UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

Processes, Threads, Concurrency, and Synchronization

Carlos Alberto Rincón Castro | carincon@uh.edu

Lecture Overview

- **Introduction:** Processes vs. Threads
  - Design Aspects
  - Concurrency
- **Threads:** Implementation Aspects
  - User-level vs. Kernel-level
  - Multithreading patterns
- **Synchronization**
  - Mutual Exclusion
  - Hardware-based solutions
  - Software-based solutions
- **Threads in the real world**
  - Synchronization and the OS
  - Software solutions based on Threads

Processes vs. Threads

Process: is an instance of a program running on a computing system.

Process = Code + Data + Memory + Process Control Block

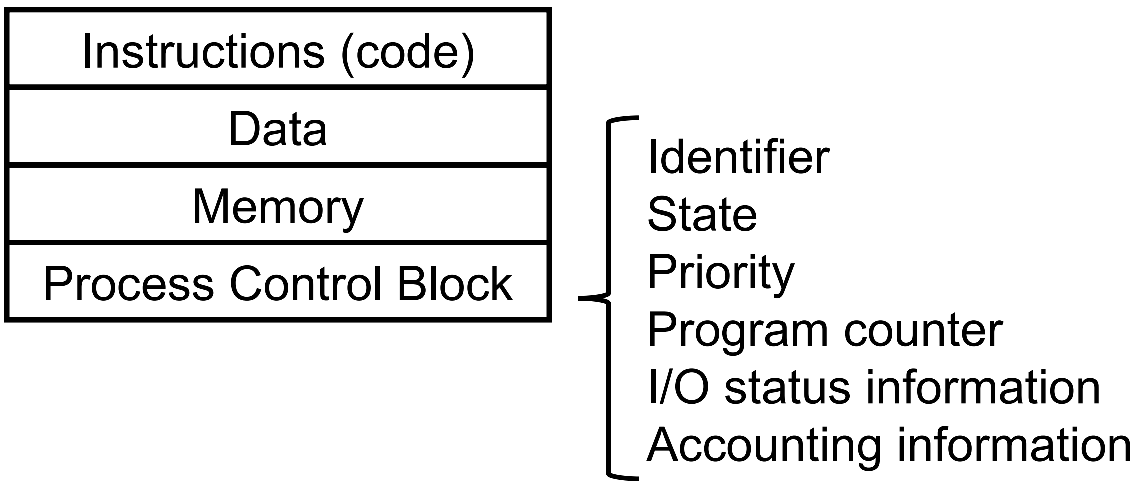
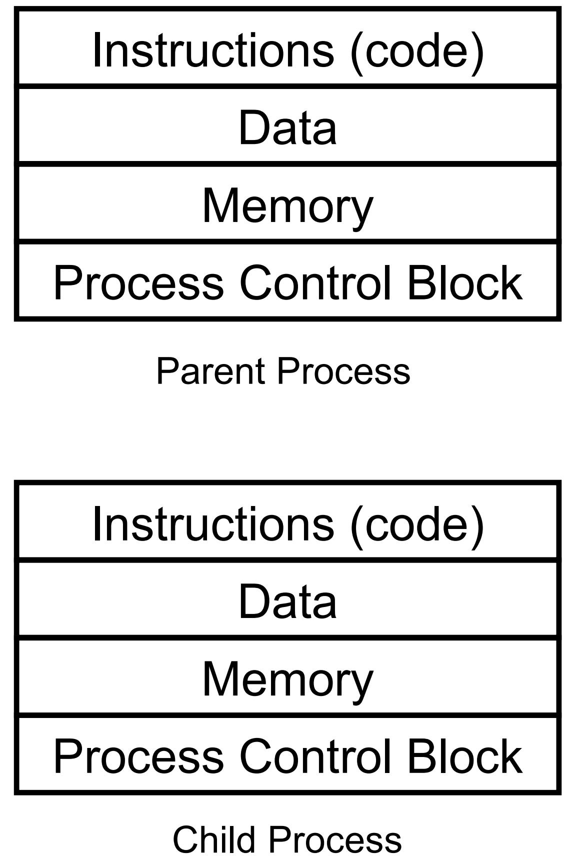


Fig 1. Simplified Process Image

Processes vs. Threads

Fork(): creating a process dynamically.

```
int main ()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) // Child Process Code
      printf(”Child process\n”);
   else
      printf(”Parent process\n”);
   return (0);
}
```



Processes do not share their memory address space, but they share the opened files.

