xianhao@comp.nus.edu.sg CG2271 AY18/19 S1

Chapter 5: Real-Time Scheduling

Worst-Case Execution Time (WCET), C: Maximum amount of time required to complete the execution of any instance of task without any interference from other activities
Deadline, D: Maximum amount of time allowed between a task being released and its completion time.
Worst-Case Response Time (WCRT), R: Maximum elapsed time between the release of a task instance and its completion
Pre-emptive Priority-Driven Scheduling: Lower priority task that is current executing gets context switched out once a higher priority task becomes ready
Hyper-Period: Least Common Multiple of the task periods
Utilization: Fraction of processor time spent in the execution of the task set, $U = \sum_i \frac{c_i}{p_i}$
Critical Instant: The instant at which the release of the task will produce the largest response time
Critical Instance: The task instance released at the critical instant

Basic Real-Time Task Model
1.Tasks are periodic with known periods and:
  a.Periods do not change with time
  b.Task is an infinite sequence of instance
  c.One instance is released at the beginning of each period
2.Tasks are completely independent of each other
3.System overhead for scheduling and context switch is negligible
4.All tasks have hard deadline
5.Assume D = P, to meet deadline, C ≤ R, R ≤ D

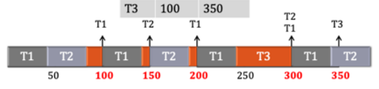
Rate-Monotonic Scheduling for Fixed Priority
1.Priority is inversely proportional to task period
2.Scheduler repeats itself every hyper-period, i.e. task set is schedulable if no deadline is missed in one hyper-period
3.If a task set is schedulable by any arbitrary but fixed-priority assignment, then it is schedulable by RMS
4.U ≤ 1 is a necessary but not sufficient condition for any task set to have a feasible schedule i.e. even if U ≤ 1, task set may not have a feasible schedule
5.Utilization Bound $U \leq n(2^{\frac{1}{n}} - 1)$ is a sufficient but not necessary condition for any task set to have a feasible schedule i.e. though condition is not satisfied, task set may have feasible schedule
6.Deadline must equal to period to use utilization bound analysis

Rate-Monotonic Analysis
A critical instant for any task occurs whenever it is released simultaneously with the release of all higher-priority task
Necessary and Sufficient Condition

$$w_i(t) = \sum_{k=1}^{i} c_k + \left\lceil \frac{t}{p_k} \right\rceil$$

where $T_1 \ldots T_{i-1}$ are tasks with higher priority than T_i, inequality for $w_i(t) \leq t$ holds for time instant t where $t = kp_j, j = 1, \ldots, i$ and $k = 1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor$

Task	WCET	Period
T1	40	100
T2	40	150
T3	100	350



- **i = 3: Higher priority tasks T_1 and T_2**
- **Time instances to consider: t = kp_j, j = 1, ..., i, k = 1, ... $\left\lfloor \frac{p_i}{p_j} \right\rfloor$**
  - j = 1; $\left\lfloor \frac{p_i}{p_j} \right\rfloor = \left\lfloor \frac{350}{100} \right\rfloor = 3$ Thus k = 1, 2, 3 and t = 100, 200, 300
  - j = 2; $\left\lfloor \frac{p_i}{p_j} \right\rfloor = \left\lfloor \frac{350}{150} \right\rfloor = 2$ Thus k = 1, 2 and t = 150, 300
  - j = 3; $\left\lfloor \frac{p_i}{p_j} \right\rfloor = \left\lfloor \frac{350}{350} \right\rfloor = 1$ Thus k = 1 and t = 350
- **Consider t = 300**

