Tasks and Task Management (Chapter 12) – vocabulary:

- **absolute time** - real world time
- **relative time** - time referenced to some event
- **interval** - any slice of time characterized by start & end times
- **duration** - distance in time between start and end times
- **reactive systems** - tasks triggered by events
- **time-based systems** - tasks triggered by evolution of time
    - **absolute** - triggered at specific time
    - **relative** - task triggered before or after some reference or interval
- **periodic** - tasks with constant duration between initiation
- **aperiodic** - not periodic
- **execution time** - CPU time to complete a task
- **jitter** - variance of durations in intended periodic system
- **delay** - amount of time between evoking event and start of task
- **hard deadline** - if action doesn't occur by some time system is considered to have failed
- **hard real-time system** - has at least one task with hard-deadline, focus of design on hard-deadlines
- **soft real-time** - usually system must meet deadline "on average" e.g. to achieve an average throughout
- **firm real-time** - mix, of hard and soft deadlines
- **predictability** - how well we know the timing of tasks completing and starting
- **interarrival time** - time between evoking events, esp. for aperiodic systems. Typically need to know bounds and/or average intervals between events to know if system can meet demands

Task Scheduling

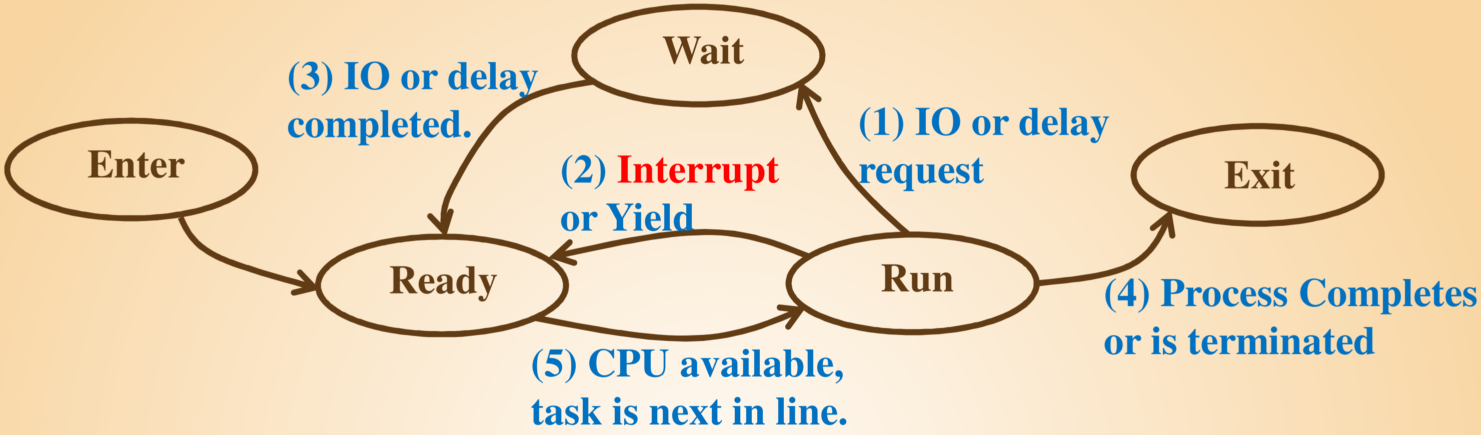
- **priority** - allows some task to be designated to run before others
- **schedulable** - in real-time context, refers to a task that can be scheduled (added to the queue) and will meet its timing constraints
- **deterministically schedulable** - in a system, means that it has been determined (can be guaranteed in-advance) that a given task will always be able to be scheduled and meet have its timing constraints met

CPU Utilization

- … is the % of time CPU is being used to complete tasks instead of being idle

- Typically about 40% to 90% (the higher the better!)

Scheduling Decisions


Wait
(3) IO or delay completed.
Enter
(2) Interrupt or Yield
(1) IO or delay request
Exit
Ready
Run
(4) Process Completes or is terminated
(5) CPU available, task is next in line.