Processes vs. Threads

Threads:

- 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.

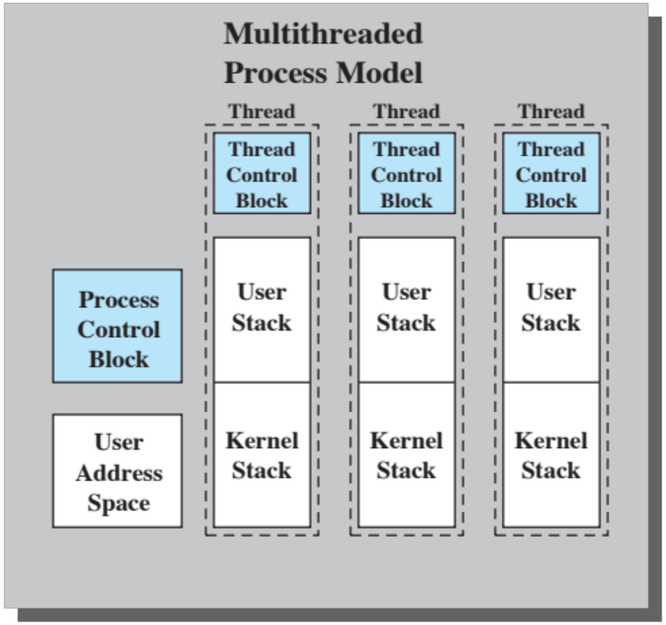


Fig 2. Simplified Thread Image (Stallings, OS8e)

Processes vs. Threads

Threads Challenges:

- No memory protection inside an address space.
- Lightweight processes can now interfere with each other.

```
void * child ()
{
   printf(”Child process\n”);
}

int main ()
{
   thread_t tid;
   pthread_create(&tid, NULL, child, NULL);
   printf(”Parent process\n”);
   return (0);
}
```

Stack Address (NULL = anywhere)

Parameters (void pointer)

POSIX Thread Example

Processes vs. Threads

Threads Example:

```
#define COUNT 5

void *inc_x(void *x_void_ptr)
{
        int *x_ptr = (int *)x_void_ptr;
        for (int i=0; i< COUNT; i++)
                (*x_ptr)++;
        return NULL;
}
```

```
(*x_ptr)++  {  load  reg, M
               inc reg
               store reg, M
```

Thread 1 loads 0
Threads 3, 4, 5 run to completion
Thread 2 runs through 4 of its 5 iterations.
Thread 1 increments 0 to 1 and stores the result (1).
Thread 2 loads 1
Thread 1 runs to completion
Thread 2 increments 1 to 2 and stores 2

POSIX Thread Example

Threads: Implementation

User-Level Threads (ULTs):

- User-level threads are managed by procedures within the task address space by the thread library.
- The kernel is not aware of the existence of threads.
- Example: POSIX Threads (POSIX = Portable OS interface)

