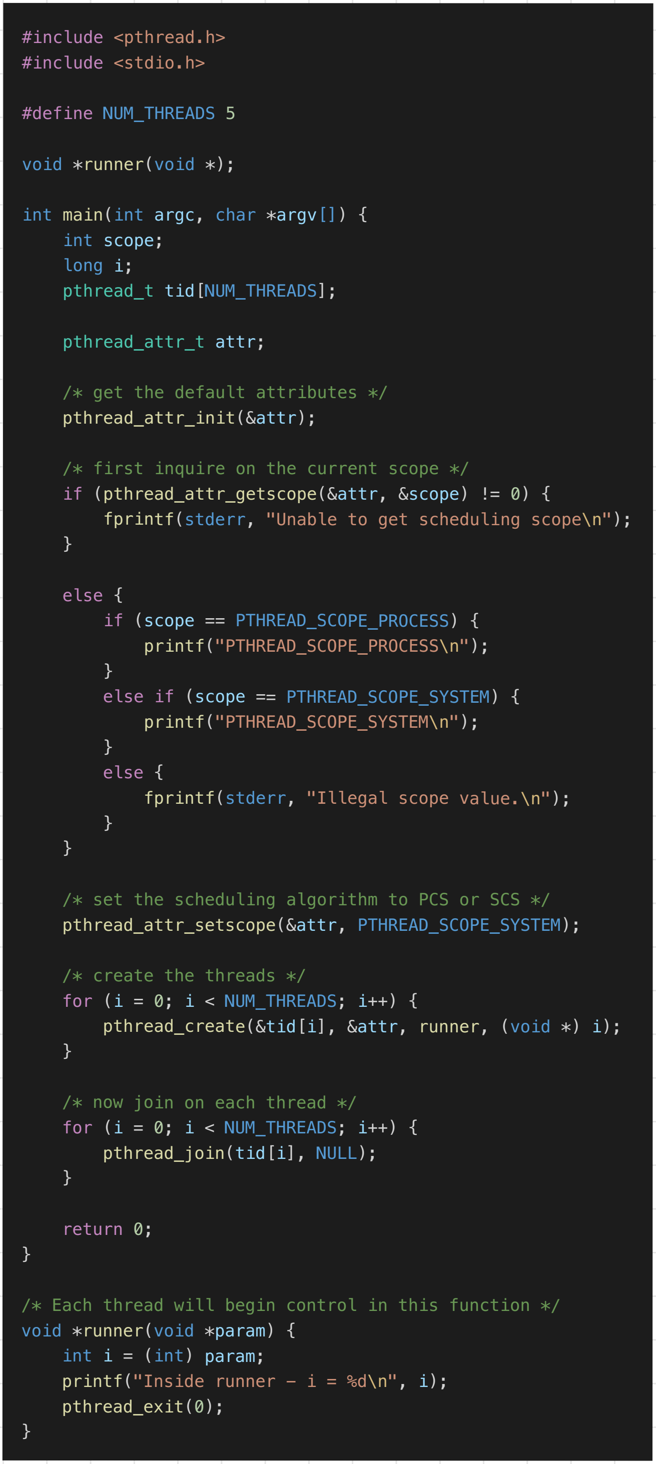
```
➜  Unit 2 gcc scope.c -lpthread -o scope
➜  Unit 2 ./scope
PTHREAD_SCOPE_SYSTEM
```

Linux

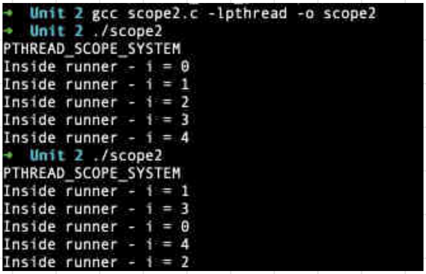


```
vibhamasti@ubuntu:~/Desktop$ gcc -pthread scope.c -o scope
vibhamasti@ubuntu:~/Desktop$ ./scope
PTHREAD_SCOPE_SYSTEM
```

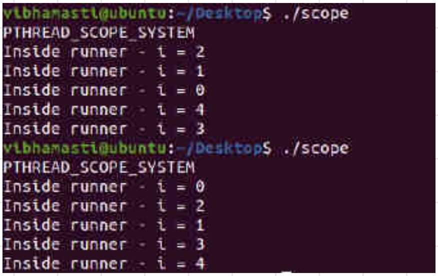
scope2.



```c
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *runner(void *);

int main(int argc, char *argv[]) {
    int scope;
    long i;
    pthread_t tid[NUM_THREADS];

    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get scheduling scope\n");
    }

    else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD_SCOPE_PROCESS\n");
        }
        else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD_SCOPE_SYSTEM\n");
        }
        else {
            fprintf(stderr, "Illegal scope value.\n");
        }
    }

    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&tid[i], &attr, runner, (void *) i);
    }

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}

/* Each thread will begin control in this function */
void *runner(void *param) {
    int i = (int) param;
    printf("Inside runner - i = %d\n", i);
    pthread_exit(0);
}
```

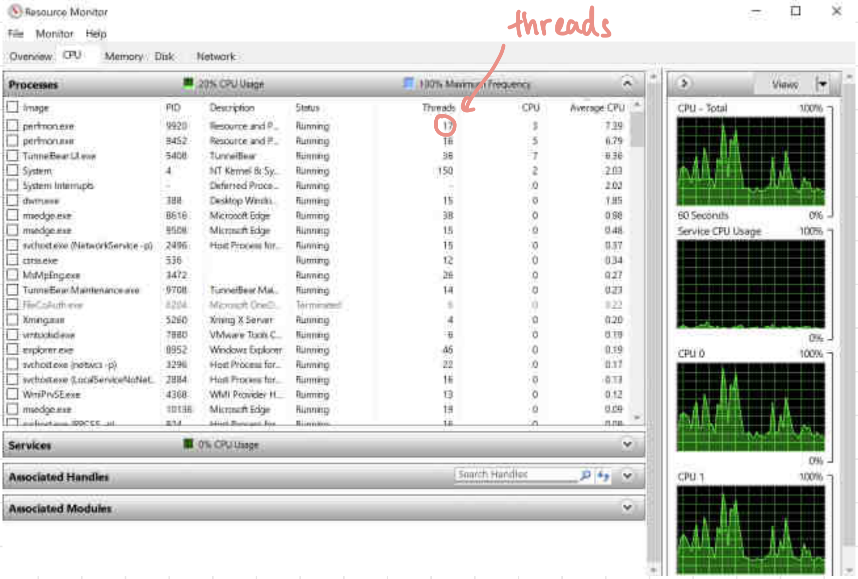


```
➜  Unit 2 gcc scope2.c -lpthread -o scope2
➜  Unit 2 ./scope2
PTHREAD_SCOPE_SYSTEM
Inside runner - i = 0
Inside runner - i = 1
Inside runner - i = 2
Inside runner - i = 3
Inside runner - i = 4
➜  Unit 2 ./scope2
PTHREAD_SCOPE_SYSTEM
Inside runner - i = 1
Inside runner - i = 3
Inside runner - i = 0
Inside runner - i = 4
Inside runner - i = 2
```



```
vibhamasti@ubuntu:~/Desktop$ ./scope
PTHREAD_SCOPE_SYSTEM
Inside runner - i = 2
Inside runner - i = 1
Inside runner - i = 0
Inside runner - i = 4
Inside runner - i = 3
vibhamasti@ubuntu:~/Desktop$ ./scope
PTHREAD_SCOPE_SYSTEM
Inside runner - i = 0
Inside runner - i = 2
Inside runner - i = 1
Inside runner - i = 3
Inside runner - i = 4
```

- Windows - Resource monitor



Processes 20% CPU Usage 100% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average CPU
perfmon.exe	9920	Resource and P...	Running	17	3	7.39
perfmon.exe	9452	Resource and P...	Running	16	5	6.79
TunnelBear.UI.exe	5408	TunnelBear	Running	38	7	6.36
System	4	NT Kernel & Sy...	Running	150	2	2.03
System Interrupts	-	Deferred Proce...	Running	-	0	2.02
dwm.exe	388	Desktop Windo...	Running	15	0	1.85
msedge.exe	8616	Microsoft Edge	Running	38	0	0.98
msedge.exe	9508	Microsoft Edge	Running	15	0	0.48
svchost.exe (NetworkService -p)	2496	Host Process for...	Running	15	0	0.37
csrss.exe	536		Running	12	0	0.34
MsMpEng.exe	3472		Running	26	0	0.27
TunnelBear.Maintenance.exe	9708	TunnelBear.Mai...	Running	14	0	0.23
FileCoAuth.exe	8204	Microsoft OneD...	Terminated	5	0	0.22
Xming.exe	5260	Xming X Server	Running	4	0	0.20
vmtoolsd.exe	7880	VMware Tools C...	Running	6	0	0.19
explorer.exe	8952	Windows Explorer	Running	46	0	0.19
svchost.exe (netsvcs -p)	3296	Host Process for...	Running	22	0	0.17
svchost.exe (LocalServiceNoNet...	2884	Host Process for...	Running	16	0	0.13
WmiPrvSE.exe	4368	WMI Provider H...	Running	13	0	0.12
msedge.exe	10136	Microsoft Edge	Running	19	0	0.09
svchost.exe (RPCSS -p)	824	Host Process for...	Running	16	0	0.08