- important state transition conditions are numbered in the diagram
- If (2)may be forced by OS interruption and not just by the task self-yielding the CPU, the system is preemptive, otherwise it is non-preemptive
- With non-preemptive OS tasks stop executing when they give up the CPU by completing (->exit), put themselves in a wait queue such as for an I/O request or desired delay (->wait), or yield the CPU themselves (->ready). Looking at the task code, these happen at well-define places.
- With preemptive OS tasks may be interrupted most places in the code

Scheduling Priority

- Non-preemptive and preemptive -> priority determines which task is selected when CPU becomes available
- Preemptive -> priority also determines if a running task should be suspended upon another task reaching the ready state
- These concepts apply to CPU access, not access to (I/O) resources.

Blocking (resource-based blocking) (what is it)

- When one task indirectly blocks another by holding onto a needed resource
- Example: Assume Task A and B with
  - priority(A) > priority(B)
  - both require resource R
  - entry order is B, A
- Example Sequence:
  1. B reserves access to R
  2. A enters and preempts B
  3. A runs until it needs access to R
  4. A must be suspended to resolve the problem to allow lower priority

Priority Inversion (indirect resource-based blocking through CPU priority blocking)

THIS ASSUMES NO RESOURCE PREEMPTION.

- Assume Tasks A , B, C such that:
  - priority(A)>priority(B)>priority(C)
  - Tasks A and C require resource R
  - Entry order is C, A, B
- Execution Scenario:
  - C begins to run and reserves R
  - A enters and preempts C
  - A runs, requests R
  - A is suspended to allow C to run and eventually free R
  - B enters and preempts C before C releases R
  - B runs until completion
  - C runs releasing R
  - A is resumed, uses R and completes
  - C is allowed to complete

- The **priority inversion** happens here. The higher priority TASK, A, waits. The scheduler took into account and handled CPU priority and resource blocking separately.
- C could have been given a higher priority to the CPU than B by the virtue of it blocking A to prevent this.

More Vocab

- **turnaround time** - the interval from the submission of a task to task completion
- **throughput** - # of process completed per unit of time
- **response time** - time delay from submission to the first action of task
- **waiting time** - the time spent in "waiting" queues
- Example Queues that could be implemented:
  - **entry queue** - has the tasks submitted but not yet ready to run
  - **ready queue** - has tasks ready to run when CPU is available
  - **I/O queue** - has tasks waiting on I/0
  - **device queue** - has tasks waiting for access to a particular I/O device

Scheduling Algorithms

- basic common types:
  - asynchronous interrupt driven
  - polled
  - polled with a timing event

Asynchronous Interrupt Driven

- main program is a trivial infinite loop
- tasks are initiated by interrupts
- the CPU can sit in a low-power sleep mode waiting for events (see microcontroller documentation for availability of sleep features and details of usage)

```
CODE OUTLINE:
global variable declaration
ISR setup
function prototypes
void main(void){
  local variable declarations
  while (1); //task loop
}
ISR definitions
Function definitions
```

- It is difficult to analyze (debug, evaluate, simulate, etc...) since it is based on external events.