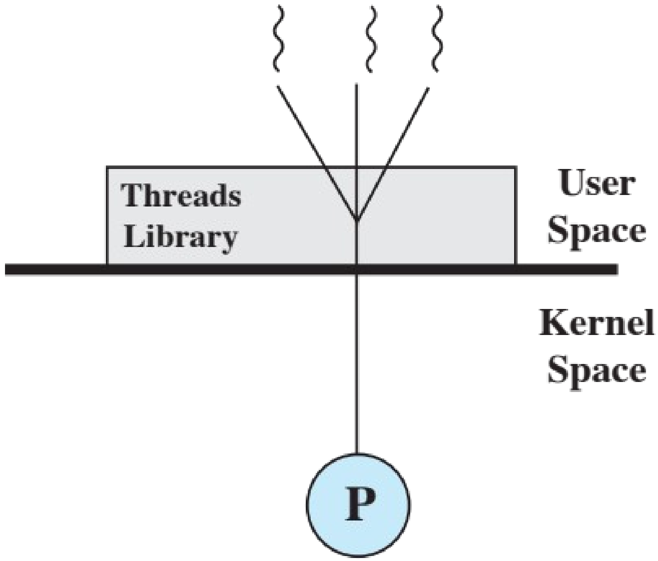
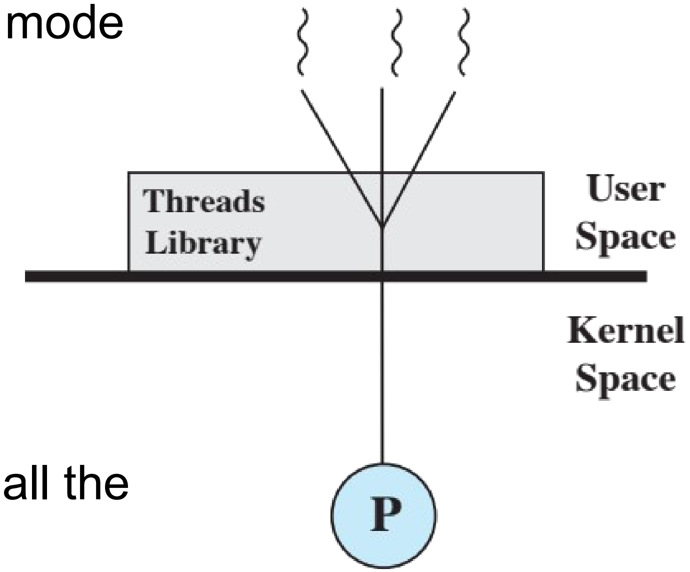


Fig 3. User-level Threads (Stalling OS8e)

Threads: Implementation

User-Level Threads (ULTs):

- **Advantages:**
  - Thread switching does not require kernel mode privileges (no performance penalty).
  - Scheduling can be application specific.
  - Can be implemented on any OS.



- **Disadvantages:**
  - A blocking call from one thread will block all the threads from the same process.
    Solution: Using non-blocking calls (not easy).
  - Multithreaded programs cannot take advantage of multiprocessing

Threads: Implementation

Kernel-Level Threads (KLTs):

- Thread management is done by the kernel (one process table entry per thread).
- Best solution for multiprocessor architectures (kernel can allocate several processors to a single multithreaded task).

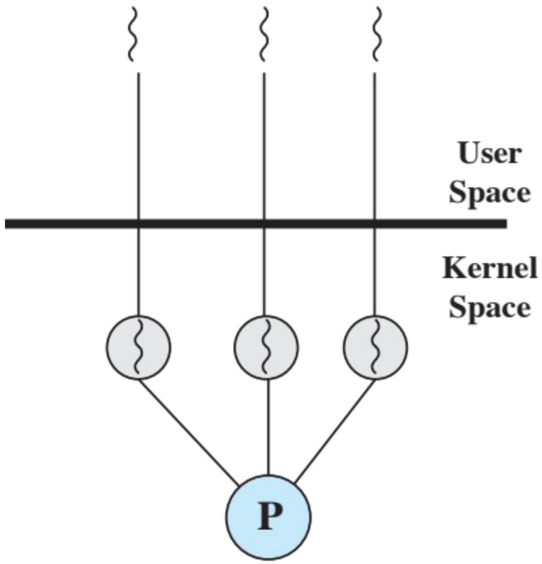
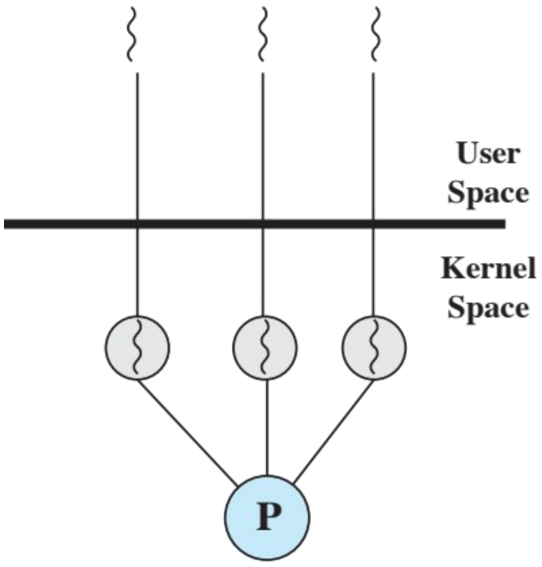


