CS370 Operating Systems

Midterm Review

Yashwant K Malaiya

Fall 2016

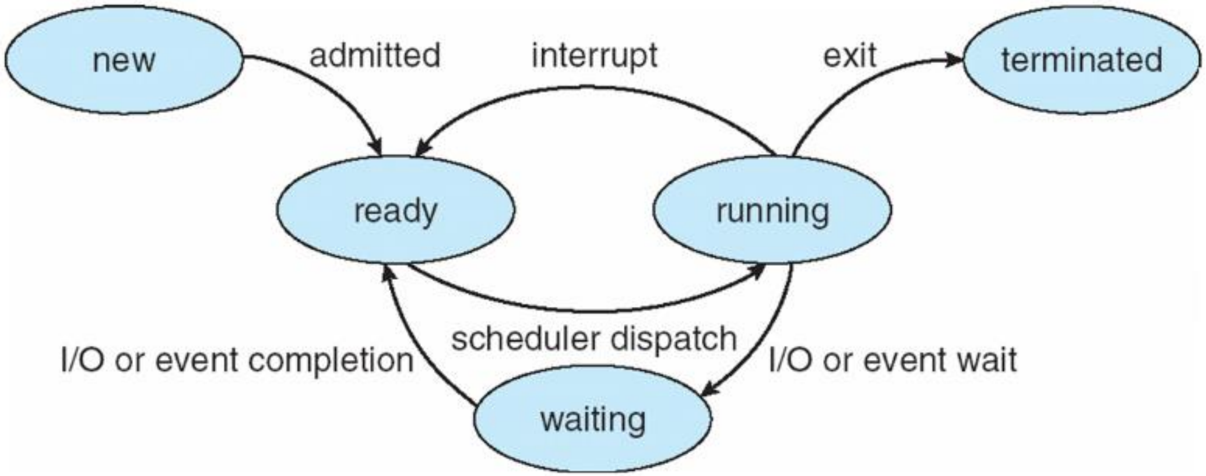
Colorado State University

Computer System Structures

- Computer System Operation
  - Stack for calling functions (subroutines)
- I/O Structure: polling, interrupts, DMA
- Storage Structure
  - Storage Hierarchy
- System Calls and System Programs
- Command Interpreter

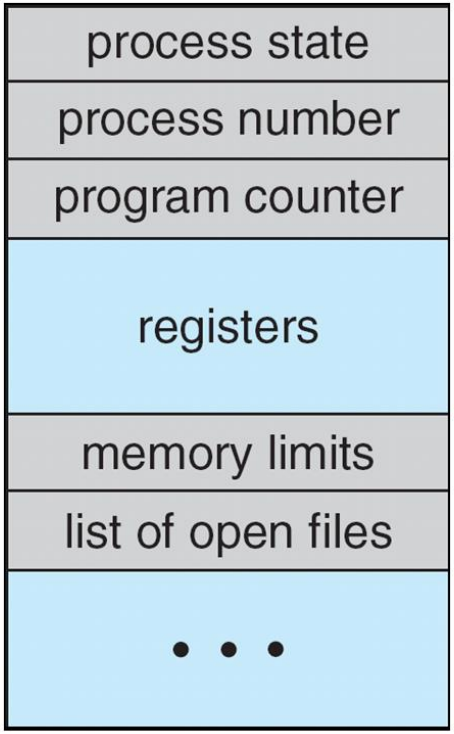
Process Concept

- Process - a program in execution
  - process execution proceeds in a sequential fashion
- Multiprogramming: several programs apparently executing "concurrently".
- Process States
  - e.g. new, running, ready, waiting, terminated.