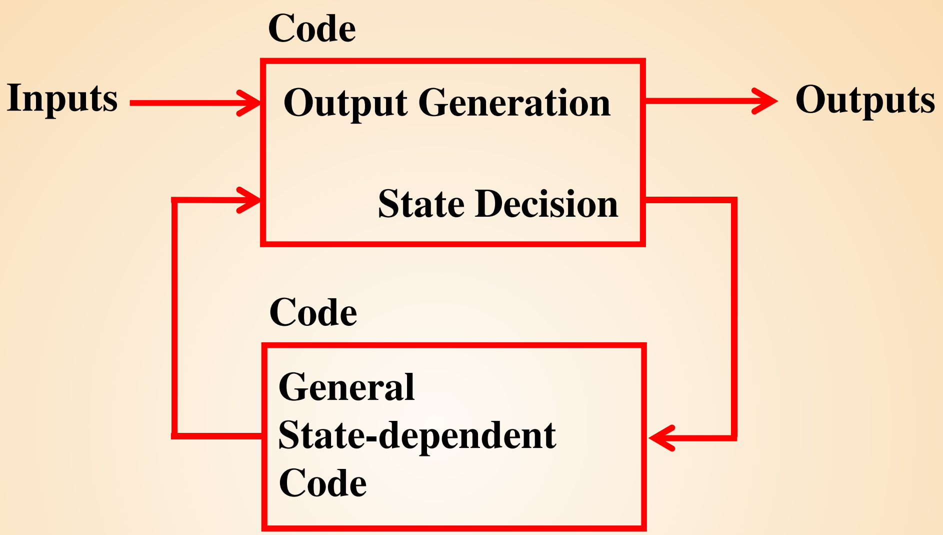
Polled and Polled with a Timing Element

- Software polls signals and dispatches tasks
- CPU is utilized more since it must check signals
- can add pauses to "align" events to steps of time (or sleep events to save power)

```
global variable declaration

function prototypes

void main(void){
  local variable declarations
  while (1){   //task loop
    /* can add pauses/waiting here if desired */
    test state of each poll signal
    an if, then,or switch construct to initiate tasks
    }
  }

function definitions
```

very predictable timing
(easy to analyze)

Approaches to Task Code Design: State-Based Approach



Code

Inputs

Output Generation

State Decision

Outputs

Code

General
State-dependent
Code

- For a state-bases approach, a loop with two types/sections of code are written.
  - Some code is dedicated to processing inputs, setting output states, and deciding the next state.
  - Other code performs operations based on the current state
- This is mainly event-driven. This model not always well suited for software especially if the number of states is large. For a few states it is an O.K. way to organize code.

Approaches to Task Code Design: Synchronous Interrupt Event Driven

- Like async.-event-driven, but event to trigger context switch decision is based on a regular timer.
- Timer triggers ISR to handle context switching and scheduling
- Can implement time-sharing systems and implement preemptive scheduling (using ISR allows this)

Approaches to Task Code Design: Combined Interrupt Driven

- Allow context switch on regular timer events AND asynchronous input events.
- ---but make asynchronous ISRs short

Approaches to Task Code Design: Forground-Background

- Mix of interrupt-driven and non-interrupt-driven tasks.
- Interrupts signal foreground tasks while background tasks run

Time-shared Systems

- Divvy CPU time to multiple tasks
  - first come -first serve (non preemptive)
    - task taken from front of queue and runs to completion
    - not real time
  - shortest job first
    - each task has an associated expected CPU time before completion
    - non-preemptive: tasks finish and shortest next one is selected
    - Preemptive: if current task has longer to complete than another task it may be suspended
  - round robin
    - blocks of time called time quantum's or slices are allocated in a rotating pattern to tasks
    - task that don't finish in give slice are suspended and moved to back of queue

Priority Scheduling

- allow scheduling dependent on priority of task
- "shortest job first" is one example of priority scheduling
- With priority scheduling there can be issues:
  - (indefinite) blocking, priority inversion, starvation(some tasks never get a chance to run in a reasonable time, causing severe delays)
- Tasks can be prioritized based on
  - execution time
  - period allowed or deadlines
- priority can be assigned upon submission or changed as conditions evolve: Details can be found in section 12.3.8 of the book, we'll put it off for later as time allows