Fig 4. Kernel-level Threads (Stalling OS8e)

Threads: Implementation

Kernel-Level Threads (KLTs):

- **Advantages:**
  - The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  - If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- **Disadvantages:**
  - Switching from one thread to another within the same process requires two context switches (overhead).

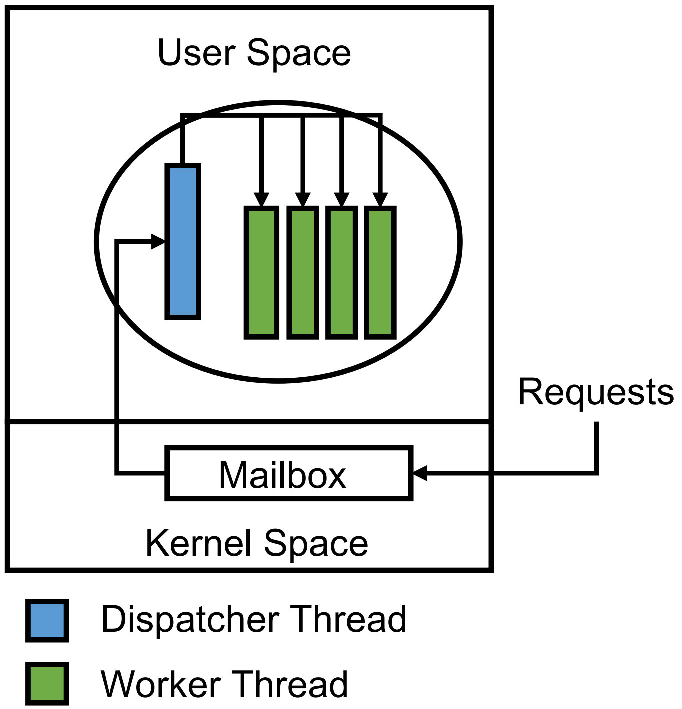


Multithreading patterns:

Dispatcher-Worker pattern:

- Dispatcher Thread: reads the requests from the mailbox.
- Worker Thread: performs the task.

- **Challenges:**
  - Dispatcher can be a bottleneck.
  - Static / Dynamic Worker generation.