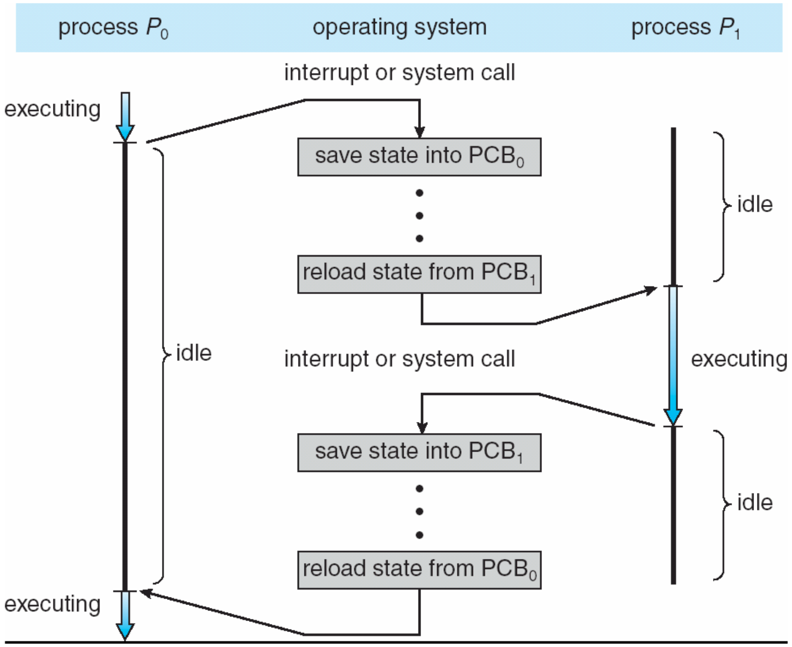


Process Control Block

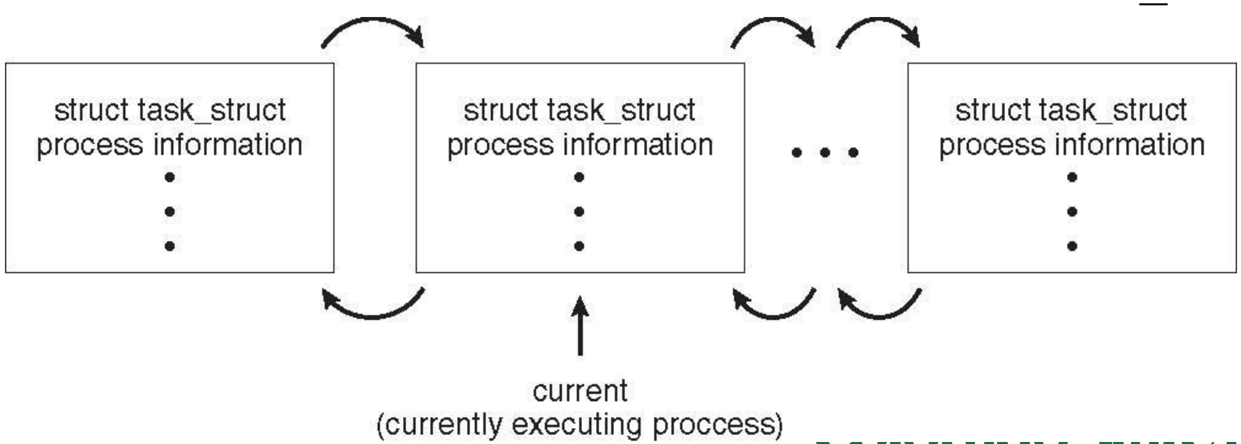
- Contains information associated with each process
  - Process State - e.g. new, ready, running etc.
  - Program Counter - address of next instruction to be executed
  - CPU registers - general purpose registers, stack pointer etc.
  - CPU scheduling information - process priority, pointer
  - Memory Management information
  - Accounting information
  - I/O Status information - list of I/O devices allocated