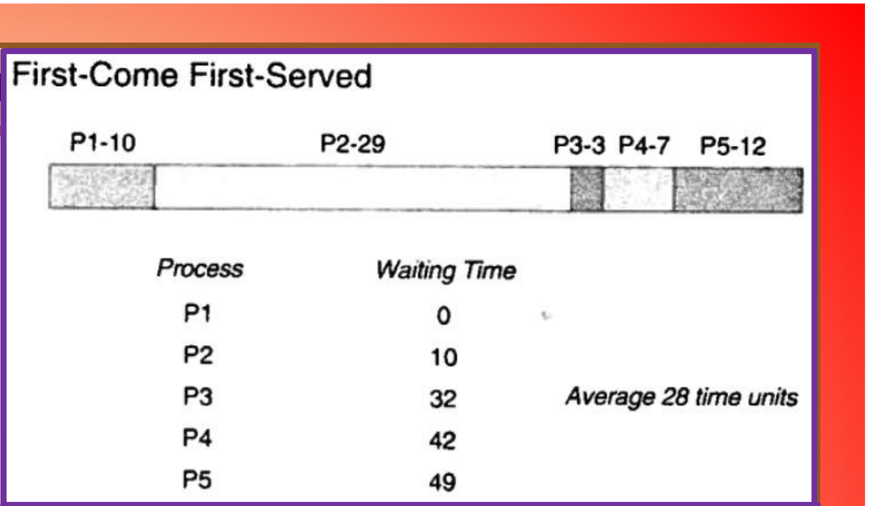
Real-time Scheduling

- hard-real time
  - scheduler accepts tasks it knows it can complete in a given time
    - requires advance information about tasks such as execution time and I/O resources required so that reservations for resources can be made
    - works with I/O devices with well-defined timing
- soft-real time
  - focuses on prioritization and low-latency dispatch, usually requiring preemption

Algorithm Evaluation

- Common to look at metrics such as average wait time or average process-completion throughput
- Take 5 processes, P1 – P5, queued at the same time:

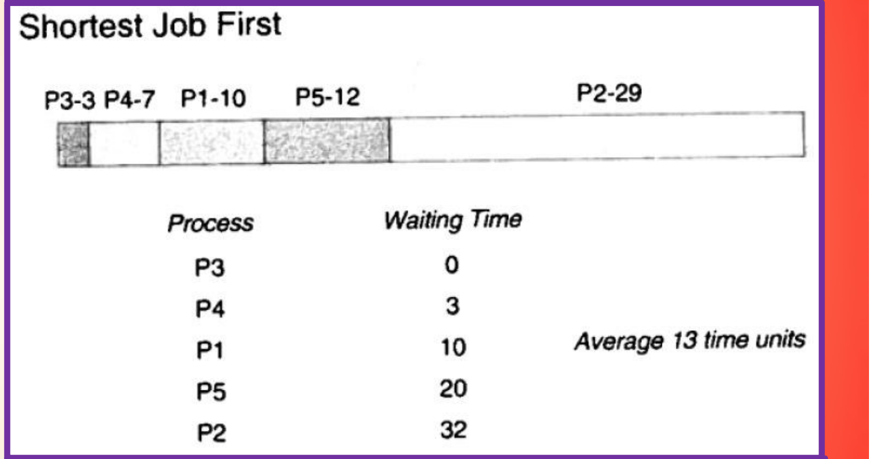
Process	CPU Time Required
P1	10
P2	29
P3	3
P4	7
P5	12