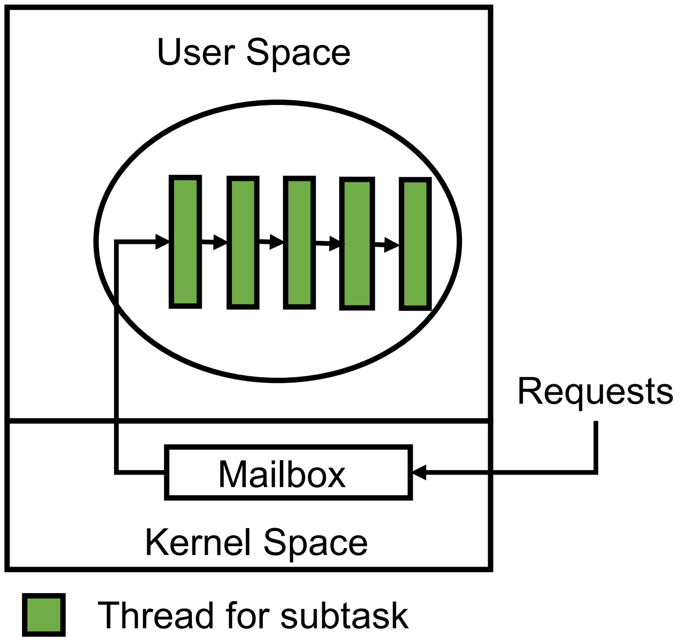
Threads: Implementation

Multithreading patterns:

Pipeline pattern:

- Threads assigned one subtask in the system.
- Entire task = Pipeline of threads.
- Multiple tasks concurrently run in the system, in different pipeline stages
- Shared buffer based communication between stages.

- **Challenges:**
  - Performance depends on weakest link.
  - Dividing requests in subtasks.



Multithreading patterns:

Example:

For 5-step toy order application, we have the solutions using:

- Dispatcher-Worker pattern.
- Pipeline pattern.

Constraints:

- For both patterns, we have 5 threads.
- For the dispatcher-workers solution, a worker produces a toy order in 100 ms.
- For the pipeline solution, each one of the five stages takes 20 ms.
- Time to read and process the request is negligible.

How long will it take for these solutions to complete an 8-toy order and a 9-toy order?

Threads: Implementation

Multithreading patterns:

Dispatcher-Worker pattern

8-toy order:

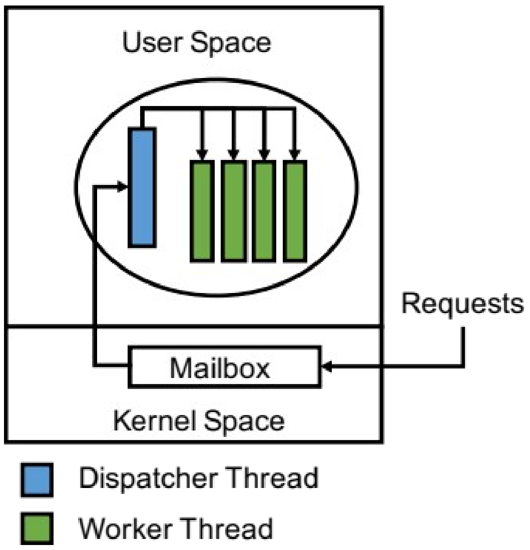
0 ms (reading + processing) +
100 ms (processing 4 toy orders concurrently) +
100 ms (processing 4 toy orders concurrently) =

Total time = 200 ms.

9-toy order:

0 ms (reading + processing) +
100 ms (processing 4 toy orders concurrently) +
100 ms (processing 4 toy orders concurrently) +
100 ms (processing 1 toy order)

Total time = 300 ms



Multithreading patterns:

Pipeline pattern

8-toy orders:

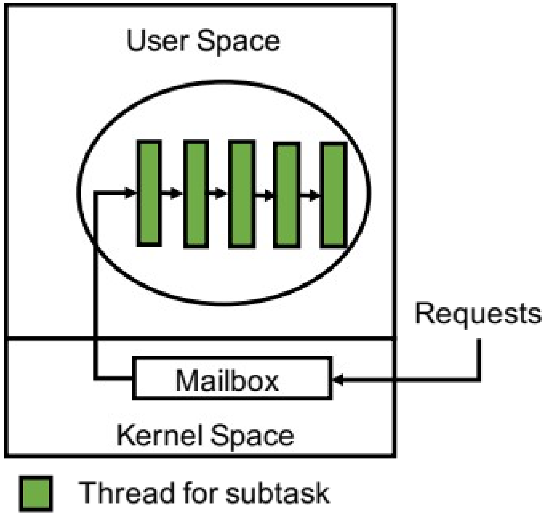
0 ms (reading + processing) +
100 ms (time to finish first toy order) +
20 ms * 7 (processing 7 toy orders) =

Total time = 240 ms.