$w_3(300) = 40 \times \left\lceil \frac{300}{100} \right\rceil + 40 \times \left\lceil \frac{300}{150} \right\rceil + 100 \times \left\lceil \frac{300}{350} \right\rceil$
$= 40 \times 3 + 40 \times 2 + 100 = 300$
$w_3(300) <= 300$

$$w_i(t) = \sum_{k=1}^{i} c_k \times \left\lceil \frac{t}{p_k} \right\rceil$$

T3 is fine→ check T2, T1…

Earliest Deadline First for Dynamic Priority
1.Dynamic priority scheduling scheme where task with the earliest headline has the highest priority
2.Two Key Properties:
  a.Priority of task depends on the current deadline of the active task instance
  b.Scheduling decision must be considered only when any new task instance is released
3.Optimal scheduling policy:
  a.EDF can always produce feasible schedule if $U \leq 1$
  b.Scheduling with dynamic priority is feasible iff $U \leq 1$
4.Check entire hyper-period for feasibility using EDF

Cyclic Executive for Fixed Tasks Sets (a.k.a hardcoding)
1.Also known as timeline scheduling or cyclic scheduling
2.Consists of periodic, independent tasks with deadline = period
3.Creates a completely offline static schedule for the hyper-period that meets all the deadlines
4.Put tasks in hypertask avoid scheduling/context switch overhead

Cyclic Executive Implementation
1.Repeats the same schedule for each major cycle
  a.Major cycle: LCM of period/deadline (hyper-period)
2.Within each major cycle, minor cycles are synchronization points
  a.Minor cycle: GCD of period/deadlines
3.Execution switches from one minor cycle to another at periodic timer interrupt
4.Tasks within a minor cycle are activated in sequence

	Positive	Negative
RMS	Simpler implementation in commercial kernels that do not explicitly support constraints (periods/deadlines)	Scheduling and context switching overhead
EDF	Full processor utilization (efficient exploitation of computational resources) Better aperiodic activities responsiveness	
CE	Minimizes pre-emption if schedule is constructed carefully No need for actual tasks; only procedural call (all procedures share same address space; task execute sequentially so no need to protect shared data as there is no concurrent access) Implementation is straightforward and effective	

Chapter 6: Synchronization

Race condition due to concurrent access to shared data may result in data inconsistency
Critical section: Code segment in a process accessing shared data
Mutex: Special case of a semaphore where only one process can access shared resources at a time

Synchronization Hardware
Modern machine provides special atomic (non-interruptible) hardware instructions using `TestAndSet` which test and modify the content of a word atomically

```
bool TestAndSet(bool *target) {
    bool rv = *target;
    *target = true;
    return rv;
}
```

```
while (1) {
    while (TestAndSet(&lock));
    // Critical Section
    lock = false;
    // Remainder section
}
```

Semaphore
1.Defines semaphore as a record

```
typedef struct {
    int value;
    //L is a queue that stores waiting processes in
the form of PCB
    struct process *L;
} semaphore;
```

Initialized queue to empty; set value to user-defined # (resources)
2.Only accessed by 2 atomic operations:
  a.`P(S)` – before accessing shared object, check whether the value of semaphore `s` is greater than 0; if yes decrement the value and return without blocking; else put the process into the queue associated with s and blocks the process
  b.`V(s)` – after accessing shared object, check whether the queue associated with semaphore `s` is empty; if yes, increment the value of `s`, else, pick a blocked process in the queue and move it into ready state for execution
3.Consists of 2 simple operations:
  a.`block()` – dequeue current process from ready queue and enqueue current process to semaphore queue
  b.`wakeup(P)` – dequeue a process `P` from semaphore queue and enqueue process `P` to ready queue

```
//P(S)
if (S.value > 0) {
    S.value++
} else {
    block();
}
```

```
//V(S)
if (S.L.empty()) {
    S.value++;
} else {
    wakeup(P);
}
```

Priority Inversion
1.High priority task is delayed by lower priority tasks because they have control over resources which high priority task requires
2.Happens when 3 or more processes are involved: low priority process has the lock, high priority process needs the lock, mid priority process doesn't need the lock
3.Mid priority process can indefinitely delay high priority process by not allowing low priority process to complete critical section to release the lock

Priority Inheritance
1.Priority of low priority process is temporarily raised to the priority of the high priority process when low priority process is running and holding a lock and if a high priority process attempts to acquire the lock
2.Priority of the low priority process is set back to low priority after releasing the lock
3.Ensures that mid priority process that does not require the lock cannot pre-empt low priority process