CPU Switch From Process to Process

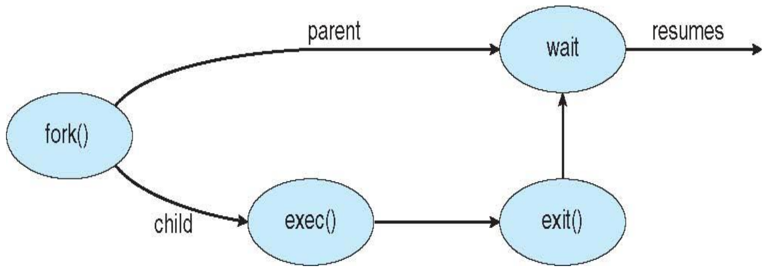


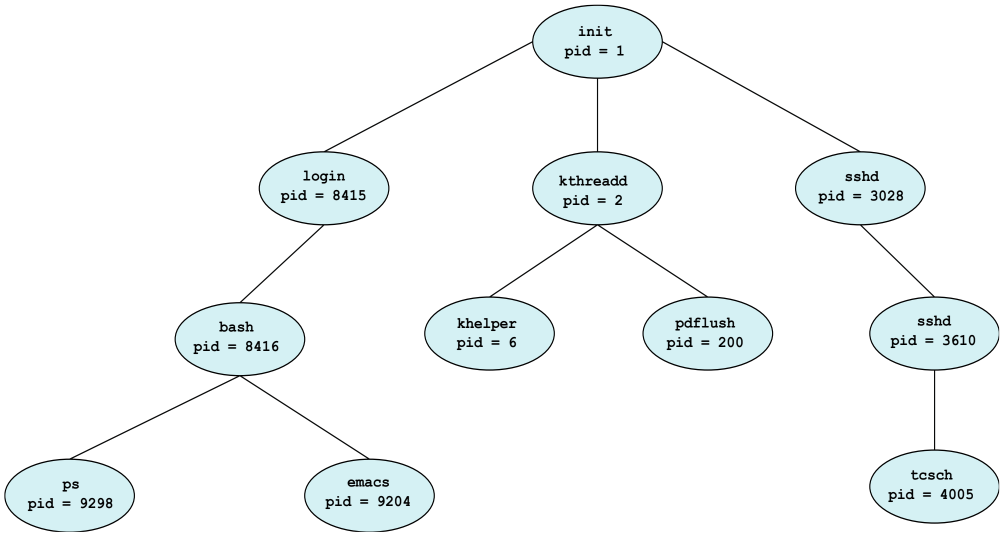
C structure
`task_struct`



Process Creation

- Processes are created and deleted dynamically
- Process which creates another process is called a *parent* process; the created process is called a *child* process.
- Result is a tree of processes
  - e.g. UNIX - processes have dependencies and form a hierarchy.
- Resources required when creating process
  - CPU time, files, memory, I/O devices etc.