First-Come First-Served

Process	Waiting Time
P1	0
P2	10
P3	32
P4	42
P5	49

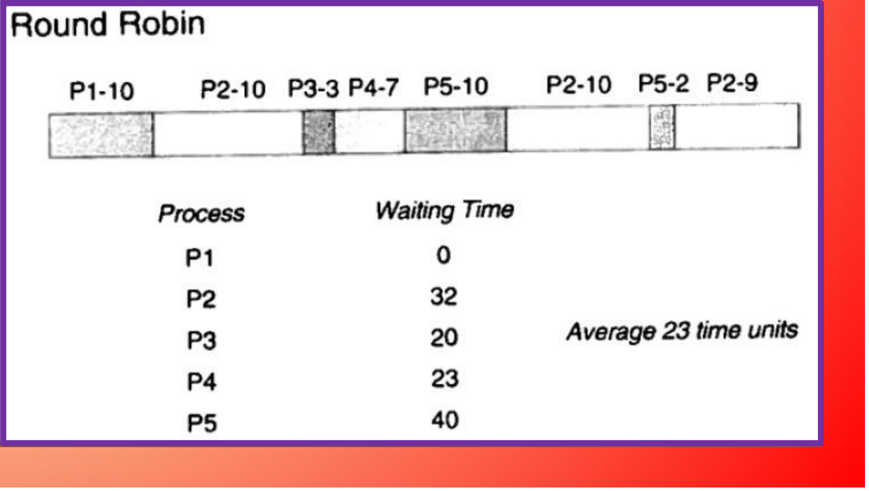
Average 28 time units



Shortest Job First

Process	Waiting Time
P3	0
P4	3
P1	10
P5	20
P2	32

Average 13 time units



Round Robin

Process	Waiting Time
P1	0
P2	32
P3	20
P4	23
P5	40

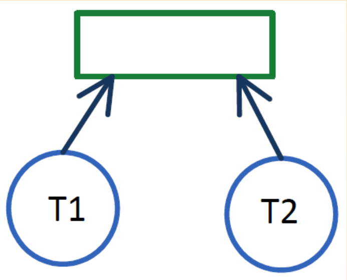
Average 23 time units

Inter-task Communication (Textbook, section 12.6)

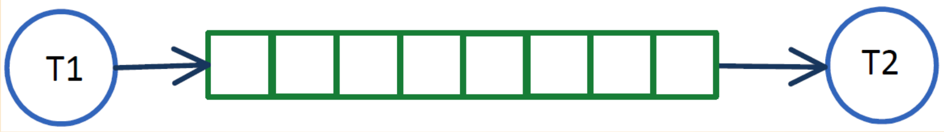
- considerations
  - →what data is to be shared
  - →where is it stored
  - →how to coordinate the usage between two processes

- Examples of data to share:
  - ▪ a number,
  - ▪ an array,
  - ▪ status about a resource,
  - ▪ synchronization signals to facilitate coordination

Shared Variables



- global variables
  - need to coordinate access
  - efficient for space and speed, important in embedded systems

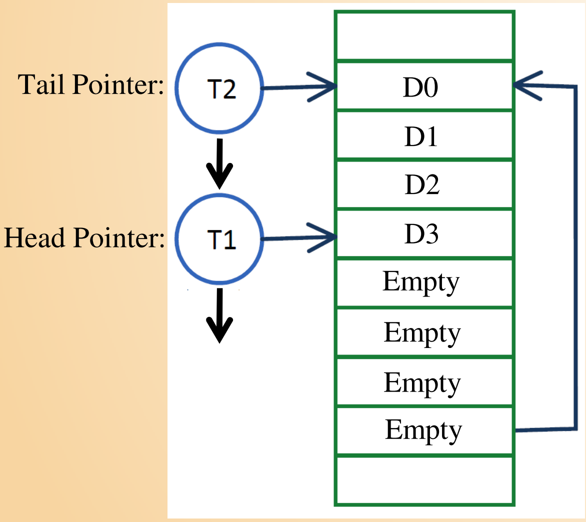


- shared buffers
  - can be a stack, FIFO queue, or other buffer type
  - **Producers** place data in the buffer and **consumers** access and remove the data
    - ➢ need methods to coordinate, like functions IsFull() and IsEmpty()
  - **Overrun** is when a producer fills a buffer faster than a consumer can remove it.
  - **Consumer**(s) remove it to the point where the buffer fills and can't accept incoming data without losing data
  - **Underrun** is when a consumer is left without any data...can be an indication of inefficient design (such as buffer that is not really required) , but it isn't a show-stopper in and of itself

Ring or Circular Buffer

- A ring buffer is created using a block of memory and two pointers, the head pointer and the tail pointer. Data is written at the head and the head pointer is advanced.
- Data is read from the tail and the tail pointer is advanced.
- When the pointers reach the end of the physical array, the will "advance" to the start of the physical array.
- When the head pointer catches up with the tail, the buffer is full.
- When the tail pointer catches up with the head, the buffer is empty