Chapter 7: Inter-Task Communications

Message Passing:
1.Less efficient but more scalable than shared memory
2.Processes communicate/synchronize without shared variables
3.Provides two operations: a. `send(message)` and b. `receive(message)` where message size fixed or variable
4.Messages are not shared among processes; ownership of message is passed from sender to receiver when message is transferred (modifications done are local and does not affect either party) so mutual exclusion is not a concern

Naming Schemes:

Type	Sender	Receiver
Direct	Name the receiver by its identifier	N.A.
Indirect	Names the same intermediate entity called mailbox/channel/message queue	
Symmetric	Names the receiver/ destination mailbox	Names the sender/source mailbox
Asymmetric		N.A.

Receiver Synchronization Model:
Blocking: Receiver wait for a message if it is not already available
Non-Blocking: Checks whether a message is available; if yes, retrieves the message, else move on without waiting for message

Synchronous Message Passing:
1.Sender invoking `send()` is blocked till receiver `receive()`
2.Receiver invoking `receive()` is blocked till sender `send()`
3.No intermediate buffering is required since sender is forced to wait till receiver is ready i.e. message can be kept by the sender till receiver is ready and transfer message from sender's to receiver's address space directly

Asynchronous Message Passing:
1.Sender is never blocked even if receiver has not `receive()`
2.System buffers the message up to a maximum capacity
3.Receiver invoking `receive()` is blocked till buffer has message
4.`receive()` performed by the receiver completes immediately
5.Asynchronous communication is generally a good choice but:
  a.Too much freedom for programmer; program is complex
  b.Finite buffer size means system is not truly asynchronous (sender waits when buffer is full or return error)

FreeRTOS Message Queues:
1.Employs symmetric, indirect naming scheme with asynchronous message passing and finite buffers
2.Consists of 2 task waiting lists:
  a.Receiving waiting list consists of tasks that wait on the queue when it is empty
  b.Sending waiting list consists of tasks that wait on the queue when it is full
3.If more than one tasks were blocked on the same queue, the task with highest priority is unblocked first

FreeRTOS Semaphores:
Relies on message queues for blocked processes

Chapter 8: Deadlocks

Deadlock: Set of blocked processes each holding a resource and waiting to acquire a resource held by another process

Resource Allocation Graph:
1.Vertices partitioned into two types:
  a.P = {P_1, P_2, …, P_n} consisting of all processes
  b.R = {R_1, R_2, …, R_n} consisting of all resources
2.Request edge: directed edge $P_i \rightarrow R_j$
3.Assignment edge: directed edge $R_j \rightarrow P_i$
4.If graph contains no cycles ⇒ no deadlock
5.If graph contains a cycle ⇒ deadlock if only one instance per resource type; possibility of deadlock if several instances per resource type

Deadlock Conditions:
1.Mutual exclusion: only one process can use a resource at a time
2.Hold & wait: process holding at least one resource is waiting to acquire additional resources held by other processes
3.No pre-emption: resource can be released only voluntarily by the process holding it after the process has completed its task
4.Circular wait: exist a set {P_0, P_1, …, P_N} of waiting processes such that P_0 is waiting for P_1, and P_1 is waiting for P_2 etc

Deadlock Prevention:
1.Mutual exclusion – not required for sharable resources; must hold for non-sharable resources
  a.Sharable resources: code section, read-only data
  b.Non-shareable resources: printer, data areas to be written
2.Hold & wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
3.No pre-emption – if a process that is holding some resources requests another resource that cannot be immediately allocated, all resources being held are released
4.Circular wait – impose a total ordering of all resource types (limit maximum number of processes running at any point in time)

Deadlock Avoidance:
1.Algorithm dynamically examines resource-allocation state to ensure that system never goes into unsafe state
2.System is in safe state if there exists a safe sequence of all processes
3.No deadlock if system is in safe state; else possibility of deadlock