9-toy orders:

0 ms (reading + processing) +
100 ms (time to finish first toy order) +
20 ms * 8 (processing 8 toy orders) =

Total time = 260 ms.



Threads: Synchronization

Mutual Exclusion:

Key Terms:

- **Atomic operation:** a function or action implemented as a sequence of one or more instructions that cannot be interrupted (indivisible).
- **Critical Section:** section of code within a process that requires access to shared resources that has to be executed as an atomic operation.
- **Mutual Exclusion:** property that guarantees that only one thread of execution will have access to a critical section for a particular shared resource at a particular time.
- **Racing Condition:** a situation when multiple threads or processes try to gain access to a particular shared resource at the same time.

Threads: Synchronization

Mutual Exclusion:

Challenges:

- **Concurrency**
  - Detecting programming errors.
  - Difficult for the OS to manage the allocation of resources optimally.
- **Busy waits**

  Example: while(TRUE); Wasting CPU cycles.
- **Deadlocks**

  Permanent blocking of a set of processes that compete for the same shared resource.
- **Starvation**

  Postponing indefinitely the execution of a process based on a condition.

Threads: Synchronization

Mutual Exclusion:

Requirements:

- A process that halts must do so without interfering with other processes.
- No deadlock or starvation.
- A process must not be denied access to a critical section when there is no other process using it.
- No assumptions are made about relative process speeds or number of processes.
- A process remains inside its critical section for a finite period of time.

Mutual Exclusion: Hardware Solutions

Interrupt Disabling

- Disabling interrupts guarantees mutual exclusion

```
int main()
{
        disableInterrupt();
        // Critical section goes here
        enableInterrupt();
}
```

- Disadvantages:
  - The performance of the system could be noticeably degraded.
  - Not suitable for Multiprocessor platforms.

Threads: Synchronization

Mutual Exclusion: Hardware Solutions

Special Machine Instructions

- Test and Set
- Exchange

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;

void main()
{
        bolt = 0;
        parbegin (P(1), P(2), . . . ,P(n));
}
```

```
void P(int i)
{
  while (true)
  {
    while (!test_and_set(&bolt))
      /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* non-critical section */;
  }
}
```

- Carried out atomically.

Mutual Exclusion: Hardware Solutions

Special Machine Instructions

- Advantages:
  - Applicable to uniprocessors and multiprocessors.
  - Simple and easy to verify.
  - Supports multiple critical sections.

- Disadvantages:
  - Busy-wait is employed.
    - We have to avoid busy waits on single-core architectures.
    - We can use them only for short waits on multicore architectures.
  - Starvation is possible.

Mutual Exclusion: Software Solutions

Peterson’s Algorithm

```
#define LOCKED 1
#define UNLOCKED 0
int lock = UNLOCKED; // shared

int main()
{
   while (lock == LOCKED); // busy wait
   lock = LOCKED;
   Critical_Section();
   lock = UNLOCKED;
   return(0);
}
```

Solution:

Move Lock=LOCKED before
while (lock == LOCKED); // busy wait

Result:

Deadlock

Mutual Exclusion: Software Solutions

Peterson's Algorithm