Physical

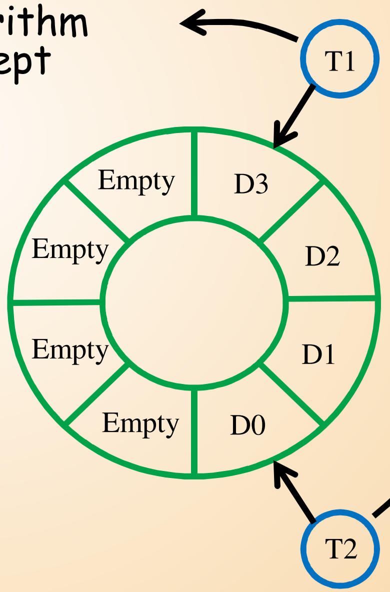


Tail Pointer: T2

Head Pointer: T1

D0
D1
D2
D3
Empty
Empty
Empty
Empty

Algorithm Concept



T1

Head Pointer:

Producer causes values to be written at the head followed by the head advancing…

Empty
D3
Empty
D2
Empty
D1
Empty
D0

Consumer causes values to be read at the head tail, followed by the tail pointer advancing…

Tail Pointer:

T2

- need to manage underflow and overflow
- should (must) provide IsEmpty() and IsFull() so that processes can query the buffer and respond accordingly

Mailbox

- Mailbox constructs are provided by a full-featured OS. The basic idea is that each process has an incoming message queue that it can access, with messages posted by other processes.
- One process can signal that it is waiting for a message (data) in the mailbox and, if nothing is available in the mailbox, it can be suspended until a message shows up
- A mailbox can have a message (data) posted to it by another process
- Typical Code Interface:
  - pend (mailbox, data)
  - post (mailbox, data)

Message-Based Approach

- Mailbox concept can be expanded, for instance, to support buffers or communication among devices on a board or larger network. General message boxes can be read and posted to by multiple processes.

- Direct Communication - sender identifiers receiver and receiver identifies sender, or at lease sender identifies receiver
  - send (T1, message) //send message to task T1
  - retrieve (T0, message) //receive message from T0

- Indirect communication - senders and receivers identify a mailbox
  - send (M0, message)
  - receive(M0, message)

Message Buffering

- one consideration is if link guarantees message order
- regarding capacity there are 3 options:
  - zero-capacity link
    - sender waits on receiver (so order is certain)
    - execution timing tied together
  - bounded-capacity link
    - sender may burst data, but can only send if buffer not full
    - receiver must match sender rate "on average"
  - unbounded
    - no waiting, requires infinite storage or be sure that receiver removes data quickly enough

Access to Shard Resources

- example using a shared buffer B0

- Producer T0

```
int in = 0;
while(1){
  while(count == MAXSIZE);
  B0[in] = nextT0;
  in = (in+1) % MAXSIZE;
  count++;
}
```

count = count + 1;

- Consumer T1

```
int out = 0;
while(1){
  while(count == 0);
  nextT1 = B0[out];
  out = (out+1) % MAXSIZE;
  count--;
}
```

count = count - 1;

- count read-and-modify operations should be **atomic**, executing from start to finished without interruption

- what happens if count = count + 1 is interrupted between the read and the write so that T1 runs?

  count = count + 1 → is a critical section of code

Atomic Access using Flags

- once the critical section of code has been identified, it can be protected with statements that
  1. check and flag use of resource, entry section
  2. flag the release of the resource, exit section

- Producer T0

```
int in = 0;
while(1){
  while(count == MAXSIZE);
  B0[in] = nextT0;
  in = (in+1) % MAXSIZE;
  await(!T1Flag){
    T0Flag = true;}
  count++;
  T0Flag = false;
}
```

- Consumer T1

```
int out = 0;
while(1){
  while(count == 0);
  nextT1 = B0[out];
  out = (out+1) % MAXSIZE;
  await(!T0Flag){
    T1Flag = true;}
  count--;
  T1Flag = false;
}
```

- This assumes that while awaiting a condition to become true, other processes can run that will allow the condition to become true, otherwise there is a deadlock