Banker's Algorithm:
Let `n` = number of processes, `m` = number of resources type:
1.`available[m]`; if `available[j] = k`, `k` instances of R_j available
2.`max[n, m]`; if `max[i, j] = k`, P_i request max `k` instance of R_j
3.`allocation[n][m]`; `allocation[i, j] = k`, P_i is allocated `k` of R_j
4.`need[n][m]`; `need[i,j] = k`, P_i may need `k` more R_j to complete i.e. `need[i, j] = max[i, j] – allocated[i, j]`

Safety Algorithm:
1.Initialize `work[m] = available`, `finish[n] = (false, n)`
2.Find an `i` such that both `finish[i] = false` and `need[i][j] ≤ work[j]` for all `j`; if no such `i` exists, proceed to step 4

3.Update `work[j] = work[j] + allocation[i, j]` for all `j`, `finish[i] = true`

4.If `finish[i] = true` for all `i`, system is in safe state; else, unsafe

Resource-Request Algorithm:

Each `i`, `request[m]`; if `request[j] = k`, P_i requires `k` instance of R_j

1.If `request[j]` ≥ `need[i, j]` for all `j`, raise error as exceed max

2.If `request[j]` ≥ `available[j]` for all `j`, process must wait

3.Pretend to allocate request resources to P_i by modifying:

a.Available = Available – Request$_i$

b.Allocation$_i$ = Allocation$_i$ + Request$_i$

c.Need$_i$ = Need$_i$ – Request$_i$

4.Safety algorithm; if safe allocate resource; else don't allocate

Chapter 9: Memory Allocation

Holes are contiguous block of free memory

Chunks are contiguous block of allocated memory

Memory Allocation:

1.Memory is requested and granted in contiguous blocks

a.`malloc` (to allocate memory) – finds sufficient contiguous memory, reserves the memory and returns the address of the first byte of the memory

b.`free` (memory becomes available for reallocation) – gives address of the first byte of memory to free

2.Ways of Memory Allocation:

a.Variable allocation size: allocates the exact size of request

b.Fixed allocation size: fixed allocation size determined by OS

Fragmentation:

Segment of memory is unusable after allocation and de-allocation

	Properties	Benefit
External	For variable allocation size memory Memory remains unallocated	Fragmented memory can be compacted later
Internal	For fixed allocation size memory Memory is allocated but unused	Less overhead to keep track of free memory (since memory are allocated in blocks)

Free List (For External Fragmentation)

1.Doubly linked list of free space; pointers build directly into holes

2.Prefer holes to be as large as possible:

a.Large holes can satisfy small requests

b.Less overhead in tracking memory

c.Faster search for available memory due to fewer holes

3.When memory is freed, place memory in free list and set next and previous pointers

4.Merge with holes before and/or after if possible

Allocation Algorithms:

	Properties	Comments
Best Fit	Pick smallest hole that satisfy request	Must search entire list (inefficient)
Worst Fit	Pick largest hole to satisfy request	Tends to leave lots of small hole (greater external fragmentation)
First Fit	Pick the first hole large enough to satisfy the request Always start from the first hole for when searching	Faster than best/worst fit Fragmentation issues like best fit
Next Fit	Pick the first hole large enough to satisfy the request Start search from where last search left off	Faster than first fit depending on arrival pattern of data

Note: No absolute solution to determine if best/worst fit is better

Compaction (for External Fragmentation)

Simple Solution: Move all allocated memory locations to one end and combine all holes on the other end to form a large hole

Problem: Tremendous overhead to copy data (system is frozen cannot serve other request); must find all pointer values (difficult)

Bitmap Allocation (For Fixed-Size Allocation)

Maintains a 1-bit array in the OS with each bit corresponds to the state of 1 block (allocation unit determined by OS)

Multiple Free Lists

1.Keep multiple lists of different hole sizes and take hole from a list that closely matches size of request ($\lceil \log_2 N \rceil$)