Services 0% CPU Usage

Associated Handles Search Handles

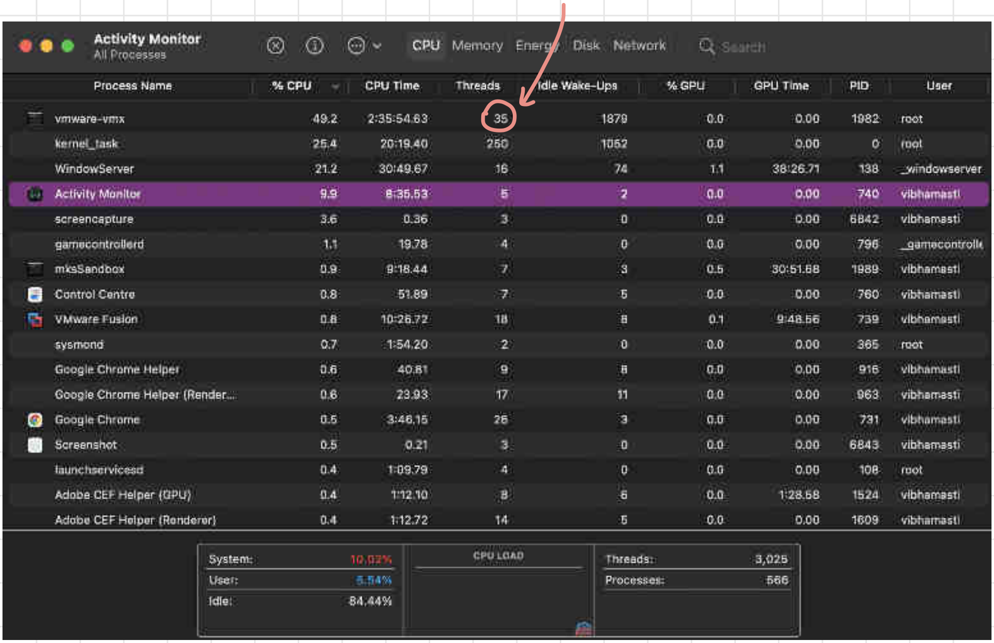
Associated Modules

threads

cores (2)

- MacOS - Activity Monitor

threads



Process Name	% CPU	CPU Time	Threads	Idle Wake-Ups	% GPU	GPU Time	PID	User
vmware-vmx	49.2	2:35:54.63	35	1879	0.0	0.00	1982	root
kernel_task	25.4	20:19.40	250	1052	0.0	0.00	0	root
WindowServer	21.2	30:49.67	16	74	1.1	38:26.71	138	_windowserver
Activity Monitor	9.9	8:35.53	5	2	0.0	0.00	740	vibhamasti
screencapture	3.6	0.36	3	0	0.0	0.00	6842	vibhamasti
gamecontrollerd	1.1	19.78	4	0	0.0	0.00	796	_gamecontrolle
mksSandbox	0.9	9:18.44	7	3	0.5	30:51.68	1989	vibhamasti
Control Centre	0.8	51.89	7	5	0.0	0.00	760	vibhamasti
VMware Fusion	0.8	10:26.72	18	8	0.1	9:48.56	739	vibhamasti
sysmond	0.7	1:54.20	2	0	0.0	0.00	365	root
Google Chrome Helper	0.6	40.81	9	8	0.0	0.00	916	vibhamasti
Google Chrome Helper (Render...	0.6	23.93	17	11	0.0	0.00	963	vibhamasti
Google Chrome	0.5	3:46.15	26	3	0.0	0.00	731	vibhamasti
Screenshot	0.5	0.21	3	0	0.0	0.00	6843	vibhamasti
launchservicesd	0.4	1:09.79	4	0	0.0	0.00	108	root
Adobe CEF Helper (GPU)	0.4	1:12.10	8	6	0.0	1:28.58	1524	vibhamasti
Adobe CEF Helper (Renderer)	0.4	1:12.72	14	5	0.0	0.00	1609	vibhamasti

System: 10.02%
User: 5.54%
Idle: 84.44%

CPU LOAD

Threads: 3,025
Processes: 566

WINDOWS MULTITHREADED C PROGRAM

- Win32 API
- one-to-one mapping
- support for a fibre library – many-to-many

- thread
  - ID
  - reg set – processor status
  - user stack
  - kernel stack
  - private storage area (DLLs)

- CreateThread()
- WaitForSingleObject() — join

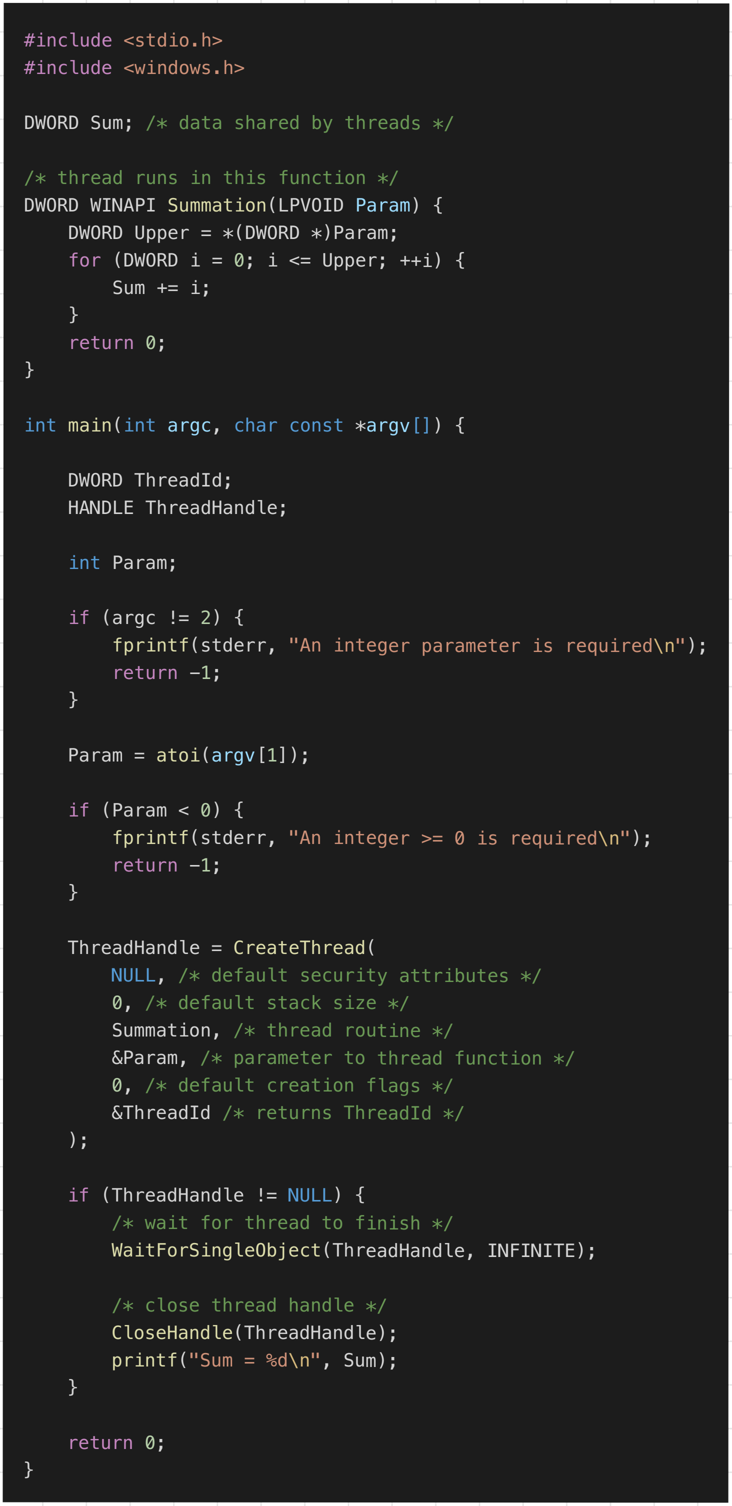
C Multi-Threaded Programs

- Global vars
  - shared by all threads
  - stored in data segment

- Local vars
  - for each thread

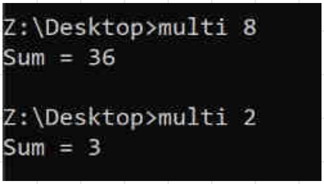
- DWORD – long int

PROGRAM IN WINDOWS



```c
#include <stdio.h>
#include <windows.h>

DWORD Sum; /* data shared by threads */

/* thread runs in this function */
DWORD WINAPI Summation(LPVOID Param) {
    DWORD Upper = *(DWORD *)Param;
    for (DWORD i = 0; i <= Upper; ++i) {
        Sum += i;
    }
    return 0;
}

int main(int argc, char const *argv[]) {

    DWORD ThreadId;
    HANDLE ThreadHandle;

    int Param;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }

    Param = atoi(argv[1]);

    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }

    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread routine */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId /* returns ThreadId */
    );

    if (ThreadHandle != NULL) {
        /* wait for thread to finish */
        WaitForSingleObject(ThreadHandle, INFINITE);

        /* close thread handle */
        CloseHandle(ThreadHandle);
        printf("Sum = %d\n", Sum);
    }

    return 0;
}
```

sum of numbers
from 1 ton



```
Z:\Desktop>multi 8
Sum = 36

Z:\Desktop>multi 2
Sum = 3
```