```
// shared variables
int reserved[2] = {FALSE, FALSE};
int mustwait; // tiebreaker
int main()
{
  int other= 1 - pid; //other process
  reserved[pid] = TRUE;
  must_wait = pid; // set tiebraker
  while (reserved[other] && must_wait==pid);
  // Critical Section
  reserved[pid] = FALSE;
  return(0);
}
```

Solution:

When two processes arrive in lockstep, last one must wait.

Result:

Mutual Exclusion

Mutual Exclusion: Software Solutions

Semaphores

- Special integer variables.
- May be initialized to a nonnegative integer value.
- Two atomic primitives
  - SemWait operation decrements the value (P() or Down)
  - The semSignal operation increments the value (V() or Up).

Mutual Exclusion: Software Solutions

Semaphores

- Normal implementation:
  - Processes waiting for a semaphore whose value is zero are put in the blocked state.
  - Busy waits are eliminated.
- Strong Semaphores: FIFO order
- Weak Semaphores: No specific order.

Mutual Exclusion: Software Solutions

Semaphores

- Producer – Consumer Problem: only one producer or consumer may access the buffer at any one time.

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer()
{
   while (true)
   {     produce();
         semWaitB(s);
         append();
         n++;
         if (n==1)
            semSignalB(delay);
         semSignalB(s);
   }
}
```

```
void consumer()
{
   semWaitB(delay);
   while (true)
   {
      semWaitB(s);
      take();
      n--;
      semSignalB(s);
      consume();
      if (n==0)
         semWaitB(delay);
   }
}
```

Semaphores: Producer - Consumer

	Producer	Consumer	s	n	Delay
1			1	0	0
2	`semWaitB(s)`		0	0	0
3	`n++`		0	1	0
4	`if (n==1) (semSignalB(delay))`		0	1	1
5	`semSignalB(s)`		1	1	1
6		`semWaitB(delay)`	1	1	0
7		`semWaitB(s)`	0	1	0
8		`n--`	0	0	0
9		`semSignalB(s)`	1	0	0
10	`semWaitB(s)`		0	0	0
11	`n++`		0	1	0
12	`if (n==1) (semSignalB(delay))`		0	1	1
13	`semSignalB(s)`		1	1	1
14		`if (n==0) (semWaitB(delay))`	1	1	1
15		`semWaitB(s)`	0	1	1
16		`n--`	0	0	1
17		`semSignalB(s)`	1	0	1
18		`if (n==0) (semWaitB(delay))`	1	0	0
19		`semWaitB(s)`	0	0	0
20		`n--`	0	–1	0
21		`semSignalB(s)`	1	–1	0

Fig 5. Producer - Consumer (Stalling OS8e)

Semaphores: Producer - Consumer

Solution 1

```
void consumer()
{
   semWaitB(delay);
   while (true)
   {
      semWaitB(s);
      take();
      n--;
      m=n;
      semSignalB(s);
      consume();
      if (m==0)
         semWaitB(delay);
   }
}
```

Solution 2

```
semaphore n = 0;
semaphore s = 1;
```

```
void producer()
{
   while (true)
   {
      produce();
      semWait(s);
      append();
      semSignal(s);
      semSignal(n);
   }
}
```

```
void consumer()
{
   while (true)
   {
      semWait(n);
      semWait(s);
      take();
      semSignal(s);
      consume();
   }
}
```

Mutual Exclusion: Software Solutions

Monitors

- Software module consisting of one or more procedures, an initialization sequence, and local data.
- Local data variables are accessible only by the monitor’s procedures and not by any external procedure.
- Process enters monitor by invoking one of its procedures.
- Only one process may be executing in the monitor at a time.

Mutual Exclusion: Software Solutions

Monitors

- Implemented in a number of programming languages including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java

- Achieved by the use of condition variables using the primitives:
    - **cwait(c):** suspend execution of the calling process on condition c.
    - **csignal(c):** resume execution of some process blocked after a cwait(c). (Hoare semantics)
    - **cnotify(c):** resume execution of all processes waiting for condition c. (Mesa semantics).

Mutual Exclusion: Software Solutions

Monitors

Producer – Consumer problem:

```
void append (char x)
{
     while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                   /* one more item in buffer */
     cnotify(notempty);                     /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                  /* one fewer item in buffer */
     cnotify(notfull);                      /* notify any waiting producer */
}
```

Fig 6. Producer – Consumer with Mesa semantics (Stalling OS8e)

Synchronization and the OS

Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses spinlocks on multiprocessor systems (Spinlocking-thread will never be preempted).
- Also provides dispatcher objects (user-space) which may act as mutexes, semaphores, events, and timers (an event acts much like a condition variable).