Solutions to Critical Section Problem

- ...must satisfy:
  - → must ensure mutual exclusionin the critical region, only one task at a time
    - can be in the critical section of code
  - → prevent deadlock if two tasks are trying to enter their critical section at the same time,
    - at least one task should be able to enter its critical section
  - → ensure ability to progress into critical section:
    - if no other Task is in its Critical section, only tasks in the exit section should have influence to affect entry
  - → bounded waiting - should limit the number of times a low priority task can be blocked by higher priority processes

Tokens and Token Passing

- Before accessing a resource a process must "acquire a token"
- token is represented by some flag or variable
- token is passed directly from task to task; initiated by task, not OS

- Problems / Challenges:
  - some task holds token forever
  - task with token crashes (never frees/passes token)
  - token is lost (such as in communication problem)
  - process with token exits without passing token
  - managing queue of process to receive token

- Solution
  - system-level token manager that can recall and issue new tokens after time expires

- Overall, tokens require a lot of communication and overhead

Interrupts

- processes can disable interrupts during critical sections and prevent preempting
- a process that hangs in critical section can cause problems... like token passing problem
- a solution is to disable interrupts below some priority level, allowing a time-out interrupt to recover the system
- not useful for processes running on different processors

Semaphores

- concept:
  - use a variable (s) to signal the locking of a resource
  - simplest version involves a binary variable accessed through two ATOMIC Functions

s is initialized to false in the system

```
wait(s){
  while(s);
  s = true;
}
```

test
& set

```
signal(s){
  s = false;
}
```

clear

Every task uses wait & signal around critical sections

- Task T0

```
{
  …
  wait(s)
  critical section
  signal(s)
  …
}
```

- Task T1

```
{
  …
  wait(s)
  critical section
  signal(s)
  …
}
```

First task
to reach
wait can
block the
other

MUTEX

- the use of a semaphore in the previous example is known as using a **mutex** (short for mutual exclusion) signal

Process Synchronization

- Aside from managing access to resources, semaphores can also be used to synchronize processes

- Task T0

```
{
   …
   get input from user
      and put in a buffer
   signal(sync)
…
}
```

- Task T1

```
{
   …
   wait(sync)
   process user data
      in the buffer
   …
}
```

Extending Semaphores

- →rather than have a of process in a "spin lock" where it uses CPU cycles to check the synchronization signal until the instant the look is gone, you could sacrifice response time and free the CPU by suspending the process and putting it in an (OS-managed) queue to be woken when another process calls the signal function
- →semaphores can take on more than a binary value; this is useful for managing a pool of identical recourses and multiple processes accessing them
- Example of both… counting semaphore with blocking

```
wait(s){
  s=s+1;
  if (s>POOLSIZE){
    add process to waiting queue
    block; //suspends process
  }
}
```

```
signal(s){
  s=s-1;
  if (s>POOLSIZE){
    remove a process p from waiting queue
    wakeup(p); //resumes process
  }
}
```

Provided
by OS

Monitors (for reference, not emphasized for this course)

- Semaphores represent a fundamental, low-level mechanism for resource locking and process synchronization.

- In general, tasks can use semaphores or you can write a "library" to access a particular resource that itself utilizes semaphores. For instance, imagine if a write command had a semaphore built into it to implement write blocking to prevent more than one process from writing at the same time.

- Section 12.9 of the text discusses Monitors, another abstraction whereby access to a resource such as a buffer is managed by a abstract data structure with a public interface (functions) to access the resource and allow process sleeping and resuming based on conditions.

Starvation and Deadlocks

- Be aware of starvation and deadlocks when using semaphores.

- **Starvation** is when a process never has an opportunity to run and complete (or has to wait a very long time). This can be caused by LIFO queues.
- **Deadlocks** occur when a series of locks can be set up that prevent tasks from ever completing.

- Necessary conditions:
  - Mutual Exclusion-once a process has been allocated a resource, it has exclusive access to it
  - No Preemption-resources can not be preempted
  - Hold and Wait-some processes are holding one resource and waiting on another resource
  - Circular Wait-a set of processes exists in a state such that each process is waiting on a resource held by another in that set

- Example 1:
  - To make a peanut butter sandwich, bread, PB, and knife are needed. Assume two uncooperative children Suzy and Jonny run to the kitchen: Jonny grabs the PB and Suzy grabs the knife.