MUTUAL EXCLUSION & SYNCHRONISATION

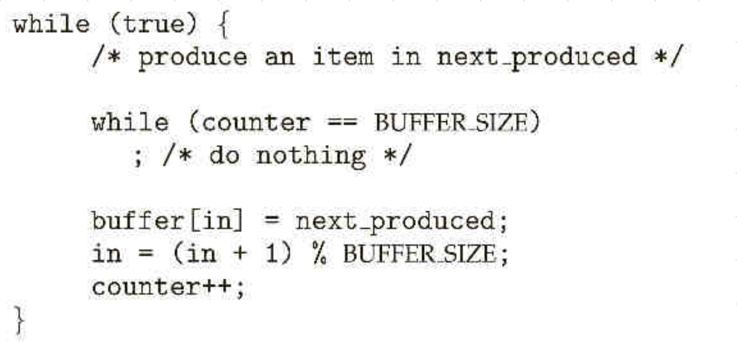
Producer - Consumer Problem

- Producer produces items for buffer
- Consumer consumes items from buffer

- Counter incremented when producer produces
- Counter decrements when consumer consumes

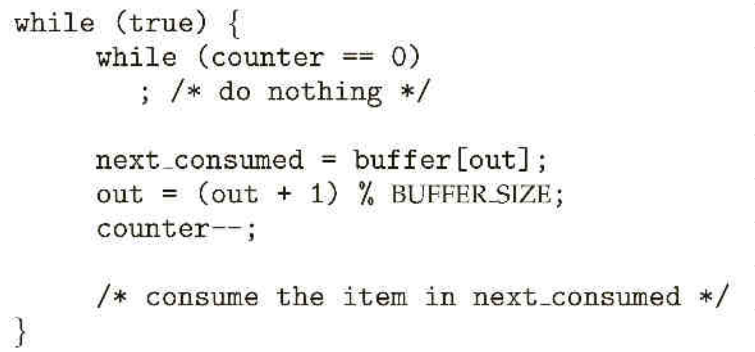
- Counter - keeps track of buffer
- Shared variable counter

PRODUCER



```
while (true) {
      /* produce an item in next_produced */

      while (counter == BUFFER_SIZE)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

CONSUMER



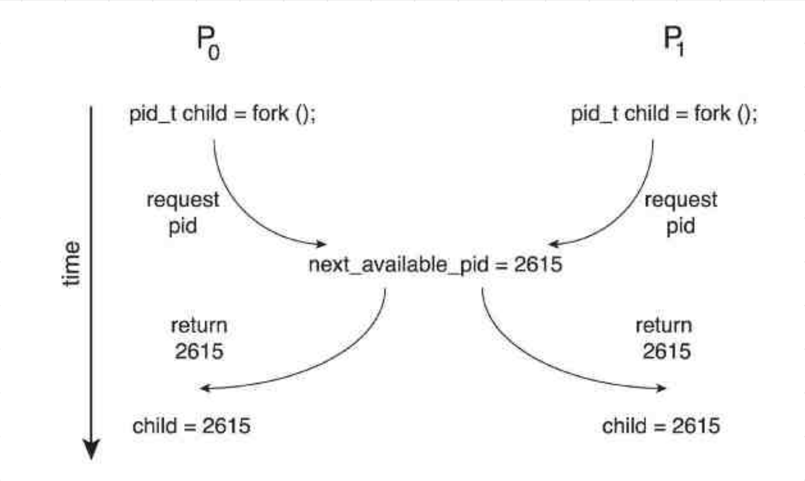
```
while (true) {
      while (counter == 0)
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;

      /* consume the item in next_consumed */
}
```

can be inconsistencies if 2 processes competing to access same resources

race condition

PROCESS SYNCHRONISATION

Race Condition



Critical Section Problem

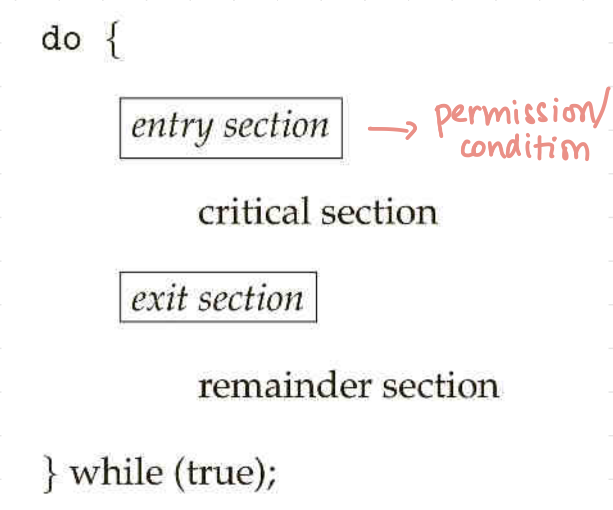
- Critical section: segment of code where kernel data structures are modified
  - changing common vars
  - updating process table
  - writing file

- System of n processes $\{p_0, p_1, \dots, p_n\}$

- Only one process in critical section at any given time; no other process allowed to enter critical section

- Each process asks permission to enter critical section

General structure for Process P_i

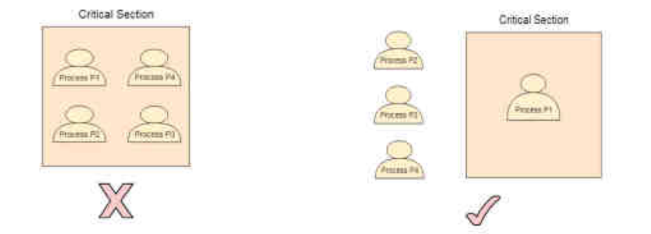


```
do {

      entry section

            critical section

      exit section

            remainder section

} while (true);
```

→ permission/ condition

SOLUTION

1. **Mutual Exclusion:** if P_i is executing in its critical section, no other process can be executing in critical section



2. **Progress:** selection of processes that will enter critical section next cannot be postponed indefinitely

3. **Bounded Waiting:** A bound on number of times that other processes are allowed to enter CS after a process has made a request to enter its CS and before that request is granted

CS Handling in OS

1) Preemptive:
   - allows preemption when running in kernel mode
   - difficult to design on SMP architectures (symmetric multi processor)

2) Non-preemptive:
   - runs until exits kernel mode, blocks or voluntarily yields CPU
   - free of race condition in kernel mode
   - free of race conditions on kernel data structures

- Preemptive kernel faster, more responsive

Peterson's Solution

- Software-based solution

- Assume load/store instructions are atomic (cannot be interrupted)

- For two processes only

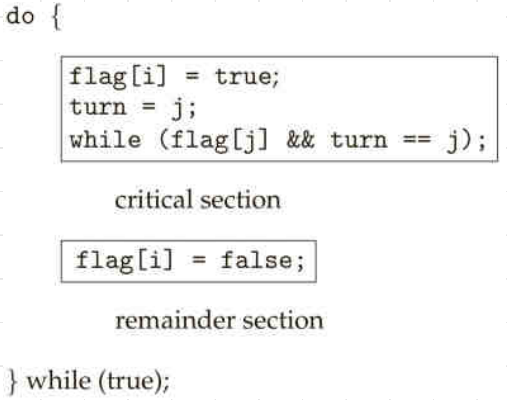
- P_i and P_j ; $j = 1-i$

- Requires 2 processes to share 2 data items

```
int turn;
bool flag[2];
```

- turn: whose turn it is to enter CS
- flag array: indicates if process ready to enter CS (wants to enter again)

Algorithm (for P_i)



```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = false;