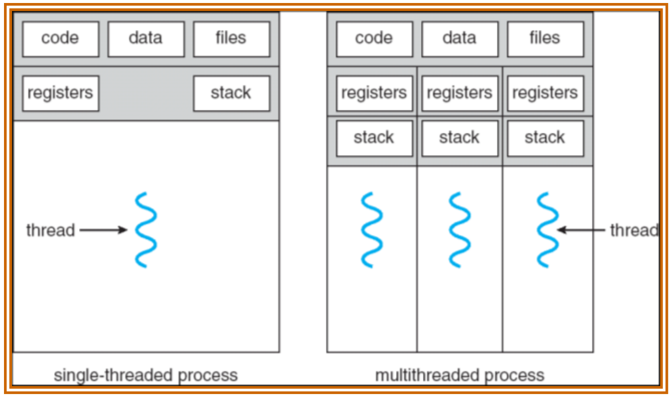




Colorado State University

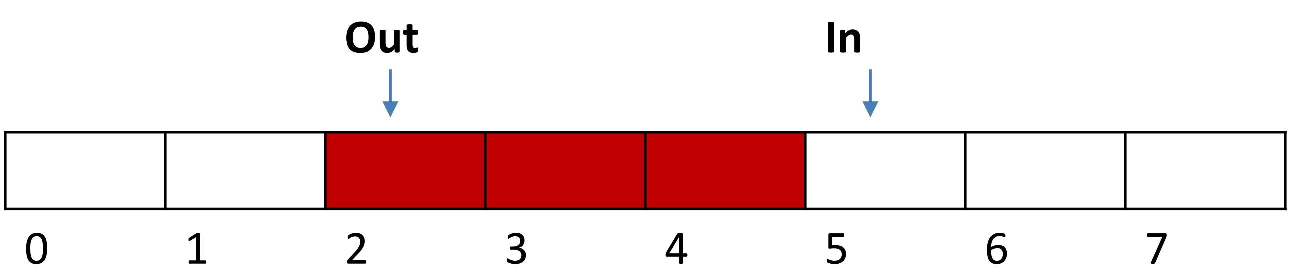
Threads

- A thread (or lightweight process)
    - basic unit of CPU utilization; it consists of:
        - program counter, register set and stack space
  - A thread shares the following with peer threads:
    - code section, data section and OS resources (open files, signals)
  - Collectively called a task.
- Thread support in modern systems
  - User threads vs. kernel threads, lightweight processes
  - 1-1, many-1 and many-many mapping
- Implicit Threading (e.g. OpenMP)
- Hardware support in newer processors



Producer-Consumer Problem

- Paradigm for cooperating processes;
  - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
  - Unbounded-buffer
  - Bounded-buffer
- Producer and Consumer must synchronize.

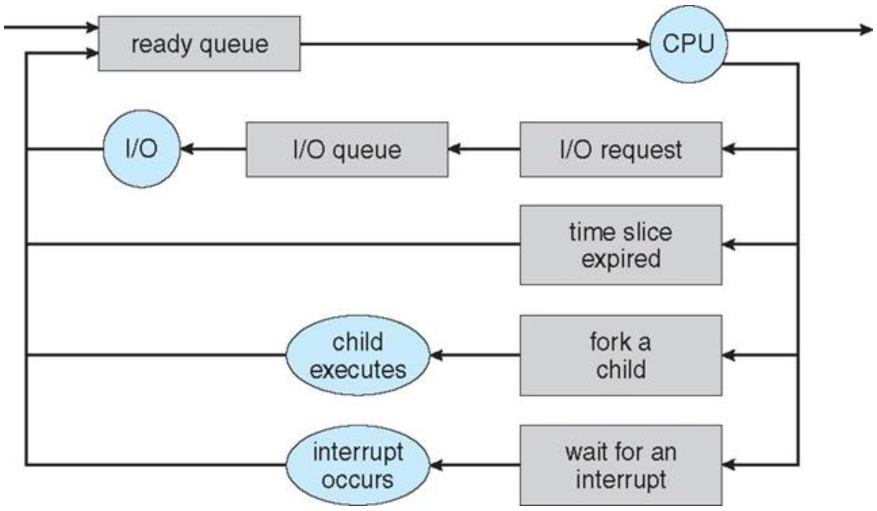


Colorado State University

Interprocess Communication (IPC)

- Mechanism for processes to communicate and synchronize their actions.
  - Via shared memory
  - Pipes
  - Via Messaging system - processes communicate without resorting to shared variables.

CPU Scheduling



- **CPU utilization** – keep the CPU as busy as possible: Maximize
- **Turnaround time** –time to execute a process from submission to completion: Minimize
- **Waiting time** – amount of time a process has been waiting in the ready queue: Minimize

Colorado State University

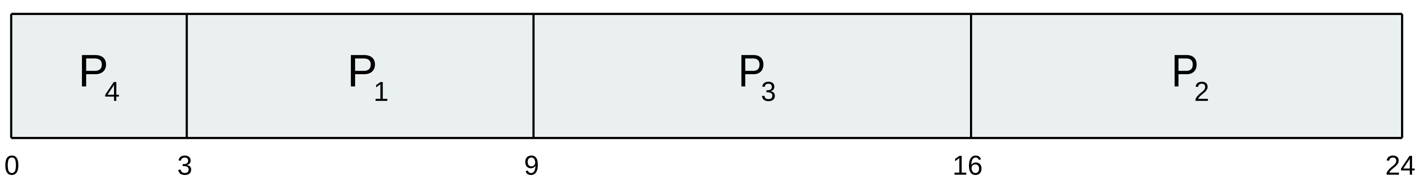
Scheduling Policies

- FCFS (First Come First Serve)
  - Process that requests the CPU *FIRST* is allocated the CPU *FIRST.*
- SJF (Shortest Job First)
  - Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Shortest-remaining-time-first (preemptive SJF)
  - A process preempted by an arriving process with shorter remaining time
- Priority
  - A priority value (integer) is associated with each process. CPU allocated to process with highest priority.
- Round Robin
  - Each process gets a small unit of CPU time
- MultiLevel
  - ready queue partitioned into separate queues
  - Variation: Multilevel Feedback queues: priority lower or raised based on history