Threads in the real world

Synchronization and the OS

Linux:

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections.

- Version 2.6 and later, fully preemptive

- Linux provides:
  - Semaphores
  - Spinlocks
  - Reader-writer locks

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption.

Synchronization and the OS

PThreads:

- Pthreads API is OS-independent
- It provides:
  - Mutex locks
  - Condition variables
- Non-portable extensions include:
  - Read-write locks
  - Spinlocks

Threads in the real world

Alternative Aproaches

- **Transactional Memory**

  A memory transaction is a sequence of read-write operations to memory that are performed atomically.

- **OpenMP**
  - API that support parallel programming.
  - #pragma omp critical directive is treated as a critical section and performed atomically.

- **Functional Programming Languages**
  - Functional programming languages do not maintain state.
  - Variables are treated as immutable and cannot change state once they have been assigned a value.
  - Examples: Earlang and Scala

Threads in the real world

Applications

Core: basic server skeleton

Modules: per functionality

Flow of Control: Similar to Event Driven model

A combination of MP + MT
Each process = dispatcher/worker with dynamic thread pool
Number of processes can also be dynamically adjusted.

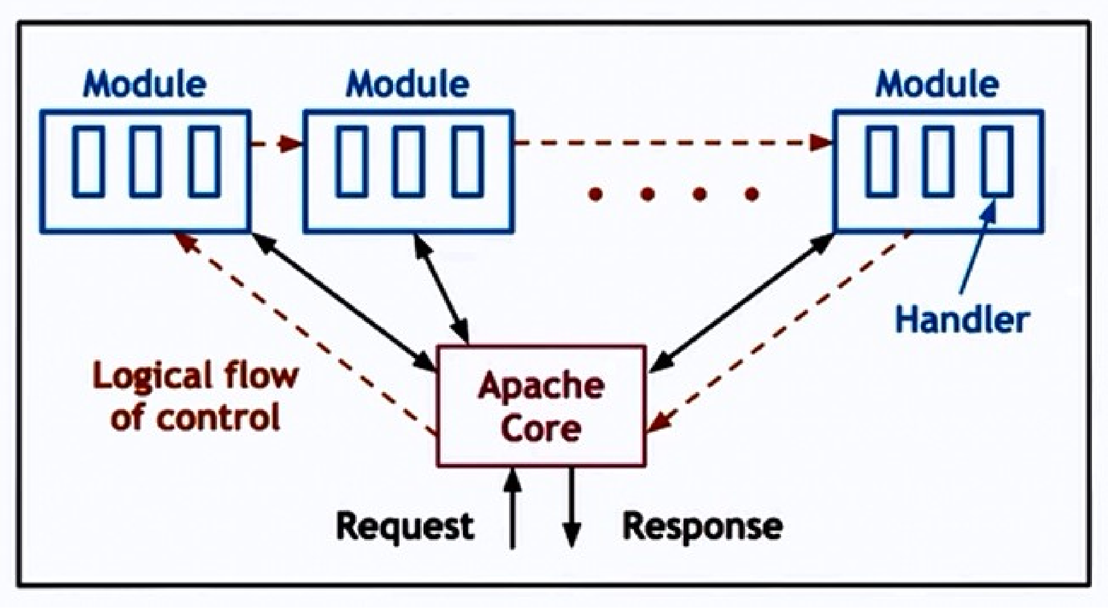


Fig 6. Producer – Apache Web Server (1)

1. https://applied-programming.github.io/Operating-Systems-Notes/3-Threads-and-Concurrency/#event-driven-model

Lecture Summary

- **Introduction:** Processes vs. Threads
  - Design Aspects
  - Concurrency
- **Threads:** Implementation Aspects
  - User-level vs. Kernel-level
  - Multithreading patterns
- **Synchronization**
  - Mutual Exclusion
  - Hardware-based solutions
  - Software-based solutions
- **Threads in the real world**
  - Synchronization and the OS
  - Software solutions based on Threads