            remainder section

} while (true);
```

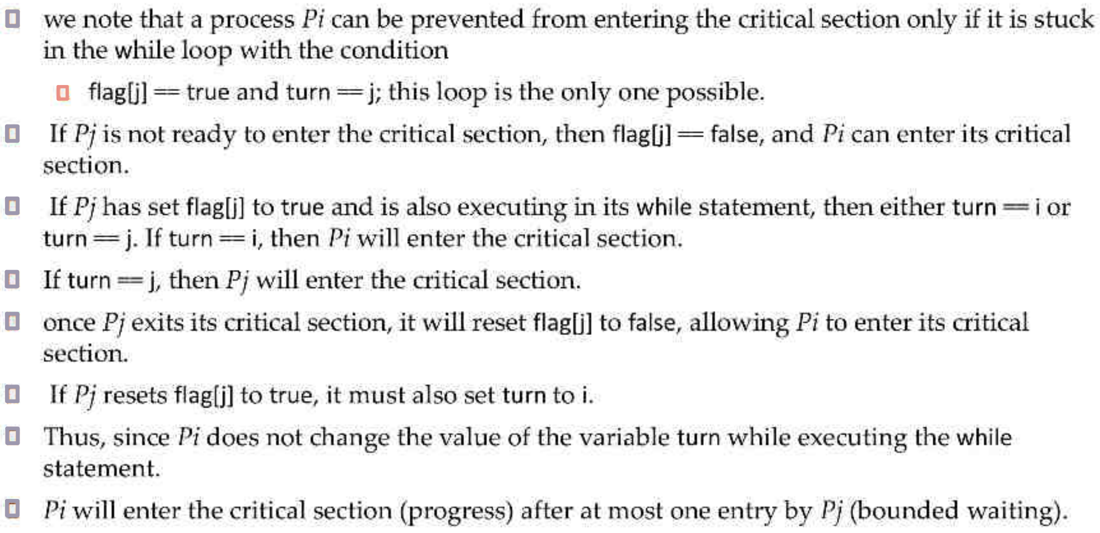
To prove Solution is Correct

1. Mutual exclusion preserved:
   P_i enters CS only if turn = i or flag[j] = false

2. Progress requirement is satisfied

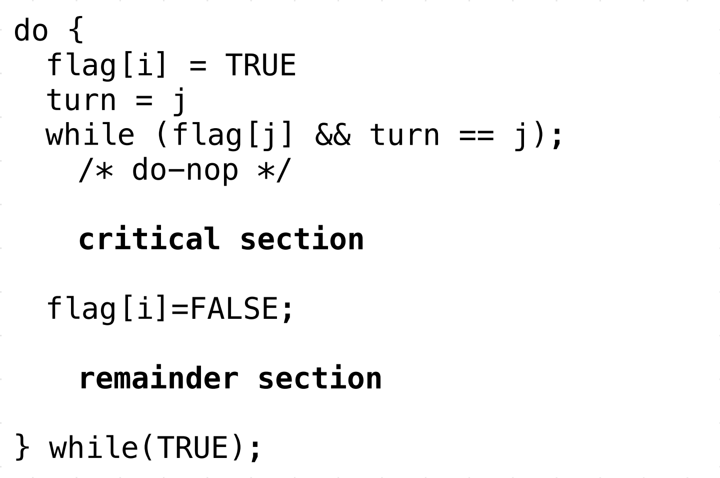
3. Bounded waiting time requirement is met

Prove 2 & 3



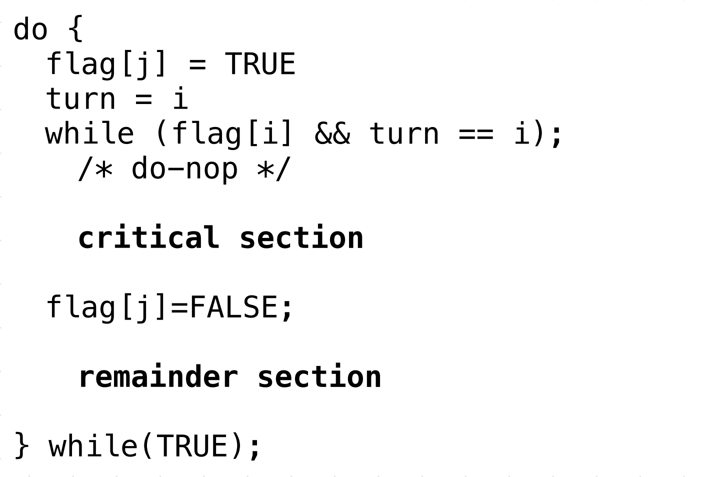
- □ we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition
  - □ flag[j] == true and turn == j; this loop is the only one possible.
- □ If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section.
- □ If *Pj* has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section.
- □ If turn == j, then *Pj* will enter the critical section.
- □ once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section.
- □ If *Pj* resets flag[j] to true, it must also set turn to i.
- □ Thus, since *Pi* does not change the value of the variable turn while executing the while statement.
- □ *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

Code for P_i



```
do {
  flag[i] = TRUE
  turn = j
  while (flag[j] && turn == j);
    /* do-nop */

    critical section

  flag[i]=FALSE;

    remainder section

} while(TRUE);
```

Code for P_j



```
do {
  flag[j] = TRUE
  turn = i
  while (flag[i] && turn == i);
    /* do-nop */

    critical section

  flag[j]=FALSE;

    remainder section

} while(TRUE);
```

- Cannot predict when each process gets interrupted

SYNCHRONISATION HARDWARE

- Hardware solution for critical section problem

- Locking: protect critical areas with locks ; lock and unlock technique

- Locking – entry section , process then moves to critcal section, then enters exit section, unlocking – exit section

- All 3 conditions of critical section satisfied

- Modern OSes cannot disable preemption

- Atomic hardware instructions — cannot be interrupted
  - test memory word and set value (test and set)
  - swap contents of two memory words

Test and Set Lock (TSL)

- test_and_set instruction – sync

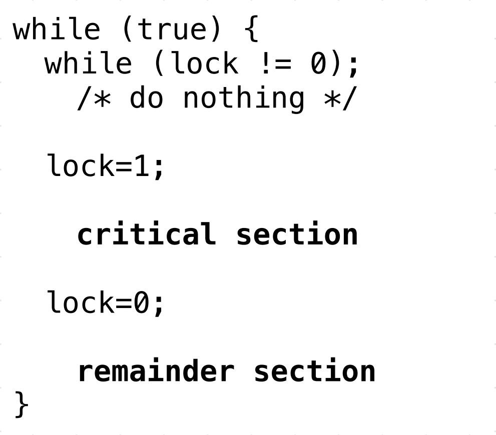
- Returns old value of memory location and sets it to 1 in an atomic operation

- One process executing test-and-set cannot be interrupted by another process executing test-and-set (atomic)

Scheme #1

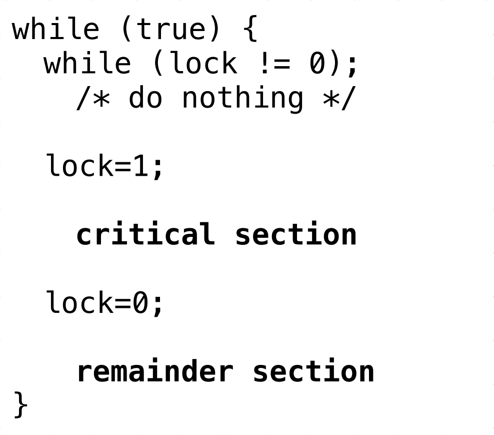
- Is mutual exclusion satisfied?
- Starting value: lock = 0

Process 0



```
while (true) {
  while (lock != 0);
    /* do nothing */

  lock=1;

    critical section

  lock=0;

    remainder section
}
```

Process 1



```
while (true) {
  while (lock != 0);
    /* do nothing */

  lock=1;

    critical section

  lock=0;

    remainder section
}
```

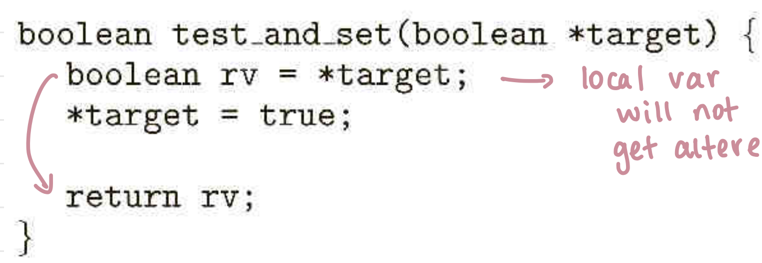
Execution Sequence

lock = 0

P0: while lock != 0;
// context switch
P1: while lock != 0;
lock = 1
// context switch
P0: lock = 1
critical section

mutual exclusion not satisfied

test-and-set Instruction



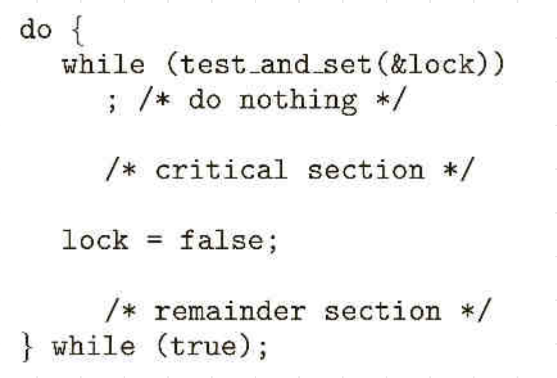
```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

→ local var will not get altered

- Atomic

lock = 0

Implementation of Mutual Exclusion

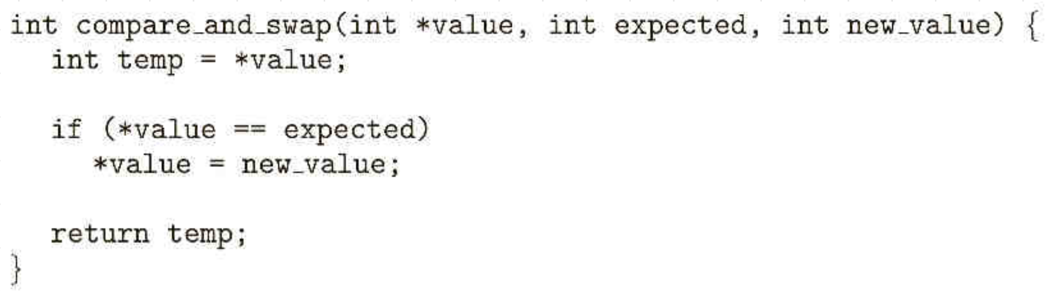


```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

*target = 0

rv = 0

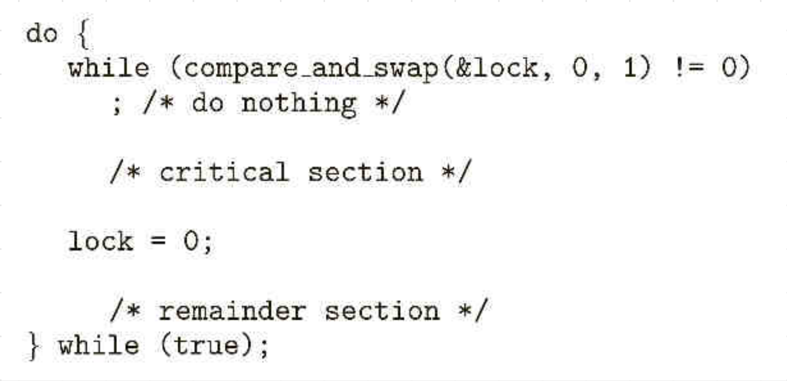
compare-and-swap Instruction



```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

- Atomic
- Only if *value == expected, *value = new_value
- x86 & Itanium → CMPXCHG