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart



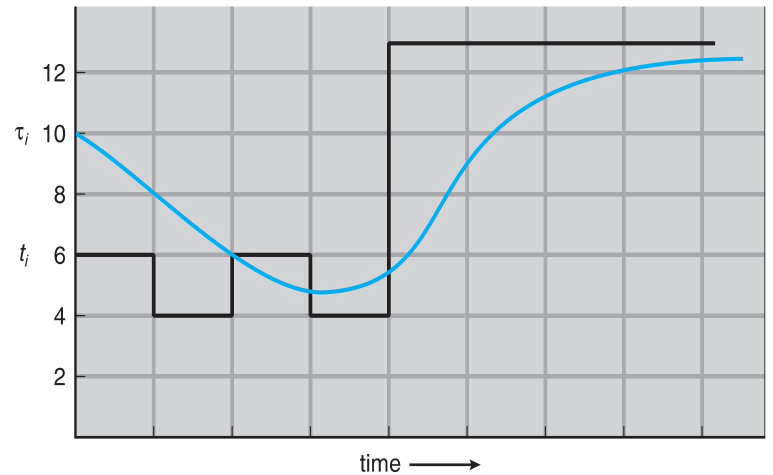
- Average waiting time for P_1,P_2,P_3,P_4 = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using *exponential averaging*

  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n$.

- Commonly, α set to ½



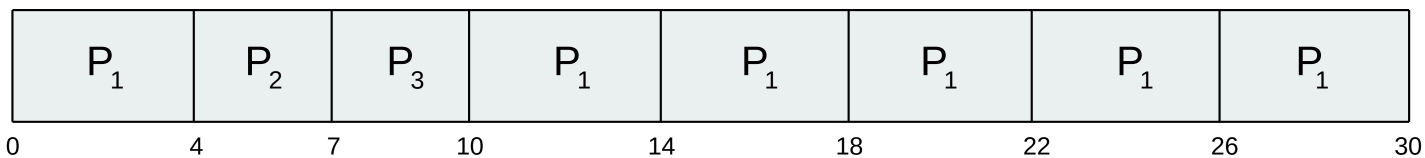
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Colorado State University

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrive a time 0 in order P1, P2, P3: The Gantt chart is:



- Waiting times: 10-4 =6, 4, 7, average 17/3 = 5.66 units
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch overhead < 1%

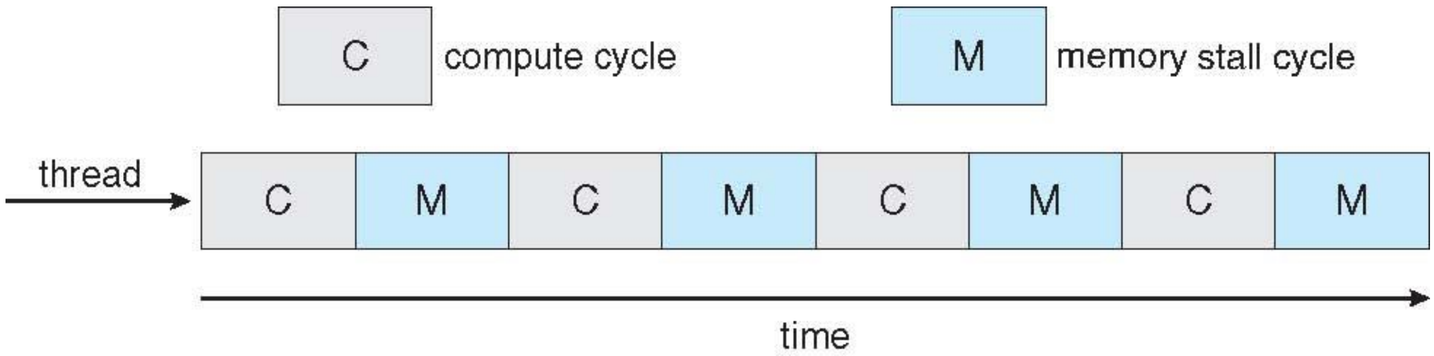
Response time: Arrival to beginning of execution
Turnaround time: Arrival to finish of execution