- Example 2:
  - variables representing back accounts actA and actB
  - T0 wants to initiate a transfer, so it reserves access to actA and is suspended
  - T1 wants to initiate a transfer from actB to actA, so reserves access to actB and waits
  - for access to actA then suspends
  - T0 resumes and waits for access to actB

Handling Deadlocks

- Two basic ideas:
  - **Deadlock prevention** - OS can monitor a deadlock condition or conditions and prevent at least one of them (prevent little Jonny from grabbing the PB)
  - **Deadlock avoidance** - the OS can be given information about resources required by the task and delay tasks that may cause deadlocks. A hard real-time system requires this information. (tell Suzy to keep watching TV)

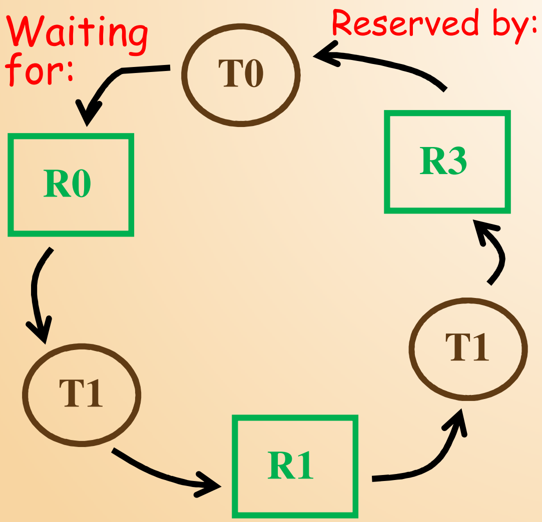
- **Deadlock Prevention** -- Causes and Mitigation
  - **Mutual Exclusion** - avoid locking access where not required. Example: though a buffer may not be written by more than one process, read access may be shareable
  - **Hold and Wait** - don't allow processes to wait for resources if it already has some reserved. This requires a process to wait for and reserve all required resources to be available at once. Also, if a process requires an additional resource, it must first free all of its resources without condition. This may lead to low resource utilization and starvation of low-priority tasks.
  - **No preemption** - allow preemption.
    - Any process that hold resources and requests another that is not available must allow its resources to be freed and if this happens a list of required resources is created and the process will resume when all of them are available
    - ...unless that resource being requested is held by another process that is itself waiting on other resources, in that case force that other waiting process to give up the resources
  - **Circular Wait** - assign an order to all resources and enforce that no resources may wait on a higher-numbered resource while holding a lower numbered resource while order for requests. This requires requesting resources in-order and/or free resources lower-number resources when requesting a higher-numbered resource.
    - In bank example, imaging if T1 waited for actA before asking for actB

- **Deadlock Avoidance**
  - Details are given in section 13.7 of the textbook and in OS course.
  - The basic idea is that process must inform the OS of what resources it potentially needs to execute a task. The OS then makes decisions about which processes may safely be allowed to continue and what order would be most efficient.

Deadlock Detection and Recovery

- If no prevention or avoidance is provided, deadlock detection and deadlock recovery must exist.
- OS must maintain a model of allocated resources and pending waits.

- Wait-for Graph
  - Arrow from resource to task indicates resource reserved by task
  - Arrow from task to resource indicates task waiting for resource



Once a deadlock is detected, the processes can be collectively or selectively terminated or have their resources preempted. This is not simple since preempting resources or terminating processes can leave resources in dangerous sates. Mechanisms for allowing safe preemption or repair should be considered. The assumption is that terminating a process only requires it to restart its work resulting in wasted time (which can drive selection of what process should be terminated to waste as little time as possible). One should consider allowing for recovery points (checkpoints) in long tasks that may be terminated...like autosave in a word processor. Recovery should not cause the deadlock to be repeated and should not starve processes.