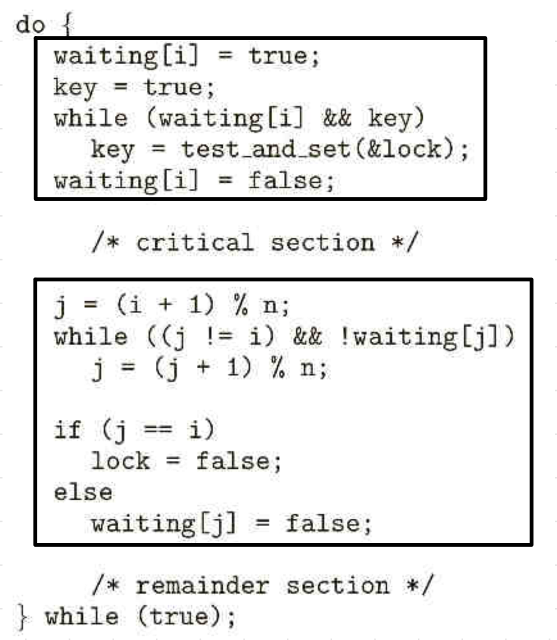
Implementation of Mutual Exclusion



```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Bounded waiting requirement not satisfied

Bounded Waiting Satisfied



```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

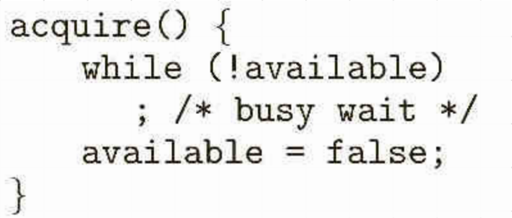
MUTEX LOCKS

- Software solution to CS problem
- Protect CS with acquire() and unlock with release() (both atomic)
- Requires busy waiting ; called spinlock (wastes CPU cycles)

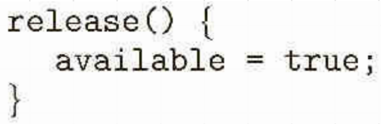
Spinlock Advantages

- no context switch when process must wait on lock
- when locks held for short times - useful
- multithreaded systems

acquire() and release()

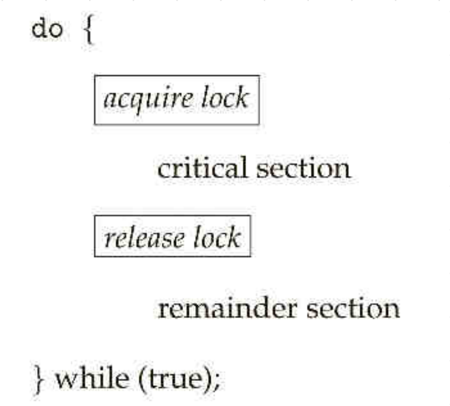


```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```



```
release() {
   available = true;
}
```

Solution to CS Problem Using Mutex Locks



```
do {
      acquire lock
          critical section
      release lock
          remainder section
} while (true);
```

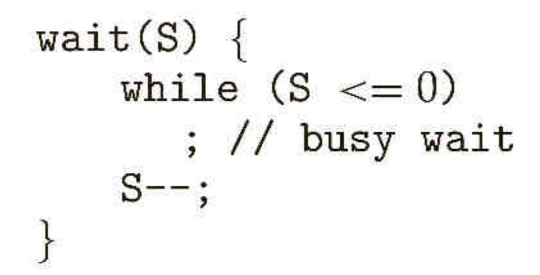
semaphore

- More sophisticated than mutex locks
- Semaphore S – integer variable
- To access : wait() and signal()

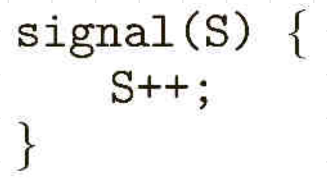
decrement, P

increment, V

DEFINITIONS



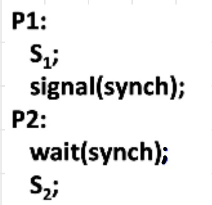
```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```



```
signal(S) {
    S++;
}
```

Semaphores

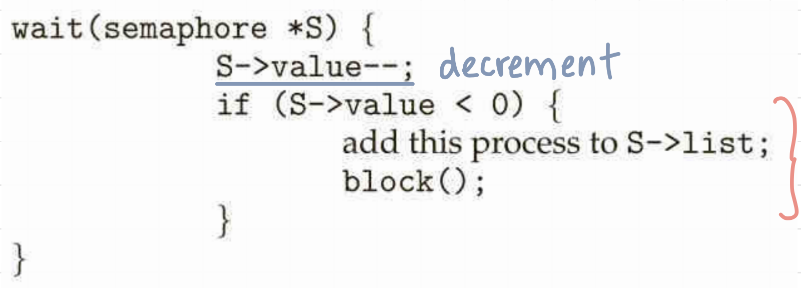
- Binary semaphore: integer value 0 or 1 ; same as mutex lock
- Integer semaphore: unrestricted domain
- Semaphore synch initialised to 0



P1:
  S_1;
  signal(synch);
P2:
  wait(synch);
  S_2;

Semaphore Implementation with no Busy Waiting

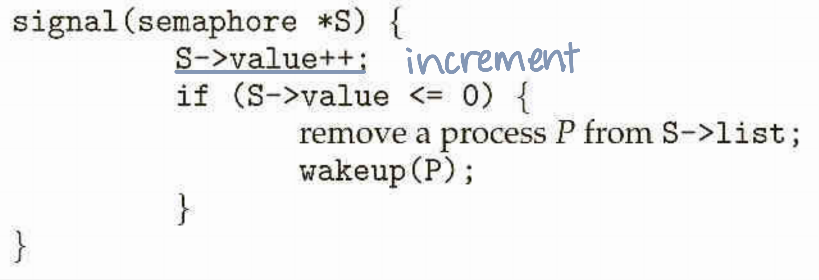
- associated waiting queue for semaphore
- block: place invoking process onto waiting queue
- wakeup: remove a process from waiting queue and place in ready queue



```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

decrement

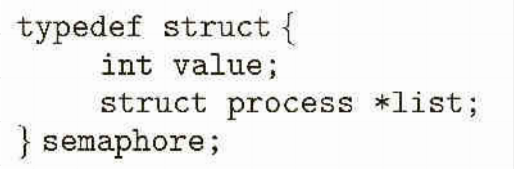
instead of busy wait



```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

increment

Semaphore struct

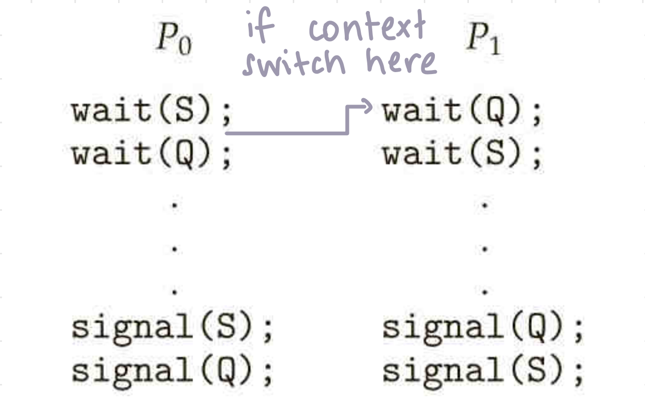


```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

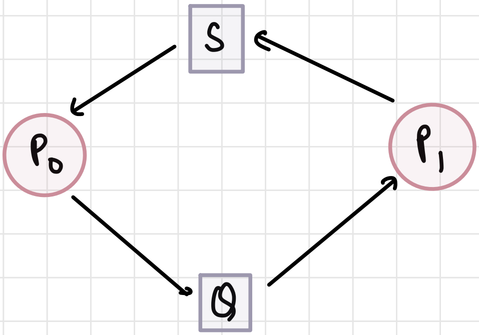
- Correct usage: first wait (mutex), then signal (mutex)

DEADLOCK

- Two or more processes waiting indefinitely for an event that can be caused only by one of the waiting processes
- Can lead to starvation; halts progress indefinitely
- Process never removed from semaphore queue
- Eg: let two processes P_0 and P_1 be trying to access two semaphores S and Q, both initialised to 1