2.Start out with a single large hole and upon request, keep dividing hole by 2 until appropriate size is reached

a.Hole size is usually a power of 2

b.New hole is added to a different free list at every division

3.Once a hole is created, it cannot merge with another hole

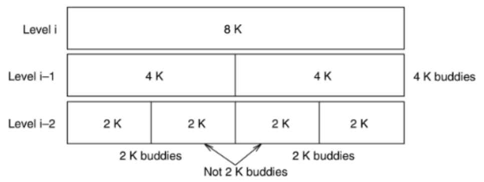
4.Request cannot be larger than maximum size of hole

5.Faster search time (log N) but lower (50%) utilization

Buddy System

1.Memory is sub-divided into a binary tree where each hole in a free list has a buddy (having the same parent)

2.Memory buddies can combine to form new holes twice the size that are aligned on proper boundary



3.When allocating memory, start from the list that closely matches size of hole ($\lceil \log_2 N \rceil$); if list is empty, go up one level, take a hole and break it into 2 before giving one to user

4.When freeing memory, if the chunk is returned and its buddy are in the free list, merge them and move the hole up one level

Chapter 10: Virtual Memory

Virtual Address Space: Set of N = 2^N virtual addresses

Physical Address Space: Set of M = 2^M physical addresses (where N > M, i.e. each byte in physical main memory has one physical address but can have 0 or more virtual addresses)

Virtual Memory:

1.Program refers to virtual memory address space

a.Memory is a very large array of byte with each byte having its own address

b.System provides address space "private" to process

2.Allocation of process memory by compiler and run-time system

a.Location different program address should be stored

b.All allocation should be within single virtual address space

3.Solves the following problem:

a.Fitting a huge memory into a tiny physical memory

b.Managing memory space of multiple processes

c.Protecting processes from interfering each other's memory

d.Allowing processes to share common parts of memory

Indirection:

1.Any problem in Computer Science can be solved by adding another level of indirection – flexible mapping between name and thing allows changing the thing without notifying the holder of the name

2.Each process gets its own private virtual address space, which is mapped to physical memory

Pages:

1.Virtual memory can be thought of as an array of N = 2^N contiguous bytes stored on a disk partitioned into number of blocks called pages (size of page P = 2^P bytes)

2.Physical main memory (DRAM) is used as a cache for some pages of the virtual memory array

3.Access to page ID with page number bits; access to individual bytes on the page with page offset bits

Status of Virtual Memory:

Unallocated: Page is not being used by the process

Uncached: Page is being used by the process; not cached in DRAM

Cached: Page is being used by the process; cached in DRAM

Page Size:

1.Page size is usually large as disk has a lower bandwidth (10,000x slower) than DRAM (slow for first byte, fast for consecutive)

2.Transfer between disk and DRAM through DMA

3.Larger page size has the advantages of fewer page faults (spatial locality), page table can be smaller and fewer TLB misses; but page faults will also be more expensive and wasted space has pages are under-utilized

Page Tables:

1.Page table is an array of page table entries that maps virtual pages to physical pages

2.One page table per process, stored in the main memory DRAM

3.Page table should have one page table entry corresponding to each virtual page (i.e. 256 virtual page requires 256 PTE)

4.Key fields in each PTE are a bit that stores the status of corresponding virtual page (0 for uncached, 1 for cached) and physical page number, if virtual page is cached in DRAM

5.In most cases, MMU performs address translation on its own without software assistance

Page Hit:

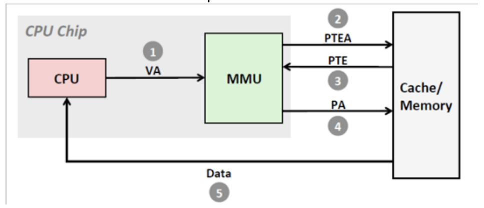
Reference to virtual memory byte that is in physical memory

1.Processor sends virtual address to MMU

2.MMU fetches PTE from the page table stored in DRAM

3.MMU sends physical address to memory to DRAM

4.DRAM sends data word to processor



Page Fault:

Reference to virtual memory byte that is not in physical memory

1.Processor sends virtual address to MMU

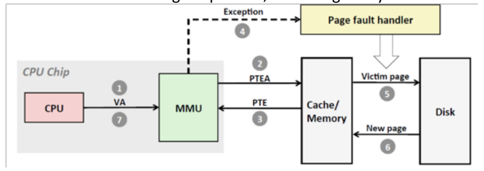
2.MMU fetches PTE from the page table stored in DRAM

3.Valid bit fetched is 0, so MMU triggers page fault exception

4.Handler identifies victim (and if dirty, pages it out to disk)

5.Handler pages in new page and updates PTE in memory

6.Handler returns to original process, restarting faulty instruction



Overhead of Page Faults: 1. I/O overhead from loading of page from disk to DRAM, writing dirty page to disk; 2. Execution overhead when executing page fault ISR

Translation Speed:

1.MMU accesses memory at least twice: once to get the PTE for translation and another to get actual memory request from CPU

2.PTE may be cached in L1 cache but they may be evicted by other data references, and hit in L1 cache still requires 1-3 cycles

3. To speed up translation, add another cache known as Translation Lookaside Buffer (TLB) to eliminate overhead:

a.Small hardware cache inside MMU that maps virtual page numbers to physical page numbers

b.Contains PTEs for small number of virtual pages

Calculation:

1.Virtual address space bit = OS address length or $\log_2$(VAS in byte)

2.Physical address space bit = $\log_2$(RAM size in byte)

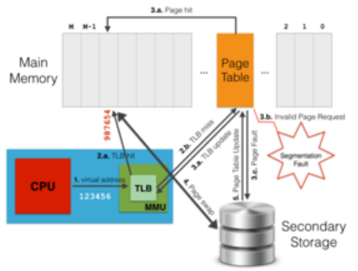
3.Offset bit = $\log_2$(page size in bytes)

4.Number of virtual pages = VAS/page size in byte

5.Number of physical pages = RAM size in byte/page size in byte

6.Virtual page number bits = virtual address – offset or $\log_2$(Number of virtual pages)

7.Physical page number bits = physical address – offset or $\log_2$(Number of physical pages)



Chapter 4: Task Management

Process in Memory:

1.Stack: stores parameters and local variables in a function

2.Heap: stores dynamically allocated variable (eg. malloc)

3.Data: stores global parameters

4.Text: Stores code section, program

Process Control Block:

1.Pointer: Locate next PCB in queue

2.Process State: Maintain state of process

3.Process ID: Identifier for each process

4.CPU Registers: Stores temporary data in CPU cache

5.Program Counter: Pointer to the next instruction

6.Memory Management Information: Starting and ending point in the memory structure (memory pointer)

7.Information of Open Files

PCBs are stored in main memory for security.

Process State: Created

•Immediately after creation with valid PCB

•OS has initialized the PCB of the process

•Process only created if exist minimum resources to support it

•Does not compete CPU with other processes for execution yet

Process State: Ready

•Process is willing and competes with other processes to execute

•No processor available to execute it yet

•Admission control of OS enables transition from Created to Ready

Process State: Running

•Process is being actively executed by a processor

•As processor can only have one running process at any moment, Scheduler of OS which is transparent to the process enables transition from Ready to Running

Process State: Block

•Process is waiting for an event i.e. completion of I/O operation, synchronization/communication with other process

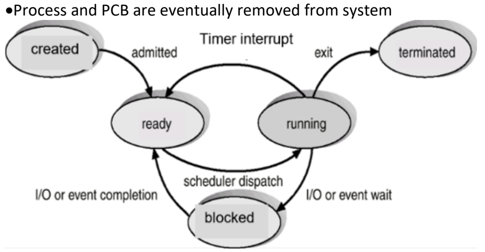
•Process does not compete for execution in this state

Process State: Terminated

•Process has completed execution and can no longer be executed

•PCB is still available for other processes to retrieve/examine information in PCB (eg. retrieval of exception handling)

•Process and PCB are eventually removed from system


created
admitted
Timer interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
blocked