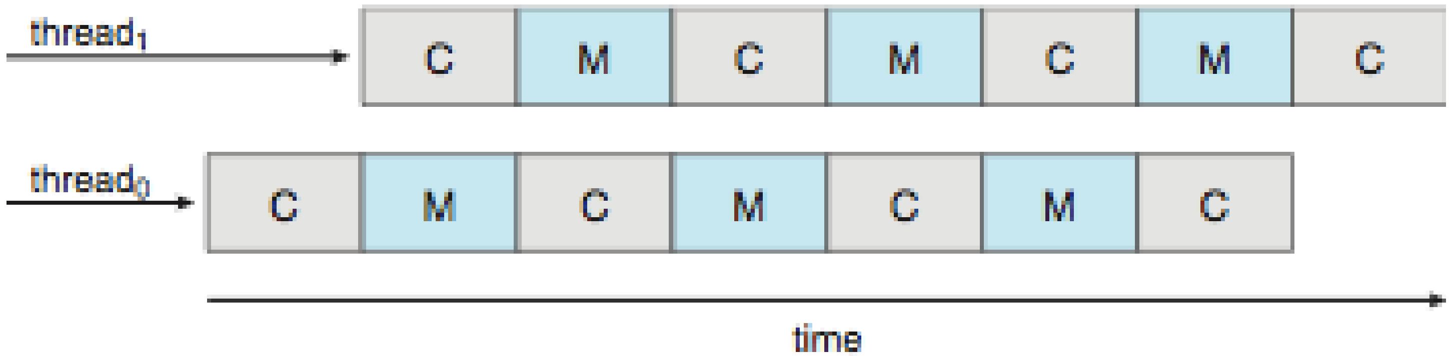
Colorado State University

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- **Assume Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
  - all processes in common ready queue, or
  - each has its own private queue of ready processes
    - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running because of info in cache
  - **soft affinity: try but no guarantee**
  - **hard affinity can specify processor sets**

Multithreaded Multicore System





This is temporal multithreading. Simultaneous multithreading allows threads to computer in parallel

Process Synchronization

- The Critical Section Problem
- Synchronization Hardware
- Semaphores

Consumer-producer problem

Producer

```
while (true) {
        /* produce an item*/
  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_SIZ
   counter--;
        /* consume the item in
        next consumed */
}
```

They run "concurrently" (or in parallel), and are subject to context switches at unpredictable times.

Colorado State University

18

Race Condition

`counter++` could be compiled as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be compiled as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

They run concurrently, and are subject to context switches at unpredictable times.

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Overwrites!

rado State University

The Critical Section Problem

- Requirements
  - Mutual Exclusion
  - Progress
  - Bounded Waiting
- Solution to the critical section problem

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

Peterson's Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);   /*Wait*/
              critical section
    flag[i] = false;
              remainder section
 } while (true);
```

Being nice!

- The variable `turn` indicates whose turn it is to enter the critical section
- `flag[i] = true` implies that process `P`$_i$ is ready!
- Proofs **for Mutual Exclusion, Progress, Bounded Wait**

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
      while (test_and_set(&lock)) ; /* do nothing */
            /* critical section */
       …. .
      lock = false;
            /* remainder section */
     …  ..
   } while (true);
```

Colorado State University

Bounded-waiting Mutual Exclusion with test_and_set

```
For process i:
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);
```

Shared Data structures initialized to FALSE

- `boolean waiting[n];`
- `boolean lock;`

The entry section for process i :

- First process to execute TestAndSet will find key == false ; ENTER critical section,
- EVERYONE else must wait

The exit section for process i:

Part I: Finding a suitable waiting process j and enable it to get through the while loop,

or if thre is no suitable process, make lock FALSE.

Mutex Locks

- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

•**Usage**

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

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

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()`
- Definition of the `wait() operation`

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

- Definition of the `signal() operation`

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

Colorado State University

Implementation with no Busy waiting (Counting Sema)

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

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

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