```
      P0                      P1

wait(S);                wait(Q);
wait(Q);                wait(S);
    .                       .
    .                       .
    .                       .
signal(S);              signal(Q);
signal(Q);              signal(S);
```



deadlock

PRIORITY INVERSION

- Scheduling problem: low priority process holds lock required by high priority process
- Solution: priority inheritance protocol
- If several tasks are waiting for a resource, the task currently holding it is given priority

Example:

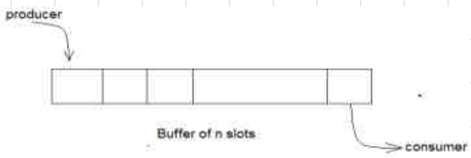
- Three processes: P1 priority 1, P2 priority 2, P3 priority 3
- P3 holding semaphore S, P1 waiting for S
- Assume P3 preempted by P2; indirectly blocks S from P1
  - P3 still holding S
  - P1 cannot access S
- To prevent: priority inheritance protocol

Priority Inheritance Protocol

- Intermediate tasks cannot preempt resource — avoiding priority inversion
- After releasing critical resource, priority set back to original priority
- Mars Sojourner in 1997: kept facing resets due to priority inversion —— solved by setting global variable to enable Priority inheritance on all semaphores

Classic Problems of Synchronisation

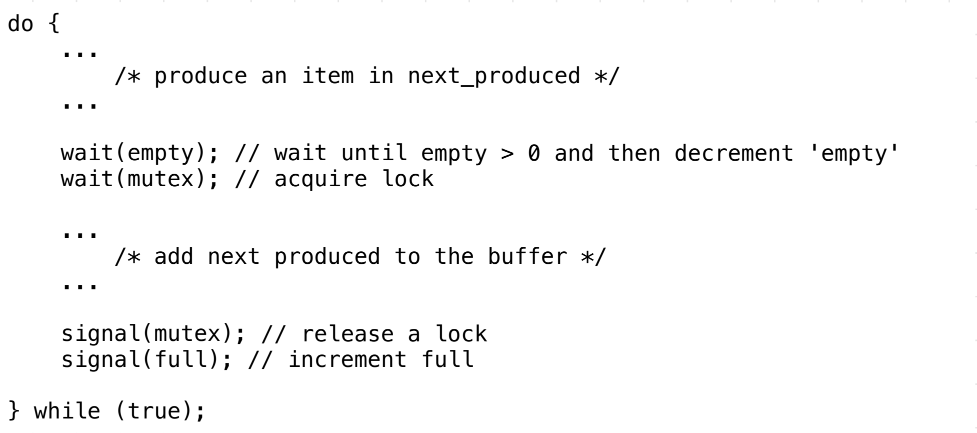
1. Bounded Buffer Problem



- n buffers
- each holds one item
- semaphore mutex: init 1 → lock
- semaphore full: init 0 → no. of full buffers
- semaphore empty: init n → no. of empty buffers

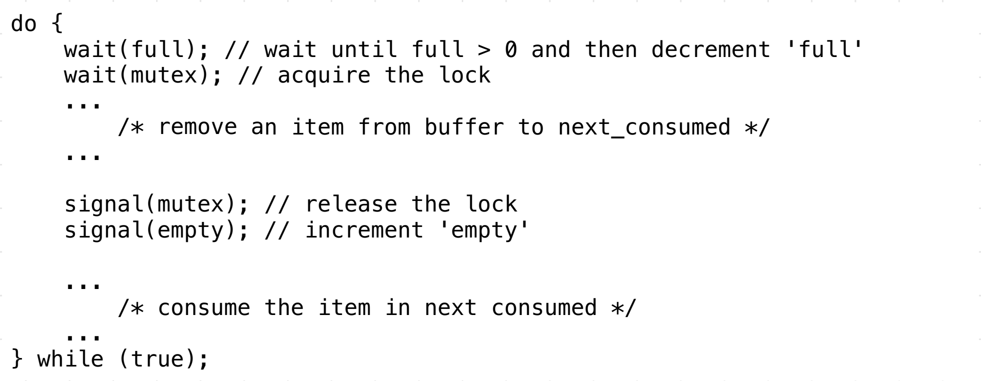
- Producer and consumer share above resources

Producer



```
do {
    ...
        /* produce an item in next_produced */
    ...

    wait(empty); // wait until empty > 0 and then decrement 'empty'
    wait(mutex); // acquire lock

    ...
        /* add next produced to the buffer */
    ...

    signal(mutex); // release a lock
    signal(full); // increment full

} while (true);
```

Consumer



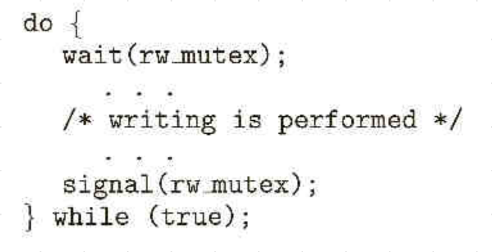
```
do {
    wait(full); // wait until full > 0 and then decrement 'full'
    wait(mutex); // acquire the lock
    ...
        /* remove an item from buffer to next_consumed */
    ...

    signal(mutex); // release the lock
    signal(empty); // increment 'empty'

    ...
        /* consume the item in next consumed */
    ...
} while (true);
```

2. Readers - Writers Problem

- Shared resource to be accessed by multiple processes
- Process: reader or writer
- Any number of readers can read simultaneously
- One writer at a time
- During write, no other read or write allowed
- No write if read happening
- File/ database

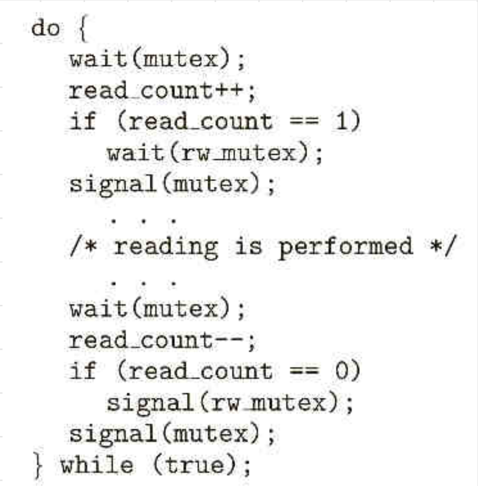
- dataset
- semaphore rw_mutex: init 1 → binary
- semaphore mutex: init 1 → lock
- integer read-count: init 0 → no. of reading processes

Writer



```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Reader

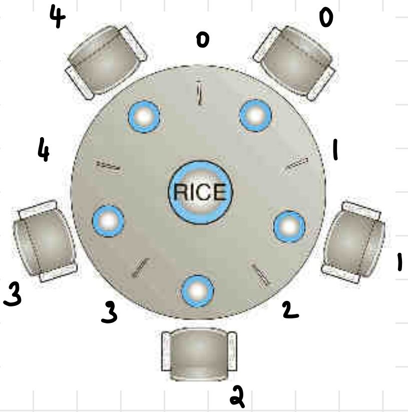


```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Variations

1. No reader kept waiting unless writer has permission to use shared object
2. Once writer ready, writes ASAP

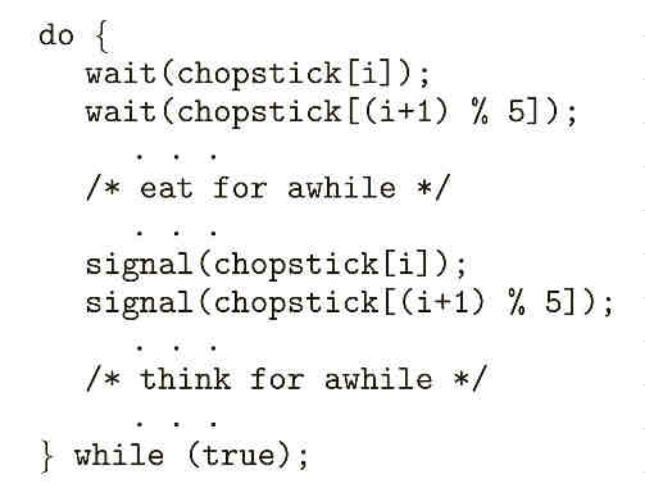
3. Dining Philosophers' Problem



- Philosophers sitting at circular table for dinner
- Pick up chopsticks one at a time
- Need both to eat
- Release both when done
- Case: 5 philosophers

- semaphore chopstick[5] : init 1
- bowl of rice (dataset)

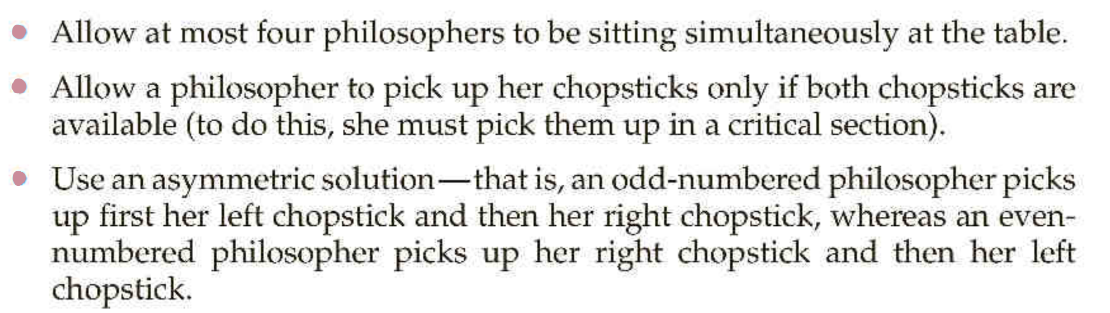
Philosopher i



```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

- No two neighbours can eat at once
- Can cause dealock (each picks up left chopstick: all starve)

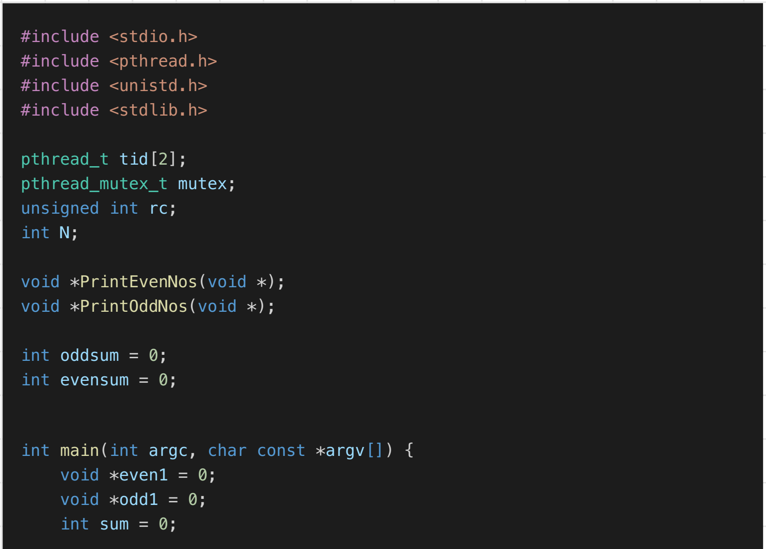
Remedies to Deadlock Problem



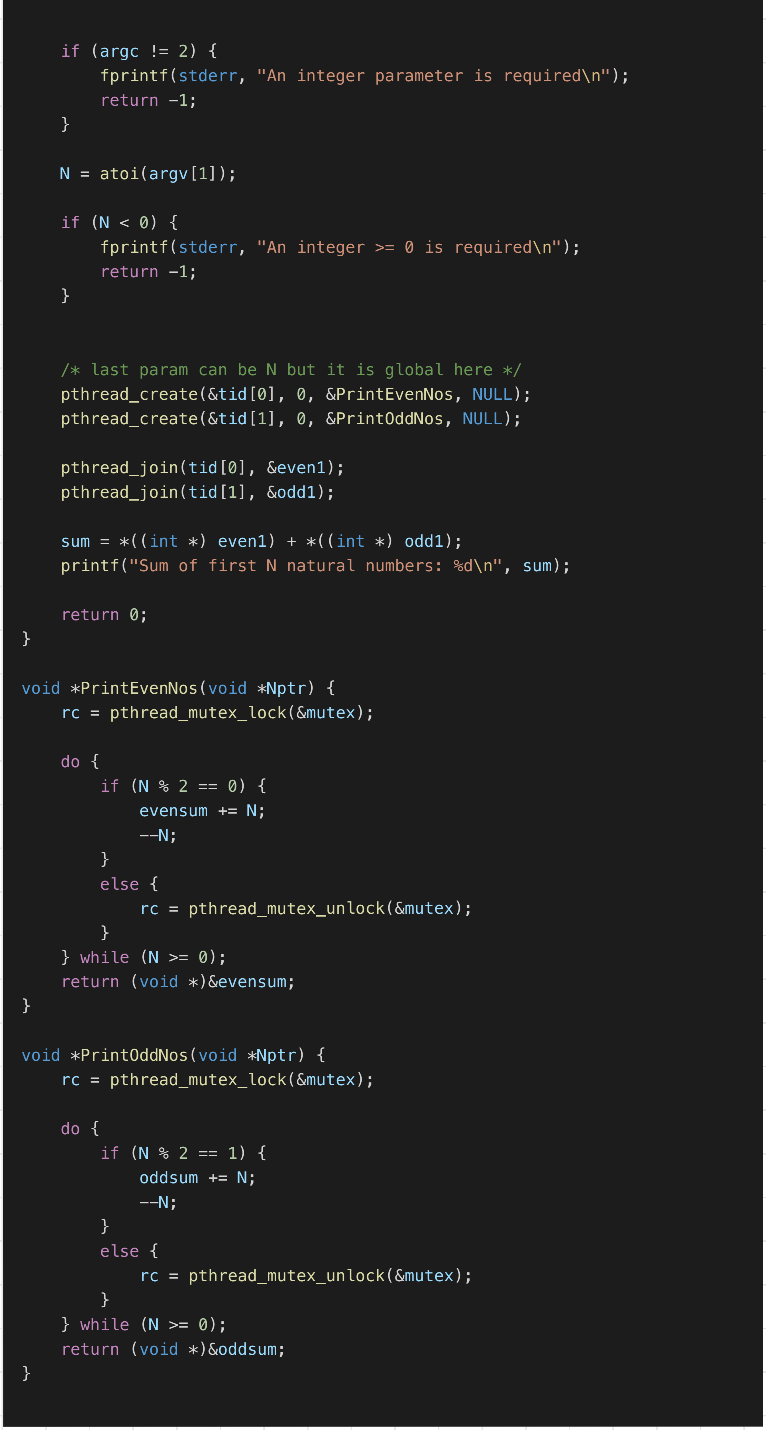
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

PROGRAMMING EXAMPLE

- Sum of n natural numbers
- Thread 1: evensum
- Thread 2: odd sum



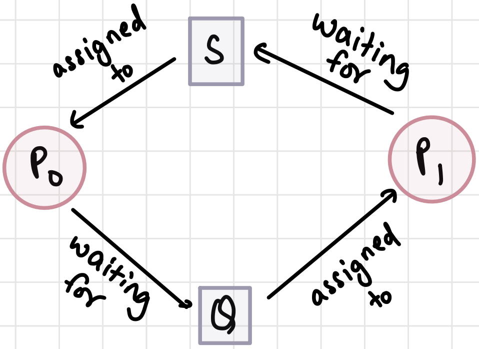
```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

pthread_t tid[2];
pthread_mutex_t mutex;
unsigned int rc;
int N;

void *PrintEvenNos(void *);
void *PrintOddNos(void *);

int oddsum = 0;
int evensum = 0;


int main(int argc, char const *argv[]) {
    void *even1 = 0;
    void *odd1 = 0;
    int sum = 0;
```

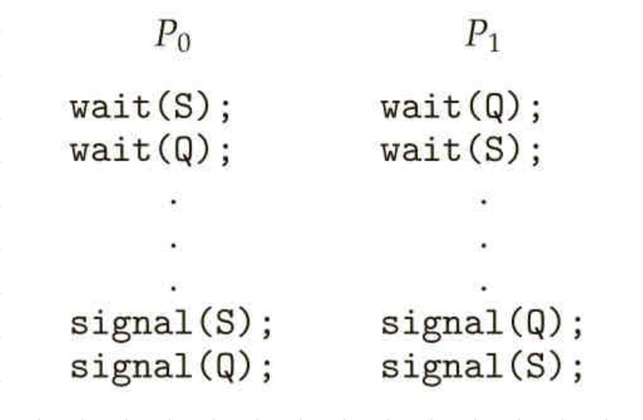


```
    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }

    N = atoi(argv[1]);

    if (N < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }


    /* last param can be N but it is global here */
    pthread_create(&tid[0], 0, &PrintEvenNos, NULL);
    pthread_create(&tid[1], 0, &PrintOddNos, NULL);

    pthread_join(tid[0], &even1);
    pthread_join(tid[1], &odd1);

    sum = *((int *) even1) + *((int *) odd1);
    printf("Sum of first N natural numbers: %d\n", sum);

    return 0;
}

void *PrintEvenNos(void *Nptr) {
    rc = pthread_mutex_lock(&mutex);

    do {
        if (N % 2 == 0) {
            evensum += N;
            --N;
        }
        else {
            rc = pthread_mutex_unlock(&mutex);
        }
    } while (N >= 0);
    return (void *)&evensum;
}

void *PrintOddNos(void *Nptr) {
    rc = pthread_mutex_lock(&mutex);

    do {
        if (N % 2 == 1) {
            oddsum += N;
            --N;
        }
        else {
            rc = pthread_mutex_unlock(&mutex);
        }
    } while (N >= 0);
    return (void *)&oddsum;
}
```

DEADLOCKS

- Several processes competing for same resource
- Process holding a resource and waiting for another resource that is being held by another process
- Finite number of resources
- Each resource type R_i has W_i instances
- Each process: request(), use(), release() resource





```
        P0                  P1

  wait(S);            wait(Q);
  wait(Q);            wait(S);
      .                   .
      .                   .
      .                   .
  signal(S);          signal(Q);
  signal(Q);          signal(S);
```

Conditions that create Deadlock

- If four conditions met by 2 processes simultaneously

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

RESOURCE ALLOCATION GRAPH

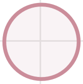
- Directed graph describing resource allocation

- $G(V,E)$, V partitioned into $P = \{P_1, P_2 \dots P_n\}$ processes and $R = \{R_1, R_2 \dots R_n\}$ resources

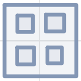
- Request edge: $P_i \rightarrow R_j$

- Assignment edge: $R_j \rightarrow P_i$

- Symbols

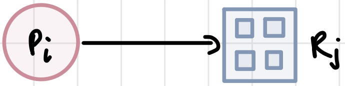


process P_i

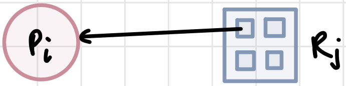


resource R_j with 4 instances

- P_i requests for instance of R_j

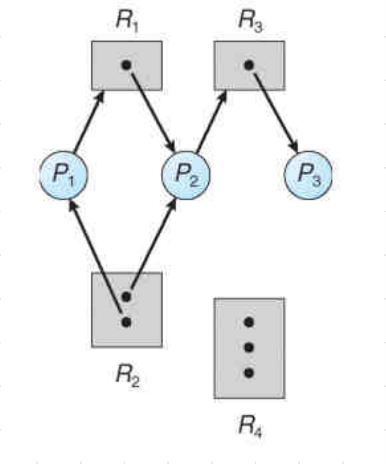


- P_i holding instance of R_j

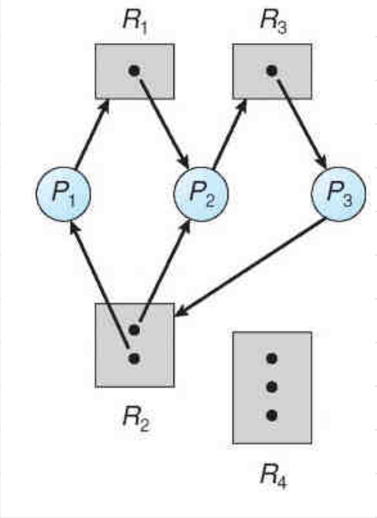


Example:

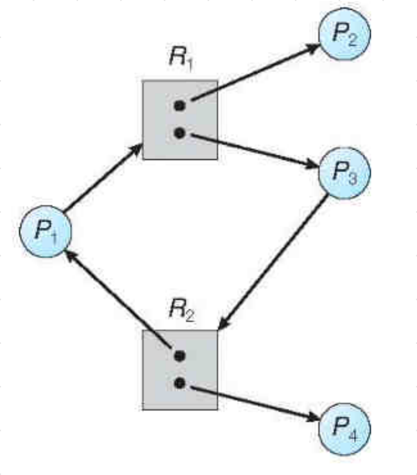
1. No deadlock - no cycle



2. Deadlock - cycle



3. No deadlock - cycle

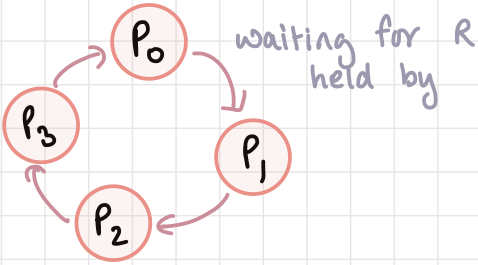


Handling Deadlocks

1. Prevention/ avoidance
2. Detect & recover
3. Ignore

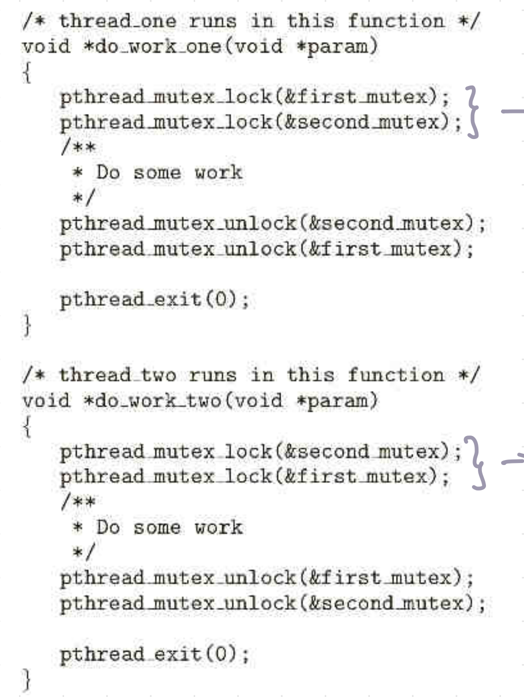
1. Circular Wait

- Each resource given a numeric value ; $R = \{R_1, R_2 \ldots R_m\}$
- Processes must request for resources in increasing order of value
- If a process is holding a resource , eg, R_5, and makes a request for R_2 , the request will not be granted
- Protocol 1: Process makes request for R_i and then R_j. Request allowed only if $F(R_j) > F(R_i)$ (where $F: R \rightarrow N$)
- Protocol 2: Process requesting resource R_j must have released all resources R_i such that $F(R_i) \geq F(R_j)$
- If protocols followed, circular wait will not hold



- eg: F(HD)=5 , F(printer) = 12, F(tape drive)=1 etc

- Must not access mutex locks in different orders

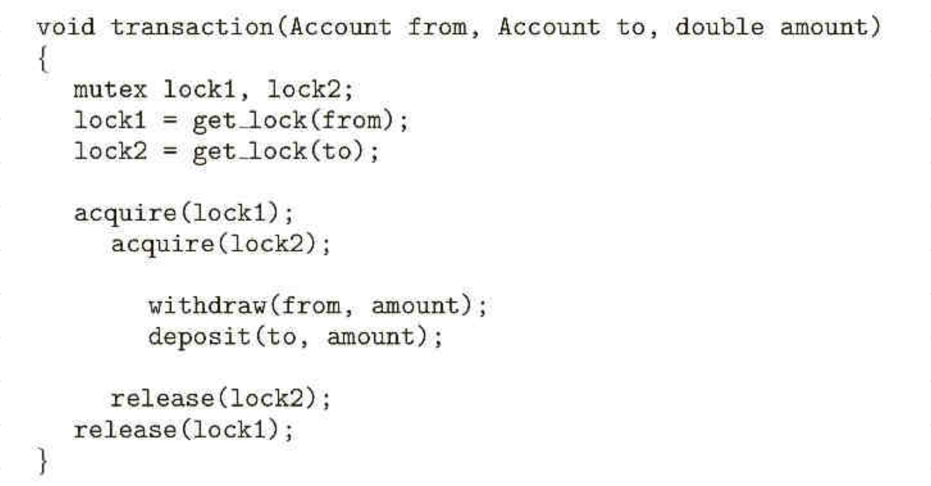


```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

→ correct

→ wrong

Simultaneous A → B and B → A: dealock



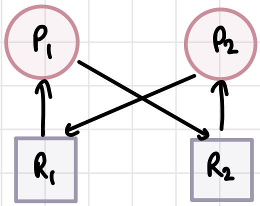
```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```

2. Mutual Exclusion

- To invalidate, some resources should be shareable
- Some will still be non-shareable (printer etc)

3. Hold and wait

- To invalidate, resources cannot make request if they are already holding on to a resource
- Can start execution only after all resources have been allocated (low utilisation of resources – limitation)



holding & requesting; if preempted & releases resource, deadlock solved

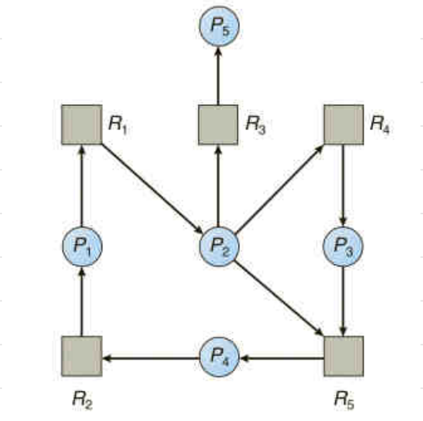
4. No preemption

- Make resources preemptible
- If process makes request for unavailable resource, all its resources are released
- Added to list of resources for which process is waiting
- Can restart process only after all required resources allocated

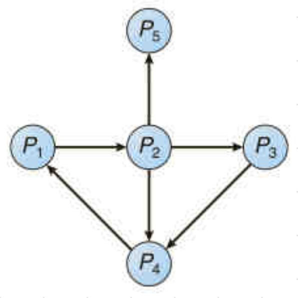
DEADLOCK DETECTION

System : One Instance of Each Resource

- Maintain wait-for graph
- $P_i \xrightarrow{edge} P_j \Rightarrow P_i$ waiting for P_j to release resource
- Periodically search for cycles in a graph
- $O(n^2)$ operations ⟶ n : verices



resource allocation graph



corresponding wait-for graph

- Can use DFS

System: Many Instances of Each Resource

- Available: vector of length m indicating no. of available resources of each type

- Allocation: n×m matrix defining no. of resources of each type currently allocated to each process

- Request: n×m matrix indicating current request of each process. Request[i][j] = k means P_i is requesting k more instances of R_j

Detection Algorithm

1. Let Work[m] and Finish[n] be vectors

   - Initialise Work = Available

   - for $i = 1, 2, \ldots, n$, if Allocation[i] $\neq 0$, then Finish[i] = false, else Finish[i] = true

2. Find an index i such that both

   (a) Finish[i] == false

   (b) Request[i] $\leq$ Work

   if no such i, go to step 4

3. Work = Work + Allocation[i]
   Finish[i] = true

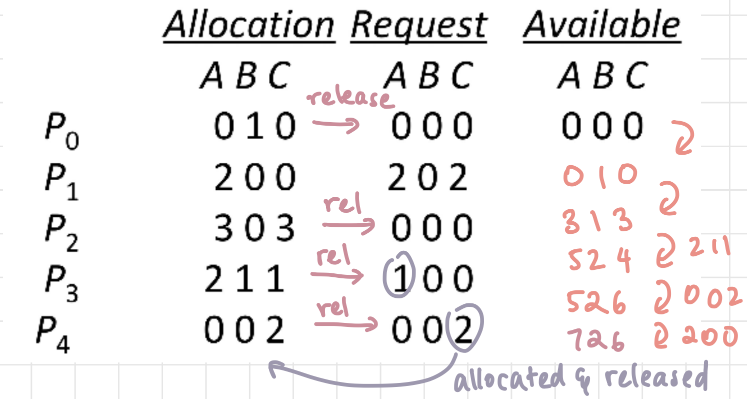
   go to step 2

4. If Finish[i] == false, for some i 1≤i≤n, then the system is in deadlock state (P$_i$ deadlocked)

- O(m x n^2) operations ; execute once an hour or so

Eg: Five Processes P$_0$ to P$_4$, A(7 instances), B(2 instances), C(6 instances)

Snapshot at time T$_0$



	Allocation		Request	Available	
	A B C		A B C	A B C	
P_0	0 1 0	release →	0 0 0	0 0 0	
P_1	2 0 0		2 0 2	0 1 0	
P_2	3 0 3	rel →	0 0 0	3 1 3	
P_3	2 1 1	rel →	1 0 0	5 2 4	2 1 1
P_4	0 0 2	rel →	0 0 2	5 2 6 7 2 6	0 0 2 2 0 0

allocated & released

<P$_0$, P$_2$, P$_3$, P$_4$, P$_1$>

- Availability = initial

- ∴ no deadlock

- If P$_2$ = 001, deadlock